UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JULIO TOSS

Parallel Algorithms and Data Structures for Interactive Applications

Thesis prepared in a co-tutelle aggreement
and presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor (UFRGS):
Prof. Dr. João Luiz Dhil Comba
Co-tutelle advisor (UGA):
Prof. Dr. Bruno Raffin

Porto Alegre
August 2017

CIP — CATALOGING-IN-PUBLICATION

Toss, Julio

Parallel Algorithms and Data Structures for Interactive Applications / Julio Toss. – Porto Alegre: PPGC da UFRGS, 2017.

83 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: João Luiz Dhil Comba; Coadvisor: Bruno Raffin.

I. Dhil Comba, João Luiz. II. Raffin, Bruno. III. Título.

ABSTRACT

The quest for performance has been a constant through the history of computing systems. It has been more than a decade now since the sequential processing model had shown its first signs of exhaustion to keep performance improvements. Walls to the sequential computation pushed a paradigm shift and established the parallel processing as the standard in modern computing systems. With the widespread adoption of parallel computers, many algorithms and applications have been ported to fit these new architectures. However, in unconventional applications, with interactivity and real-time requirements, achieving efficient parallelizations is still a major challenge.

Real-time performance requirement shows up, for instance, in user-interactive simulations where the system must be able to react to the user's input within a computation time-step of the simulation loop. The same kind of constraint appears in streaming data monitoring applications. For instance, when an external source of data, such as traffic sensors or social media posts, provides a continuous flow of information to be consumed by an online analysis system. The consumer system has to keep a controlled memory budget and deliver a fast processed information about the stream.

Common optimizations relying on pre-computed models or static index of data are not possible in these highly dynamic scenarios. The dynamic nature of the data brings up several performance issues originated from the problem decomposition for parallel processing and from the data locality maintenance for efficient cache utilization.

In this thesis we address data-dependent problems on two different applications: one on physically based simulations and another on streaming data analysis. To deal with the simulation problem, we present a parallel GPU algorithm for computing multiple shortest paths and Voronoi diagrams on a grid-like graph. Our contribution to the streaming data analysis problem is a parallelizable data structure, based on packed memory arrays, for indexing dynamic geo-located data while keeping good memory locality.

Keywords: Parallel processing. data locality. stream processing. real-time processing. physically based simulation.

Algoritmos Paralelos e Estruturas de Dados para Aplicações Interativas

RESUMO

A busca por desempenho tem sido uma constante na história dos sistemas computacionais. Ha mais de uma década, o modelo de processamento sequencial já mostrava seus primeiro sinais de exaustão pare suprir a crescente exigência por performance. Houveram "barreiras"para a computação sequencial que levaram a uma mudança de paradigma e estabeleceram o processamento paralelo como padrão nos sistemas computacionais modernos. Com a adoção generalizada de computadores paralelos, novos algoritmos foram desenvolvidos e aplicações reprojetadas para se adequar às características dessas novas arquiteturas. No entanto, em aplicações menos convencionais, com características de interatividade e tempo real, alcançar paralelizações eficientes ainda representa um grande desafio.

O requisito por desempenho de tempo real apresenta-se, por exemplo, em simulações interativas onde o sistema deve ser capaz de reagir às entradas do usuário dentro do tempo de uma iteração da simulação. O mesmo tipo de exigência aparece em aplicações de monitoramento de fluxos contínuos de dados (*streams*). Por exemplo, quando dados provenientes de sensores de tráfego ou postagens em redes sociais são produzidos em fluxo contínuo, o sistema de análise on-line deve ser capaz de processar essas informações em tempo real e ao mesmo tempo manter um consumo de memória controlada.

A natureza dinâmica desses dados traz diversos problemas de performance, tais como a decomposição do problema para processamento em paralelo e a manutenção da localidade de dados para uma utilização eficiente da memória cache. As estratégias de otimização tradicionais, que dependem de modelos pré-computados ou de índices estáticos sobre os dados, não atendem às exigências de performance necessárias nesses cenários.

Nesta tese, abordamos os problemas dependentes de dados em dois contextos diferentes: um na área de simulações baseada em física e outro em análise de dados em fluxo contínuo. Para o problema de simulação, apresentamos um algoritmo paralelo, em GPU, para computar múltiplos caminhos mínimos e diagramas de Voronoi em um grafo com topologia de grade. Para o problema de análise de fluxos de dados, apresentamos uma estrutura de dados paralelizável, baseada em *Packed Memory Arrays*, para indexar dados dinâmicos geo-localizados ao passo que mantém uma boa localidade de memória.

Palavras-chave: algoritmos paralelos, localidade de dados, processamento de fluxo de dados, processamento em tempo real, simulação baseada em física.

Algorithmes et Structures de Données Parallèles pour Applications Interactives

RÉSUMÉ

La quête de performance a été une constante à travers l'histoire des systèmes informatiques. Il y a plus d'une décennie maintenant, le modèle de traitement séquentiel montrait ses premiers signes d'épuisement pour satisfaire les exigences de performance. Les barrières du calcul séquentiel ont poussé à un changement de paradigme et ont établi le traitement parallèle comme standard dans les systèmes informatiques modernes. Avec l'adoption généralisée d'ordinateurs parallèles, de nombreux algorithmes et applications ont été développés pour s'adapter à ces nouvelles architectures. Cependant, dans des applications non conventionnelles, avec des exigences d'interactivité et de temps réel, la parallélisation efficace est encore un défi majeur.

L'exigence de performance en temps réel apparaît, par exemple, dans les simulations interactives où le système doit prendre en compte l'entrée de l'utilisateur dans une itération de calcul de la boucle de simulation. Le même type de contrainte apparaît dans les applications d'analyse de données en continu. Par exemple, lorsque des donnes issues de capteurs de trafic ou de messages de réseaux sociaux sont produites en flux continu, le système d'analyse doit être capable de traiter ces données à la volée rapidement sur ce flux tout en conservant un budget de mémoire contrôlé.

La caractéristique dynamique des données soulève plusieurs problèmes de performance tel que la décomposition du problème pour le traitement en parallèle et la maintenance de la localité mémoire pour une utilisation efficace du cache. Les optimisations classiques qui reposent sur des modèles pré-calculés ou sur l'indexation statique des données ne conduisent pas aux performances souhaitées.

Dans cette thèse, nous abordons les problèmes dépendants de données sur deux applications différentes : la première dans le domaine de la simulation physique interactive et la seconde sur l'analyse des données en continu. Pour le problème de simulation, nous présentons un algorithme GPU parallèle pour calculer les multiples plus courts chemins et des diagrammes de Voronoi sur un graphe en forme de grille. Pour le problème d'analyse de données en continu, nous présentons une structure de données parallélisable, basée sur des *Packed Memory Arrays*, pour indexer des données dynamiques géo-référencées tout en conservant une bonne localité de mémoire.

Mots clés: algorithmes parallèles, localité de données, traitement de flux de données, traitement en temps réel, simulation physique.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

Since the invention of early computers, we have seen an impressive growth in the amount of applications and problems that could be solved by these machines. The role of computers have nowadays by far surpassed the applications thought by their inventors. A long road has been traveled, from the first calculation machines up to the modern supercomputers. Practical limits of what we can do with computers are still unknown. For every technology improvement made to the computing systems, new possibilities and challenges arise pushing the frontier of what can be done on these pieces of silicon. At the same time, scientists have created a kind of "chicken-egg" problem. For every new improvement made to current computing systems, larger problems and more demanding applications require the development of a new generation of computers. The quest for performance has been a constant through the computing history.

Amongst the fascinating possibilities, computers can simulate our world, capture information from it, process the data and output it back to the world in a glimpse of eyes. For a large class of applications, performance is closely related to the reactivity of the system, the ability to process information within a negligible time. Real-time processing for example happens in interactive simulation systems where the user can interact with a virtual model of the world without experiencing unrealistic delays. For instance, such kind of systems is common in medical simulation applications. The realism and complexity of the model to simulate is limited by the real-time processing power that the computing system is able to deliver.

Not only user-interactive systems require real-time processing. In big-data systems, for instance, sensors from road networks or even social media on the web generate massive amounts of data every second. On-line data processing systems must be able to process the continuous stream of data to produce valuable information in real-time. In such cases, the maximum affordable computing performance limits the amount of information that can be digested on-the-fly.

Until recently, solutions to the performance requirements of these problems could rely on the sequential processing paradigm backed by a constant improvement on the sequential hardware architectures. This model of computing has now reached a point of exhaustion and is not able anymore to continue delivering the same growth in processing speed. Pushed by physics limitations, the computing paradigm made a shift to the parallel processing model. Nowadays, and until quantum computers become viable, any hope for performance to keep the pace with the continuous increase of problems scale passes through the parallel computing paradigm.

Parallel processing capabilities are nowadays widespread and available in every computing device whatever its scale. Current operating systems already do a good job in scheduling different sequential applications to run concurrently on multicore processors. However the OS alone can't provide individual applications. Application performance depend on how well the problem can be decomposed into independent tasks to benefit from parallel architectures. Generally, effective parallelization depends on two main aspects:

Computation decomposition: how the amount of computation can be evenly distributed among all the processing cores;

Memory access and locality: How to keep data close to the cores to reduce access latency, and how to maintain data locality during an interactive computation.

Some applications are trivially parallelizable, like on image processing, where filters and transformations can be applied independently on each pixel in a data parallel style, also known as SIMD (Single Instruction Multiple Data). These problems are characterized by very limited data-dependency and by a balanced amount of computation of each piece of data, which make it simple to split and schedule for parallel execution. The computation to be executed and corresponding data are known since the beginning of the execution, allowing to optimize the data partitioning and scheduling on processors.

Unfortunately for the majority of applications, data partitioning and computation parallelization are not straightforward. In the case of interactive computation, like user-interactive physically based simulations or on-line streaming processing, data and computation are subject to the interactions of an external agent hard to predict. The challenges presented on these domains require special data-structures and algorithms capable of adapting to the dynamic nature of the problems while still keeping good properties of memory locality and computational load balance.

In this research we tackle the challenges of parallelizing and keeping data locality in unconventional problems like interactive simulation and streaming data processing. Each problem has its particular characteristics but share a similar performance issue that has its roots on how to deal with data dependent computations dynamically. We deal with the *performance* problem of these two applications. To address it we propose two techniques, one targeting graphics parallel processors (GPUs) and another focusing in a dynamic data organization structure to maintain locality of references.

1.1 Outline

The core of this document is organized in chapters according to the contributions proposed:

- The first problem is related to physically based simulations: it involves a user-interactive system that changes the state of the data in a virtual model. Challenges on how to update data to keep the real-time execution of the simulation are addressed by parallelizing the underlying data structures on the GPU. We present this problem and the associated contributions in Chapter 2.
- The second one is on *Big* and *Fast* data processing: the real-time requirement in this case comes from the need of a data exploration system to keep the pace with a continuous real-time data stream. We propose a data-structure that stores the incoming information and is able to keep a good memory organization favoring data locality and speed-up queries on the data. The contributions are presented in Chapter 3.

Additionally, before diving into the specific contribution topics, we provide further context that motivates our work on Section 1.2. Finally, the last chapter provides final remarks and perspectives for possible continuations of this work (Chapter 4).

1.2 General Context And Motivations

In this work we face problems of how to enhance the computation of data dependent problems. In the context of physically based simulations, we need to compute values based on neighborhood data. As we will see later in Section 2.3.3, the arithmetic operations executed over them are simple and represent a much lower cost than accessing the data itself. Keeping data locality during the simulation is paramount.

One main characteristic of this problem is that with the right underlying data-structure we are able to expose the data parallelism underneath it. This leads us to propose efficient solutions based on the SIMT (Single Instruction Multiple Threads) paradigm of modern massively parallel architectures (GPUs). However optimizing load balance and occupation of these data-dependent programs still represents a major challenge. Effectively placing data in memory to speed-up locality access is a major concern to keep all the processors busy.

At the same time, in the context of interactive and streamed data problems, the volume of data to process can change during execution and the pre-computed data dependency relations can be modified in unpredictable ways. Exploration of large datasets requires structure and organization of data to reduce memory access latency. Maintaining this organization when the dataset is dynamic and when the system has to respect real-time constraints is a challenging task. We refer to this problems as *dynamic-data problems*.

In the next sections we discuss the more fundamental problem of keeping the computational efficiency on dynamic-data problems. To understand the role that memory plays in computation and how historically computers turned into the parallel processing paradigm, we start with a brief review of how memory and computation capacities evolved over the past years and what are the boundaries they are facing. Finally we also present the current perspectives towards the era of exascale computing and big data problems that motivates our contribution on the development of new dynamic data structures for parallel machines.

1.2.1 Walls to Sequential Computation

The evolution from the sequential programming paradigm to the current parallel paradigm happened relatively recently in the past two decades (MCCOOL et al., 2012). The idea of parallel computation however is much older that this. There were factors that pushed a shift in the computing paradigm that are commonly know as the "three walls to sequential computation":

The Power wall refers to unacceptable increases in power consumption of the chip with the increase of processor clock rates. The maximum frequency at which a processor can operate has reached a limit and is now generally around 3Ghz. With the Moore's law still holding true, the industry kept increasing the amount of transistors per unit area, which led to the design of multicore chips.

The Instruction-level parallelism (ILP) wall refers to a set of parallel techniques used at chip level that for some years allowed to speed-up sequential programs. Techniques like instruction pipelining, superescalar execution, out-of-order execution, register renaming, branch prediction, and speculative execution have fulfilled to a very large extent their potential.

The Memory wall is the consequence of several years of improvements in CPU processing speed that were not followed at the same rate by improvements in the memory speed. The memory and communication problems are related to two "speeds": *latency* and *bandwidth* . Latency is the time spent for starting a transaction. Once the transfer has started the bandwidth is the rate at which data is delivered at the destination. The use of larger caches allowed to temporarily cope with these problems, however, latency is difficult to decrease due to fundamental physical limits such as the speed-of-light. On sequential processing streams the latency for a memory access could not be hidden anymore causing the processor to stall waiting for data. Bandwidth on the other hand is easier to improve by increasing the number of lanes and sending big chunks of data at a time. This favored parallel executions for example in the SIMD computation paradigm.

1.2.2 Big Data and Exascale Computing

In the scientific and engineering community, *exascale* computing refers to computing systems able to process 10^{18} operations per second. To achieve this goal, industry and scientific community still have several challenges to tackle (REED; DONGARRA, 2015).

Big data is one of the main use cases that will benefit from exascale computing systems. Processing the enormous quantity of data generated today poses several challenges in different fields. In the exascale regime, the energy cost to move a unit of data will exceed the cost of a floating-point operation (REED; DONGARRA, 2015). Improvements on the hardware will continue to enhance memory features. New advanced memory technologies will provide large capacities and a high performance interconnect will provide energy efficient, low-latency and high-bandwidth data exchange among hundreds of thousands of processors.

However, hardware improvements alone will fail to provide application speedups. At the software level there is still a need for locality aware algorithms to improve computation efficiency and reduce energy needs.

1.2.3 In-memory Big Data Processing

Big data has become widespread in industry with several systems being designed to address a wide range of problems (ZHANG et al., 2015). More recently, with the current improvements in memory technologies like Non Volatile Memories (e.g. SSD) and larger capacities of DRAMs, traditional disk-base database systems have been shifting to a heavier use of in-memory storage and processing.

We may differentiate these data processing systems into "classes" according to the *structure* and *dynamicity* of data. *Data-processing* can be well structured like relational databases, semi-structured like graph processing, or completely unstructured as in texts. The *dynamicity* refers to the amount of insertion, deletion and update operations that are applied on the data. Such datasets can vary from a static set of records with regular updates with low insertion ratio, up to completely dynamic elements as in real-time streams of data. Each scenario requires different memory optimization techniques and often presents a trade-off between efficient indexing and querying of records. In this work we will focus on the problem of keeping an *efficient update* ratio on dynamic data while keeping good *access performance* by favoring spatial locality.

1.2.4 A Use Case Problem: Interactive Exploration of Geo-Located Data Streams.

Several business applications rely on stream processing to get real-time knowledge about events currently happening and take fast decisions, such as in the stock market, sensor networks or in social networks. These systems often employ Complex Event Processing (CEP) to identify patterns of events in the stream and if needed triggers the appropriate alarm. The main characteristic of this stream processing is that historic data is usually not stored due to the high unbounded amount of data that a stream generates. This comes with the drawback that we must know in advance what kind of event we are expecting to setup a continuous query that will filter the *incoming* stream. Once an automatic alarm is triggered we have lost access to the previous historic raw data, which could have been useful for a more detailed analysis by a specialist.

For instance, a system that monitors microblogs streams such as twitter could detect natural disaster and point the location where humanitarian aids should be deployed with priority (SAKAKI et al., 2010). While storing full history of tweets would be unpractical and out of purpose, having a significant time window of past tweets stored about the area of interest would enable a more detailed analysis after a natural disaster alarm is triggered.

The real-time nature of such systems requires data to be resident in memory. This not only allows fast answer time of exploratory queries but also keeps a high rate of insertions and index update capable of processing the incoming data stream. Furthermore, geo-spatial data must be indexed and stored in a way to provide fast access time. While several types of spatial-indices exist, it is important in real-time systems to avoid expensive updates and physical reorganization of data in memory.

2 A PROBLEM ON PHYSICALLY BASED SIMULATIONS

2.1 Context

Physically based simulations are widely used in several domains of applications. Digital animation, as used in films and games, rely on physical models to try to reproduce their behavior as realistically as possible to achieve the best visual effects. Physical models, traditionally used in scientific computing and visualization domains, are also largely applied in the engineering industry. Here, the use of virtual models can help to anticipate problems in the project of a new product still in the earliest stages of conception, before going to production. For example, the automobile industry uses physically based simulations on virtual car models to perform crash tests, which permits reducing the number of expensive real crash tests. More recently, another growing field using this kind of simulation is that of medical applications such as computer-assisted surgery. In this case, a physically based simulation can be used for planning and training surgical procedures reducing costs and avoiding complications during the real surgery.

On the other hand, the algorithms used in physically based simulations usually face a compromise between speed and accuracy (COURTECUISSE, 2011). This led us to analyze these application fields with different requirements. Computer games for example have a strong requirement for speed as it has to be done in real-time while accuracy just needs to be approximated using a visually correct simulation. In contrast, virtual crash tests simulations, for example, require very accurate results but can be done offline. Surgery simulation is, perhaps, the most challenging field. Here, we have an equal interest for efficiency, to enable real-time, and accuracy to get valid simulation results that corresponds to the actual surgery on a human body.

Many methods have been proposed over the past decades for simulation of deformable objects. A good survey on physically based deformable objects can be found in (NEALEN et al., 2006). A common problem of these methods is the sampling size used. Accuracy of simulation greatly depends on how many samples are used in the model. This is a major problem when we need to simulate deformations of a complex object composed of many different materials with various degrees of stiffness. Softer regions would require higher sampling than stiffer ones and would incur in higher computation costs.

In (FAURE et al., 2011) a new frame-based method is proposed to simulate deformations on objects of heterogeneous materials. Their method uses meshless models with sparse sampling combined with specific functions, called *shape functions*, to interpolate deformations within the *simulation nodes*. Their major contribution was the use of a novel material-aware shape function that computes distances based on compliance. This approach allows to have fewer simulation nodes and still capture the most relevant deformation modes to achieve good realism. Computation of shape functions and placement of simulation nodes are done at initialization steps. After initialization, the simulation can efficiently run in real-time at interactive frame rates. However, if any topology modification is applied to the object during the simulation, the initialization step needs to be recomputed. The overhead added by on-line modifications of the model is prohibitive for an interactive simulation. For example, if we simulate a cut of an organ during a virtual surgery, the whole distribution of simulation nodes and shape functions has to be recomputed on-the-fly. Therefore, a major concern of this method is to speed up the computation of shape functions to allow real-time simulations.

At the same time, the hardware technology of modern computers has evolved considerably. Over the last decade, parallel computers have become ubiquitous and it is now usual to have multiple processors in the same computer. Massively parallel hardware such as Graphics Processing Units (GPU) are becoming commodity processors and can easily be found in general desktop computers. This clear trend of increasing the core count of modern processors has direct implications at the software and programming levels. Efficiently programming applications for this kind of platform requires an extra amount of effort from programmers. Algorithms and data structures have to be carefully designed to harvest the benefits of parallelism. As a consequence, new parallel languages and programming tools have been developed to help on such tasks.

In this chapter we look into the method of sparse meshless simulation proposed by Faure et al. (2011) and study a parallel approach to the costly initialization steps where the shape functions are computed. Material-aware shape function computation involves computing a special kind of Voronoi diagram on a graph with grid topology. We will abstract the concepts from the physically based simulation domain and focus on the algorithms. As we will see, the underlying problem is strongly related to the Multiple-Source Shortest Path (MSSP) problem.

In the next sections we will present in more details how shape functions are computed and what kind of algorithms are used (Section 2.2). In Section 2.3 we review the related works about parallel computation of Voronoi diagrams and shortest-paths.

2.2 Background

2.2.1 Sofa - Simulation Framework

The work described in this chapter is part of the real-time simulation framework called SOFA (2017). It results from a cooperation work with the IMAGINE research team[1] from Inria-Alpes, one of the main contributors to the development of SOFA.

The Simulation Open Framework Architecture (SOFA, 2017) seeks to reduce complexity in interactive physically based simulations by providing a well-defined common interface for physical algorithms. Its goal is to improve research collaborations, allowing to reuse and easily compare a variety of available methods. Although the primary target application of SOFA is medical simulation such as in virtual surgery, applications from different domains also rely on this framework. Examples include motion planning and control in robotics simulations (LARGILLIERE et al., 2015; RODRIGUEZ et al., 2017), augmented reality in deformable objects (PAULUS et al., 2015) and virtual immersion environments (PETIT et al., 2009).

The current SOFA implementation targets execution on a single machine, with some extensions allowing parallel applications on multicore and GPUs, but not on clusters or supercomputers architectures. The use of parallelism in this framework is highly desirable as a way of enhancing the performance of its simulation algorithms. The contribution we bring to the SOFA project is a new parallel algorithm for Voronoi-shape functions computation.

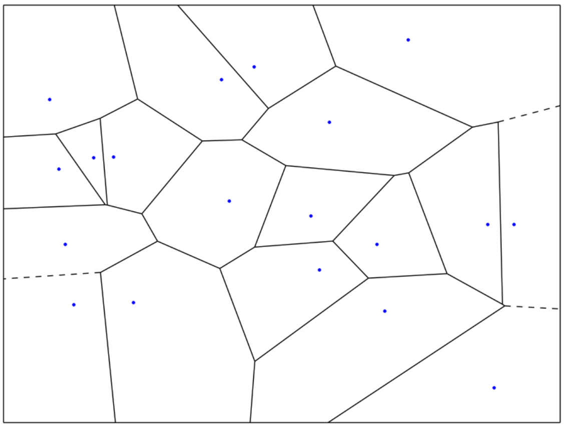
2.2.2 Voronoi Diagrams

The Voronoi diagram is a data structure extensively studied in the context of computational geometry for many different applications, but most of the time on a contiguous Euclidean space (RONG et al., 2011) or computing its discrete approximation (HOFF et al., 1999; RONG; TAN, 2006). Originally the Voronoi diagram (Figure 2.1) was defined for a set of *seed* points $P = \{p_1, p_2, ..., p_n\}$ as the partitioning of the space into n cells such that every seed p_k is the closest one to any other point enclosed in the cell k, according to a distance metric (usually Euclidean distance)(BERG et al., 2008).

Another particular type of Voronoi diagram is the *Centroidal Voronoi Diagram*, also known as Centroidal Voronoi Tessellation (CVT). In this case, the seeds must be the

[1] <https://team.inria.fr/imagine/>

Figure 2.1: Voronoi diagram in Euclidean space for a given set of seeds (blue dots).



centroids of their cells. The computation of a CVT is usually done using the Lloyd's algorithm. In this iterative method, seeds are moved to the current center of each cell and the Voronoi diagram is recomputed at each iteration until convergence.

Although most works found on the literature study Voronoi diagrams on the Euclidean space, there are situations where a distance is defined using other metrics, such as the distance between vertices in a graph (ERWIG, 2000; HURTADO et al., 2004). In this context, the distance metric considered corresponds to the shortest path (also referred as geodesic) between two vertices. This formulation of Voronoi diagrams arises, for instance, in the problem of facility location, where clients and suppliers lie in an interconnection network. Computing these Voronoi diagrams basically consists in computing the shortest paths on a weighted graph.

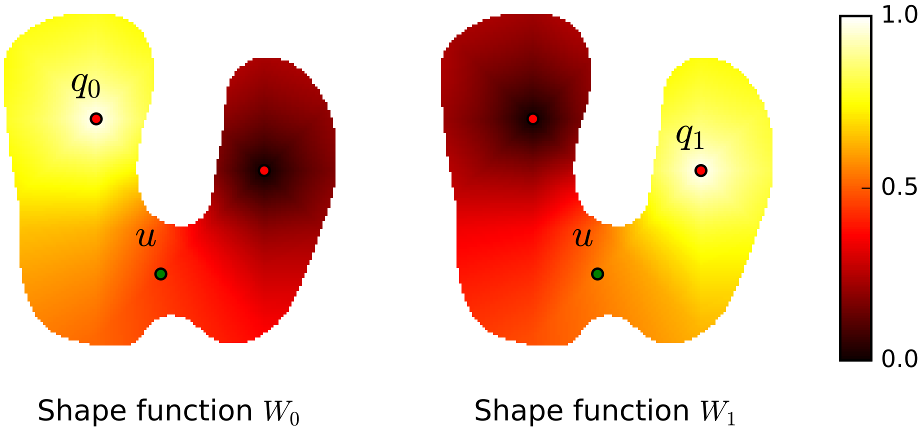
The *Parallel Dijkstra* algorithm proposed by Erwig (2000) is a variant to the Dijkstra algorithm for computing the Voronoi diagram on directed weighted graphs. The term "Parallel" in this algorithm refers to the fact that the shortest-path trees, rooted at each Voronoi seed, grow rather simultaneously, although the algorithm is still sequential.

2.2.3 *Shape Functions* in Numerical Simulation

In computer graphics, the numerical simulation of continuous deformable objects is based on a set of independent control points called *Degrees of Freedom (DoFs)*. For instance, in the popular Finite Element Method (FEM), the DoFs are the vertices of the mesh representing the discretized model.

In this work we deal with another particular class of methods for modeling deformable objects know as *meshless* models. In particular, in the *meshless frame-based* models proposed by Gilles et al. (2011), the simulation does not use any underlying struc-

Figure 2.2: Example of an object with two Degrees of Freedom (q_0 and q_1) and their corresponding shape functions (W_0 and W_1). Displacement of a point u is the weighted sum of sampled displacements at q_0 and q_1.

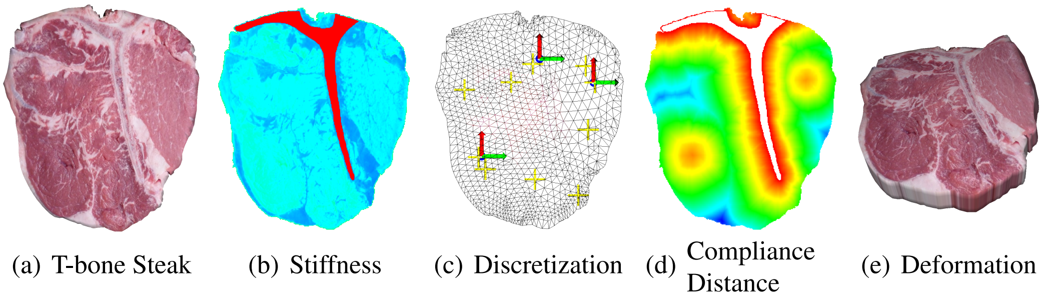


ture like a mesh. In this case, the DoFs are unstructured control points called *frames* to which are associated *shape functions*. For each control point of the model there is a corresponding *shape function* that defines a region of influence. The design of theses shape functions plays a central role in the simulation. It will determine how the displacements captured at different control points will be combined to result in the final deformation of the object.

Consider the following steps when simulating the deformation on an object subjected to some forces. The displacement of the object is sampled at the control points (DoFs). Each DoF and associated shape function define where and how other points in the object are influenced. This region of influence is commonly referred as the *support* of the shape function, i.e. the region of the object where the function is defined. The evaluation of the *shape function* at any point within the support results in a normalized *weight* that encodes the influence of the corresponding DoF at that point. In general, these weights decrease with the distance to the control point. For instance, in Figure 2.2 there are two control points in the object (red dots). The shape function of each one is defined everywhere within the object boundaries. Their normalized weights are shown by a heatmap and varies from 1, at each respective frame's location, down to zero at the coordinates of the other frames. The displacement at any point u (green dot), parameterized with coordinates x in the object, is therefore computed by a weighted sum of the displacements sampled at the DoFs (Equation 2.1).

$$u(x) = w_0(x) * u_0 + w_1(x)u_1 \tag{2.1}$$

Figure 2.3: A frame-based method to simulate complex deformable solids composed with heterogeneous material properties.


(a) T-bone Steak (b) Stiffness (c) Discretization (d) Compliance Distance (e) Deformation

Source: (FAURE et al., 2011)

Finally, it remains the problem of defining how weights should be computed for a given placement of control nodes. These weights describe how shape functions from different DoFs are blended on the rest of the domain. In (FAURE et al., 2011), these weights are computed using a novel method based on Voronoi diagrams, called the *Voronoi Shape Functions*. We conceptually explain their method in the next section. A more detailed discussion on shape functions is deferred to Section 2.5 where we describe different schemes for interpolating node values using the Voronoi diagram generated from the simulation nodes.

2.2.4 Material-Aware Shape Functions

This section briefly introduces the frame-based simulation method presented in (FAURE et al., 2011), which is the motivation problem of this chapter.

Shape functions are a fundamental part in the simulation of deformable solids. They are used to compute the displacement in the object within the simulation nodes. The displacement applied to simulation nodes is combined according to their shape functions (also called *weights)* and is interpolated inside the deformed object.

A material-aware shape function takes into account the material properties, such as stiffness, to compute the displacement. This is particularly interesting in objects composed with various types of materials, with different stiffness, as these regions will not deform in the same way or with the same intensity. For example, a steak composed of bones, fat and flesh will present different degrees of deformation in each material region (Figure 2.3).

The input data for this method is given as a 3D voxel map of material properties and a quantity of simulation nodes according to the expected execution time. The problem consists in defining the placement of nodes and the shape function associated. A major challenge here is to define a proper region of influence of each shape function. For example, consider the steak object from Figure 2.3, a deformation applied to a node on the left side of the rigid bone should not be propagated to the other side.

To limit this kind of unrealistic behaviour, the authors create Voronoi partitions of the voxelized material property map. Similarly, the problem of placement of simulation nodes can be approached as the *Centroidal Voronoi Diagram* computation on a weighted graph. The distances used to compute the Voronoi diagrams are based on the values of compliance (inverse of stiffness) of each voxel. The *compliance distance*, as referred in their work, is the length (compliance) of the shortest (stiffest) path from one point to the other.

We highlight some specific features in this problem that differs from a general Voronoi computation. First of all, it should be noted that the Voronoi diagram here is not computed on the Euclidean space. This means that distances between any two points cannot be computed directly. Instead, we have a geodesic on a 3D discrete voxel map where each voxel knows the local distances (compliance distance computed from the material property map) in its 26-neighborhood. Each voxel is mapped to a vertex of the graph and is connected with the 26 vertices corresponding to the adjacent voxels in a 3D image [2]. Finally, edges are weighted according to the compliance value of the pair of adjacent voxels.

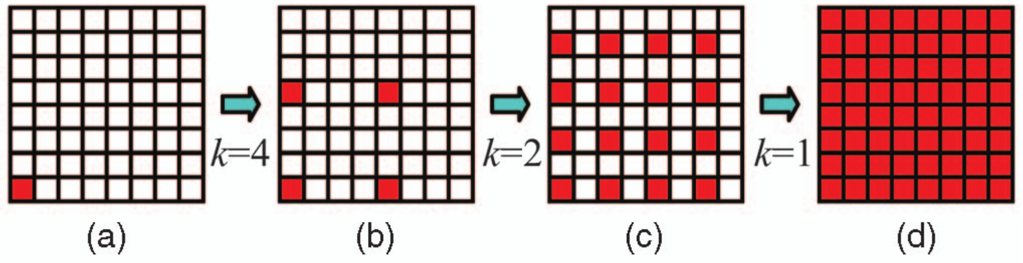
2.3 Related Work

2.3.1 Parallel Voronoi Computations

The parallelization of Voronoi diagram computations has been studied through several works in literature with a lot of different approaches. Early studies used graphics hardware to compute an approximation of these diagrams using the OpenGL graphics library, before the proposal of the CUDA or OpenCL architecture (HOFF et al., 1999). For most works found in the literature, a discrete approximation of the Voronoi diagram is sufficient to fulfill the precision requirements of its applications. Approximations can be either computed by the Euclidean distance on 2D pixel-maps and in 3D surfaces using the Euclidean distance between 3D-coordinates of sampled points over the surface (RONG et al., 2011).

[2] The 26-neighborhood results from voxels sharing faces (6), edges (12) or corners (8).

Figure 2.4: JFA: starts propagating the information from the seed (bottom left corner). At each step, the neighbours with coordinates offset k are reached and start propagating the data about the seed.



Source: (RONG et al., 2011)

The many parallel approaches vary in the way the information of proximity from the Voronoi centers is propagated to each pixel.

Rong and Tan (2006) propose *jump flooding* (JFA) as an algorithmic paradigm for GPGPU and show its application on Voronoi diagram computation. In JFA, the seeds start propagating their coordinates to neighbour pixels according to a pattern that halves the offset at each step (Figure 2.4). Each pixel compares the new information received with the previous one and keep the coordinates of the closest seed. In this case, distances can easily be computed on the Euclidean plane.

Weber et al. (2008) introduce an interesting parallel algorithm called *parallel marching method (PMM)* to compute distances on surfaces with application on Voronoi diagrams. This method is indeed an extension of the fast marching method, which is based on a priority-queue. However, this kind of data structure is difficult to be efficiently parallelized. Instead, their method replaces it by using a specific traversing order of the grid, called *raster scan*, and show an efficient parallelization algorithm.

Reem (2012) proposed a substantially different method from previous ones. It uses a combinatorial approach to compute the exact polygons that form each cell of the Voronoi diagram in parallel. This work, however, is mainly theoretical, focusing on formal proofs of correctness and complexity analysis while leaving aside the implementation issues and experimental results.

All of these works only consider distance computation on the Euclidean space. None of them were found to deal with Voronoi diagrams on graph space. In those contexts the shortest distance is always a straight line, hence these methods cannot be directly applied on graph problems. Nonetheless, Weber et al. (2008) show that PMM can also be used to compute distance on curved domains by repeating N_{iter} iterations of raster scans, where N_{iter} is a data-dependent bound.

2.3.2 The Shortest Path Computation

Clearly, the shortest-path problem is an essential building-block for Voronoi diagram computation on graphs. Efficiently finding shortest paths on graphs is a well known problem widely studied in the literature for a plethora of applications. Probably the best known sequential algorithm for Single-Source Shortest-Path (SSSP) was developed by Dijkstra. The original Dijkstra's algorithm has $O(V^2)$ complexity on the number of vertices, while the min-priority-queue based version shows $O(E + VlogV)$ where E is the number of edges (CORMEN et al., 2009).

Naturally, the use of parallel processors to compute shortest paths has attracted a lot of attention from researchers. Early implementation of a parallel Dijkstra's SSSP is reported in (CRAUSER et al., 1998). However, this is an inherently sequential algorithm with lots of synchronizations leaving no possibility for an efficient PRAM implementation (HARISH et al., 2009). As an alternative, Kumar et al. (2011), worked on parallel SSSP based on Bellman-Ford's algorithm, which is less efficient than Dijkstra on sequential implementations but has a higher degree of parallelism.

Most of the parallel SSSP algorithms introduced in the literature have to deal with a trade-off between the amount of parallelism exposed and the extra work generated. Madduri et al. (2006) proposes a parallel $\Delta - stepping$ method that shows a good compromise between these two factors. They report an implementation exhibiting 30x speed-up on a CRAY MTA-2 shared memory architecture with 40 processors.

Implementations on Graphics Processors also showed to be a viable and relatively cheap solution for SSSP computation (HARISH et al., 2009; HARISH; NARAYANAN, 2007). On massively dense graphs, Kumar et al. (2011) use a modified Bellman-Ford algorithm on GPU and reports 10 to 12 times speedup over previous work for SSSP computation.

Edmonds et al. (2006) introduce the Parallel Boost Graph Library (Parallel BGL), a library of graph algorithms for distributed-memory computation on large graphs. Their implementation of SSSP extends Dijkstra by allowing to remove several vertices at once from the top of the priority queue. This technique is used to expose more parallelism but introduces unnecessary computation of edges.

Regarding the graph topologies, the literature shows that, in general, the proposed methods scale better on random-topology graphs with regular vertex-degrees than on structured graphs (grid-like topology) or scale-free graphs (containing few vertices with very high degree and a large majority with small degree) (EDMONDS et al., 2006; HARISH et al., 2009; HARISH; NARAYANAN, 2007).

2.3.3 Existing Implementations of SSSP Algorithms

Several GPU implementations have been proposed over the last years for different graph algorithms (HARISH et al., 2009). For the shortest path problem specifically, Dijkstra-based parallelizations are more frequently used (MADDURI et al., 2006; HARISH et al., 2009; ORTEGA-ARRANZ et al., 2013). Even so, there are also other approaches based on the Bellman-Ford algorithm, like on (KUMAR et al., 2011) targeting dense graphs on GPUs.

In general, Dijkstra based algorithms use a technique known as edge relaxation. In this technique, each vertex maintains a shortest-path estimate with distance $v.d$. The process of relaxation consists of trying to improve this estimate by going from vertex u to v through an edge of weight $w(u, v)$ (Algorithm 2.1).

Algorithm 2.1 Shortest path computation: relaxation algorithm

```
1: procedure RELAX($u, v, w$)
2:     if $v.d > u.d + w(u, v)$ then
3:         $v.d \leftarrow u.d + w(u, v)$
4:     end if
5: end procedure
```

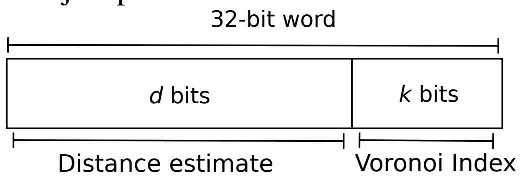
When done in parallel, each vertex u is assigned to threads that may update $v.d$ concurrently, thus creating a critical section. Consequently, lines 2 - 4, of Algorithm 2.1, have to be protected in an atomic region. In modern CUDA devices this atomic region can be efficiently implemented by the single atomic instruction $atomic\_min(addr, val)$.

> $atomic\_min(addr, val)$ reads word old located at the address $addr$, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction (NVIDIA, 2017).

2.4 Parallel Graph Voronoi on GPU

We present in this section our parallel algorithm for computing Voronoi diagrams on graphs. As mentioned previously, the Graph Voronoi can be seen as an extension of the shortest path problem. However, its parallelization poses additional problems of concurrent access to shared variables. In the Voronoi diagram problem, each voxel has to

Figure 2.5: Encoding information of distance and Voronoi index in a single word. Values *d* and *k* can be changed to adjust precision.

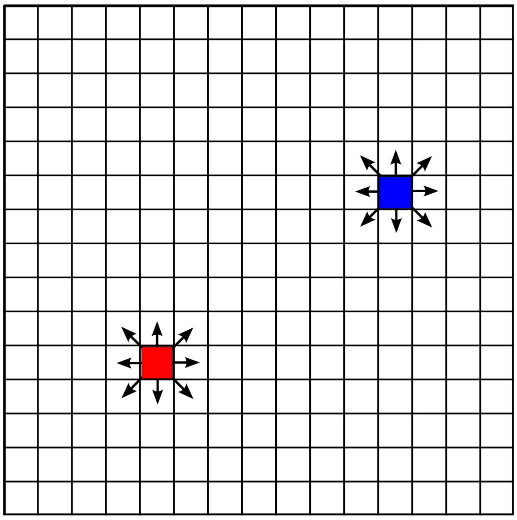


keep the distance estimate value to the seed and an extra variable for its Voronoi cell index. These variables would then be updated serially in the relaxation procedure, which, if executed by two threads in parallel, could lead to any combination of results in these variables. On concurrent programming this is a classical case of race condition. The straightforward solution for this problem would be to enclose the whole critical section (Algorithm 2.1) within mutex locks. However, mutexes are expensive structures to implement on GPUs. To deal with this problem, we choose to encode both variables, Voronoi index and distance estimate, in a single 32-bit word (Figure 2.5) that can then be atomically updated in a single *atomic_min()* instruction. Our encoding can be adjusted to balance distance precision and maximum number of Voronoi cells. In our implementation, we reserved 24 bits for the distance and 8 bits for the Voronoi region index.

2.4.1 Data Structure

Our data representation in memory substantially differs from the classical graph data structures. Instead of using adjacency matrices or lists for the shortest path computation, like in (HARISH et al., 2009), we are dealing directly with 3D images of voxels. In order to use the graph nomenclature, we refer to *voxels* as *nodes* of a graph. Each node has in general 26 neighbors (except at the image boundaries). Information stored at each node contains its compliance value, Voronoi cell index, and distance to a Voronoi source. The weight of each edge is given by some generalized distance function, $dist(C_v, C_u)$, defined for every pair of voxels (v, u) as long as they are adjacent in the image. In this work specifically, we employ the compliance scaled distance function used by Faure et Al. (FAURE et al., 2011). In this case, the distance between two adjacent nodes is a function of the measure of compliance of the material at each node.

Figure 2.6: Scatter updates: each active thread propagates its current information about distance and Voronoi index to its neighbors.



2.4.2 Parallel Algorithm

Our algorithm uses five internal arrays, C_0, C_1, Vor, $Mask$ and Mat, stored on the GPU global memory. Each one has the same size of the input volume (Algorithm 2.2). The Mat matrix contains the discretized material map, and is used to computed the distance between neighbor voxels. The cost arrays C_0 and C_1 are used to keep the shortest-path estimates of each voxel. They are initialized with 0 at the seeds and ∞ (maximum unsigned integer value) everywhere else. The Voronoi diagram, stored on array Vor, is initially empty on every voxel, except for those corresponding to the seed's coordinate that are initialized with a unique Voronoi cell index. Finally, the boolean array $Mask$ is used as activity mask to mark which voxels have an updated cost estimate indicating that it will be relaxed on the next step.

We assign one thread to every voxel. The execution then follows a scatter approach (Figure 2.6) where each active thread, marked on $Mask$, will relax the cost estimate of its neighbors and set their correct Voronoi index. The algorithm is divided in two parallel phases: *relaxation* and *update*. The host code (Algorithm 2.2) initializes the data structures and then iteratively calls the GPU kernels RELAXKERNEL (Algorithm 2.3), add UPDATEKERNEL (Algorithm 2.4), until the termination condition is satisfied. The distance function, at line 5 in RELAXKERNEL, computes the local distance between two neighbor voxels based on their compliance values in the material map (FAURE et al., 2011). At each iteration, C_1 maintains the intermediate values computed during the relaxation. In the UPDATEKERNEL procedure, the values from C_1 are copied back to C_0

and the activity mask is updated. The duplication of these cost matrices is needed to avoid read-after-write hazards when writing to global memory.

Algorithm 2.2 Parallel Voronoi computation in CUDA (host code).

```
procedure VORONOI($Seeds$, $Vor$, $Mat$)
    for all $v \in Mat$ do
        $C_0[v] \leftarrow \infty; C_1[v] \leftarrow \infty$
    end for
    for all $s \in Seeds$ do
        $C_0[s] \leftarrow 0; C_1[s] \leftarrow 0$
        $Mask[s] \leftarrow true$
        $Vor[s] \leftarrow idx++$
    end for
    repeat
        RELAXKERNEL($Mask$,$C_0$,$C_1$,$Mat$)
        $TERM \leftarrow true$
        UPDATEKERNEL($Mask$,$C_0$,$C_1$)
    until $TERM$
end procedure
```

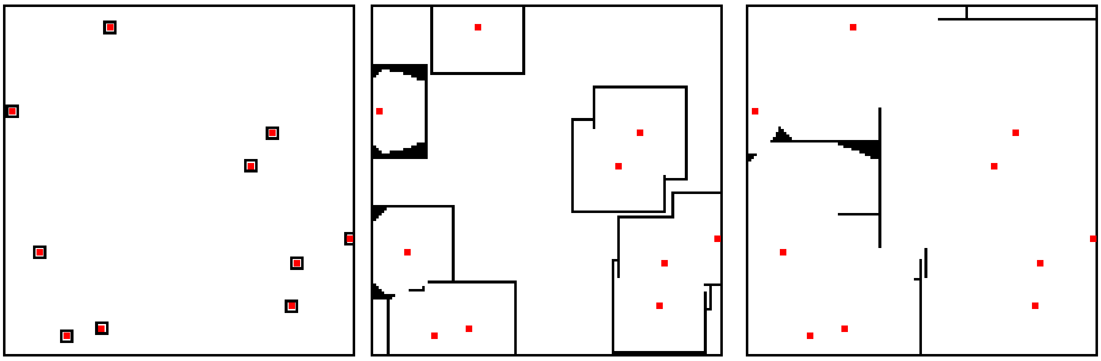
Algorithm 2.3 CUDA kernel for the relaxation algorithm: atomically updates the current shortest path estimates and the closest Voronoi seed.

```
procedure RELAXKERNEL
    $tid \leftarrow getVoxelIndex()$
    if $Mask[tid]$ then
        for all neighbors $nid$ of $tid$ do
            $d_{new} \leftarrow C_0[tid] + localDist(tid, nid, Mat)$
            AtomicMin($(C_1[nid] | Vor[nid])$, $(d_{new} | Vor[tid])$)
        end for
        $Mask[tid] \leftarrow false$
    end if
end procedure
```

Algorithm 2.4 CUDA update kernel: verifies the termination condition and updates the activity mask.

```
procedure UPDATEKERNEL
    $tid \leftarrow getThreadIndex()$
    if $C_0[tid] > C_1[tid]$ then
        $C_0[tid] \leftarrow C_1[tid]$
        $Mask[tid] \leftarrow true$
        $TERM \leftarrow false$
    end if
end procedure
```

Figure 2.7: At each iteration, the thread activity mask is updated. This process triggers propagation waves leaving from each Voronoi seed (red dots). As the distances are not linear some pixels will be recomputed causing the effect of "thicker waves" (b).



(a) Iteration 3 (b) Iteration 17 (c) Iteration 42

The propagation of distance and cell Id starts simultaneously at each seed and, for each iteration of lines 11 to 13 in Algorithm 2.2, are expanded one step further away. Plotting the trace of thread activations at each step generates the wave-like pattern shown in the 2D plan of Figure 2.7. The algorithm finishes when the Voronoi diagram computation reaches a fixed point, where no more voxel is updated.

2.4.3 Experimental Evaluation

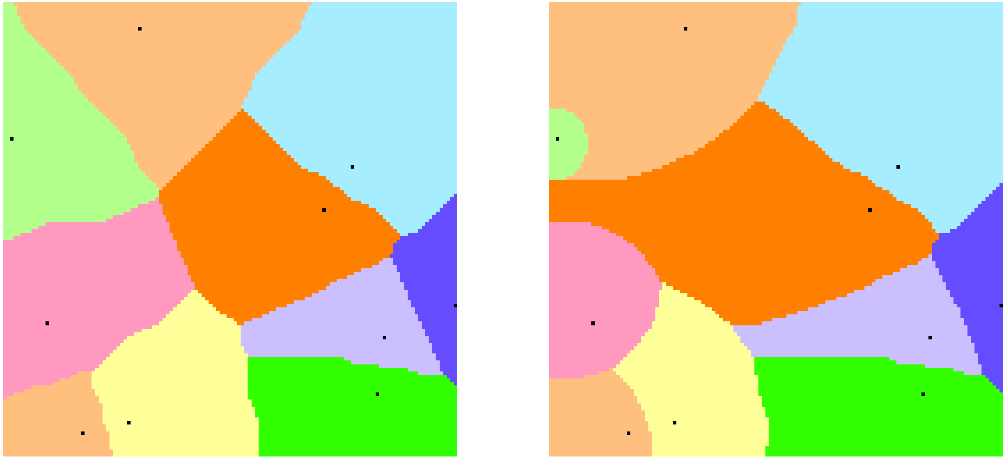
Several benchmarks were performed to evaluate the performance of our algorithm. In the following sections we describe our test environment and input instances used for the experiments.

2.4.3.1 Test Environment

The platform used for the CPU benchmarks was an Intel Core$^{\text{TM}}$ i7 CPU model 930 with 4 cores running at 2.89Ghz and 12 GB memory. Despite the multi-core architecture, the CPU implementation is strictly sequential.

The results of our GPU algorithm were obtained on a NVIDIA GPU GTX480 with 1.5 GBytes of global memory and 15 Multiprocessors with 32 cores each, totaling 480 CUDA cores. The CPU codes were compiled with GCC 4.8 using -O2 optimization flags. The CUDA driver is version 6 while the run-time is version 5.5.

Figure 2.8: Comparison of Voronoi diagrams generated with the same set of seeds on two different material maps. Left: with an uniform stiffness. Right: with a stiffness gradient.



2.4.3.2 Input Instances

The input dataset differs on 3 different parameters: volume size, material map topology and number of Voronoi seeds. For the material map topologies we considered both synthetic and real-application data. The synthetic topologies represents a cube volume with (a) an uniform constant stiffness and (b) a gradient stiffness varying uniformly from left to right (called *Gradient*). In these topologies, we variate the volume from 32^3 to 256^3 voxels, which are the common discretization sizes used for physically based simulations in (FAURE et al., 2011). For each volume size, a placement of seeds was randomly generated and kept constant for each run of the benchmark. We note that, due to the compliance-scaled distance function employed, the same set of seeds actually generate very different Voronoi diagrams, depending on the topology of the material map used (see Figure 2.8).

The real-application dataset is the discretized material map of the T-bone steak (Figure 2.3). The map of the steak has a volume of size `64x64x15` voxels and exhibits non-uniform stiffness distribution. The data of the steak is freely available for download with the SOFA framework (SOFA, 2017; FAURE et al., 2012).

2.4.4 Results and Discussion

In this section we evaluate the implementation described in the previous sections. Performance of Voronoi computation is analyzed regarding 3 main parameters: volume size, topology of the material map and number seeds of the diagram. Following these

Table 2.1: Benchmark results with execution times and speed-up obtained with the base GPU algorithm.

				CPU			GPU			
		Volume	#Seeds	Iters.	Time(ms)	99% CI	Iters.	Time(ms)	99% CI	Speed-up
Topology	Gradient	32^3	10	32768	32.24	±0.23	30	1.96	±0.02	16.46
		64^3	10	262144	296.77	±1.48	70	12.13	±0.04	24.46
		128^3	10	2097152	3863.54	±11.36	126	105.91	±0.15	36.48
		256^3	10	16777216	38756.80	±176.48	195	985.80	±0.20	39.31
		100 × 40 × 20	20	80000	77.04	±0.37	21	2.51	±0.02	30.68
	Constant	32^3	10	32768	21.37	±0.19	20	1.27	±0.01	16.80
		64^3	10	262144	190.81	±0.56	52	9.20	±0.03	20.73
		128^3	10	2097152	1957.65	±6.59	79	61.85	±0.02	31.65
		256^3	10	16777216	22103.64	±55.85	159	812.03	±0.28	27.22
		100 × 40 × 20	20	80000	52.31	±0.26	22	2.47	±0.01	21.15
	Steak	64 × 64 × 15	3	12276	10.80	±0.10	38	2.16	±0.03	5.01
		64 × 64 × 15	10	12276	10.39	±0.19	45	2.72	±0.02	3.82

results we proposed an enhancement to the base algorithm which will be presented on Section 2.4.5.

2.4.4.1 Base Algorithm Speed-up

We start by comparing our parallel algorithm with its sequential reference implementation on CPU. For each input instance, we ran the benchmark 10 times, for the CPU and GPU algorithms, and computed the mean execution time. The obtained means and 99% confidence intervals are reported in Table 2.1.

Figure 2.9 presents the speed-ups obtained with GPU implementation over the sequential one on CPU. Each bar represents a different input instance where labels $cube32^3$ *10s*, $cube64^3$ *10s* , $cube128^3$ *10s* and $cube256^3$ *10s* denote a cube with gradient topology with dimensions 32, 64, 128 and 256 respectively, each with 10 seeds. The *plate 100x40x10 20s* input is a plate of stiffness gradient with 20 seeds. Both *steak* instances have a bounding volume of 64x64x15 voxels.

In this benchmark the speedup achieved varies from 3.8x for small volumes (Steak) up to almost 40x for bigger ones. These results show that our algorithm benefits from bigger input sizes, because they expose more parallelism. This observation is also confirmed by the trace of thread activity shown in Figure 2.10. Notice for instance the different scale on y-axis between facets (a) and (b) (Figure 2.10) showing a higher maximum of active parallel threads in the `Cube` volume of size 128^3 than in the `Plate` one with 100 × 40 × 20 voxels.

Figure 2.9: Speed-up for different input sizes. Gradient and constant topologies are presented for synthetic benchmarks only. Steak's topology corresponds to the real dataset of Figure 2.3.

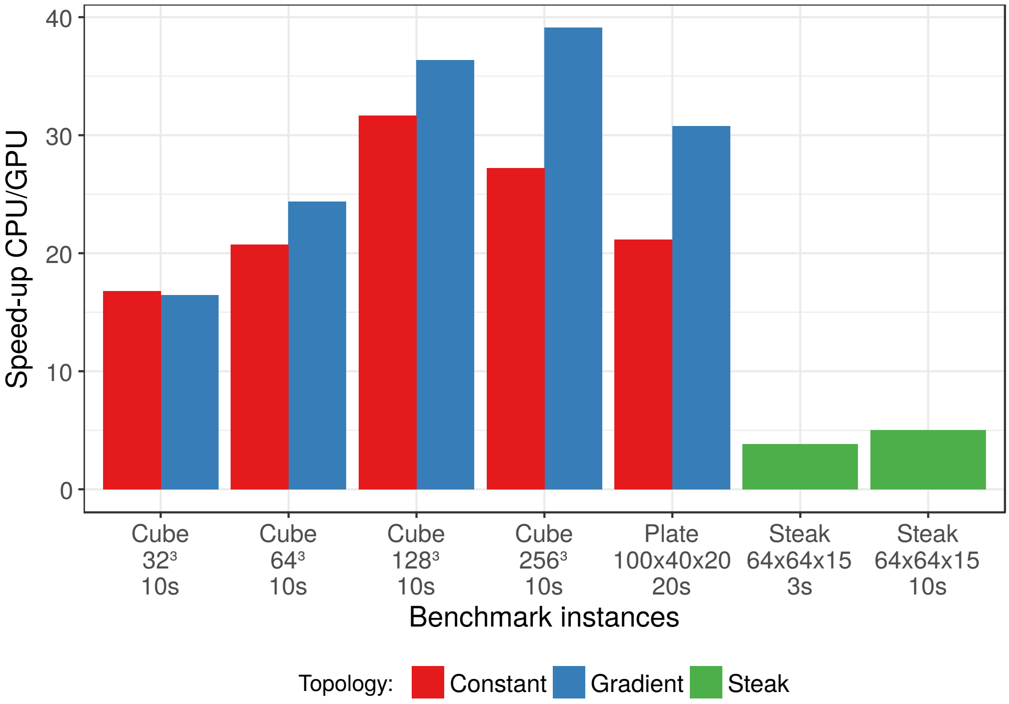


Figure 2.10: Trace of the number of active threads running in RELAXKERNEL at each each iteration of Algorithm 2.2. (threads counted in thousands on the y axis)}

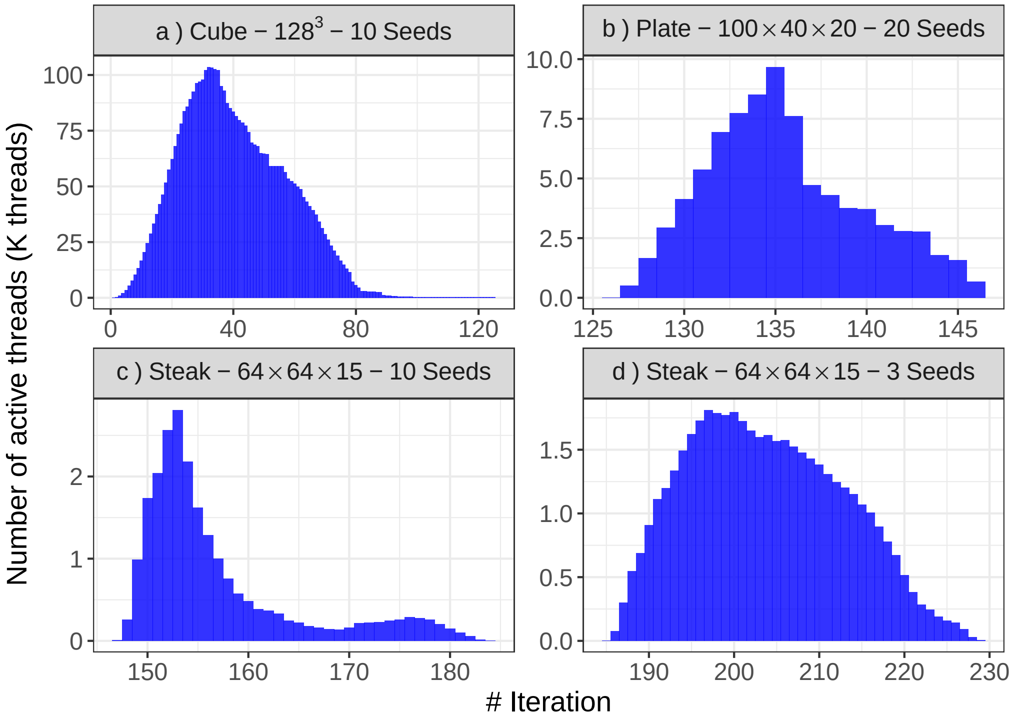
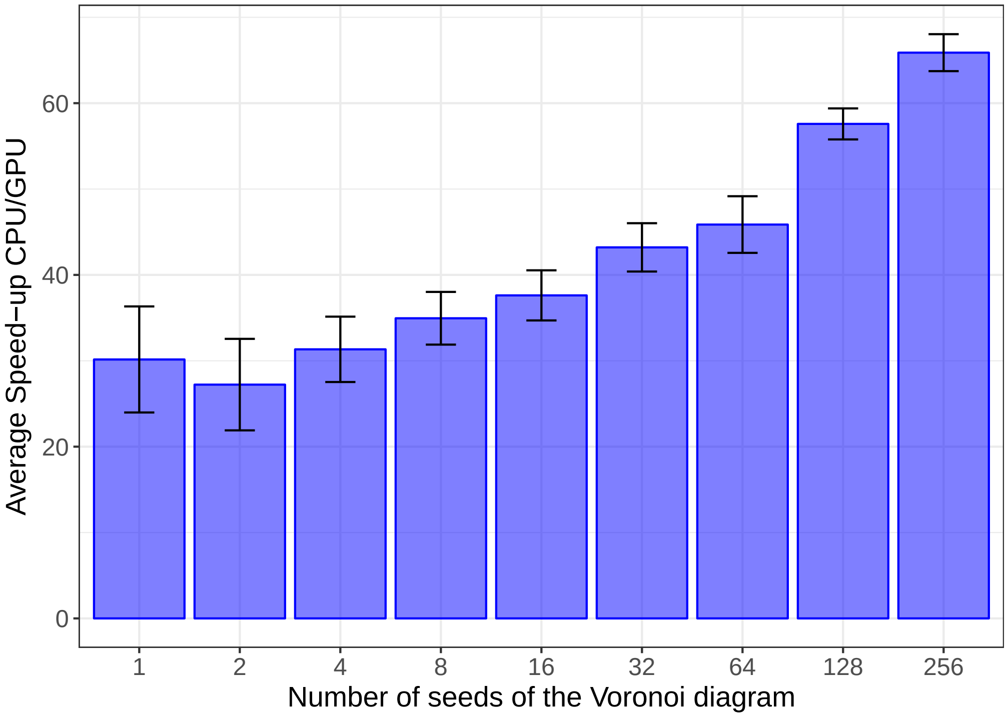


Figure 2.11: Average speed-up when increasing the number of seeds of the Voronoi diagram. Standard deviations are shown on top of each bar.



2.4.4.2 Voronoi Seeds

On a second scenario, we investigated the impact of the number of seeds in the performance. For this benchmark we fixed the volume size at 128^3 and used the same gradient topology to vary the number of seeds. For each number, we randomly generated 10 different seed placements.

The execution time was measured for each configuration of seeds and the speed-up was computed. Each reported value in the bar plot of Figure 2.11 is an average of the speed-ups over 10 different seed placements. The standard deviation plotted on top of each bar gives a notion of the variability of the speed-up according to the placement of seeds in the object. With more than 4 seeds, these results suggest that for larger amounts of seeds the speed-up generally increases. Indeed, having more Voronoi seeds has the effect of allowing more active threads right at the first iterations. Moreover, assuming a uniform distribution of seeds, the number of iterations of the algorithm tends to reduce as more Voronoi cells expand concurrently. Notice the different number of iterations (x-axis) shown on Figure 2.10, facets (c) and (d) (same volume with 10 and 3 seeds respectively).

In real scenarios, more sophisticated algorithms are used to configure the placement of seeds. However, the rationale behind them is usually to keep a uniform distribution over the material map. Therefore, the random placement strategy used in this experiment remains a fair choice for our synthetic benchmarks.

2.4.5 Optimization with Stream Compression

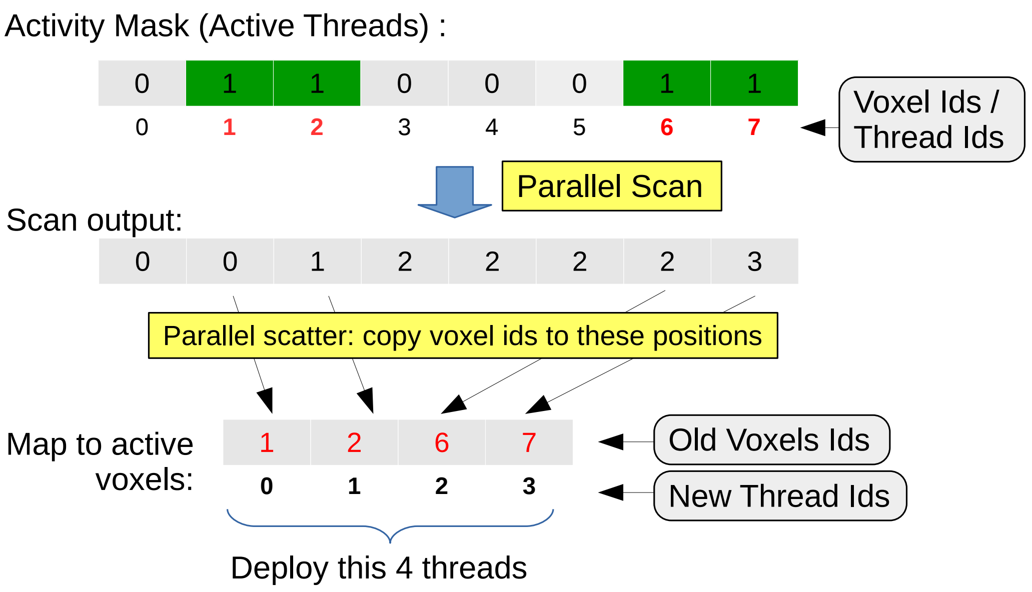
The base implementation presented on the previous section already exhibits considerable speed-up over the sequential algorithm. The mapping from voxels on the volume to CUDA threads is done directly. This means that for a volume of dimensions 32^3, there will be 32^3 threads, one for each voxel, deployed every iteration of the algorithm. This direct mapping from pixels to threads is a common practice on GPU algorithms and benefits from their lightweight thread scheduling features. Nonetheless, as observed in picture Figure 2.10 and Figure 2.7, the number of threads that are actually updating values at each iteration is much smaller than the total size of the volume (i.e. the amount of threads deployed at each iteration). Figure 2.7 shows how the computation propagates to neighbors in a wave-like form. The black front indicates, at each step, which threads are set to *true* in the activity mask at the beginning of the relaxation kernel (line 3 in Algorithm 2.3). Over the execution of the algorithm, the number of active threads is much lower than the total grid size and varies considerably along time (Figure 2.10). This causes our thread blocks to be very inefficient as most of the threads will actually evaluate the conditional to *false*, without computing anything (line 3, Algorithm 2.3) .

One might question if a finer control of the number of threads reflects in a better utilization of the GPU with consequent performance enhancements. To tackle this issue, we proposed a modification to our base algorithm that applies a technique known as *stream compression* (HOBEROCK et al., 2009; HARISH et al., 2009).

With stream compression, we perform an extra pass on the activity mask to count the number of voxels marked for update and to generate a new indirect mapping from thread Ids to voxel Ids. This way, the active threads are grouped in fewer and more compact blocks, thus reducing branch divergence and runtime overhead of scheduling idle threads. Stream compression is achieved by a *Scan* operation over the activity mask followed by a *Scatter* as shown in Figure 2.12. These operations can be easily implemented in CUDA using the Thrust template library (HOBEROCK; BELL, 2011). The result of the compression is an array mapping thread indices to voxel coordinates, that are used in the RELAXKERNEL to retrieve the correct data (Algorithm 2.3 line 2).

This process adds a non-negligible overhead that sometimes can actually super-

Figure 2.12: Stream compression can be used to deploy only the exact amount of threads required to update the active voxels.



sede the gains of performance. To be able to balance the trade-off between performance gains of compression and time spent by the scan+scatter process, we implemented a *coarse grain compression*. This optimization defines a coarser subdivision over the activity mask as shown in Figure 2.14 (in 2D) and detailed in Figure 2.13. The coarser mask is parametrized by a grain size. The algorithm then scans the coarser mask, identifying which grains contain active threads and launches only this amount of threads. Note that the coarser the grain used, the more idle threads will be deployed. As we are dealing with 3D volumes, the granularity of the mask is defined by their x, y and z dimensions. We use the notation $dim_x \times dim_y \times dim_z$ to refer to different grains used in our performance analysis.

2.4.6 Stream Compression Evaluation

As explained in the previous section, stream compression optimization can be parametrized by setting the grain dimensions used for the subdivision of the coarser mask (Figure 2.14). We used the CUDA profiling tools to analyze the trade-off between overhead of stream compression and gained performance at several grains. We summarized the most representative result in the stacked histogram of Figure 2.15, which shows results for a volume size of 128^3 with gradient topology and a set of 10 fixed seeds. The bars are sorted by total execution time and each grain size is indicated on the *x* axis, where "static" refers to the base algorithm (without stream compression).

Figure 2.13: A coarser activity mask is used to reduce overhead of stream compression.

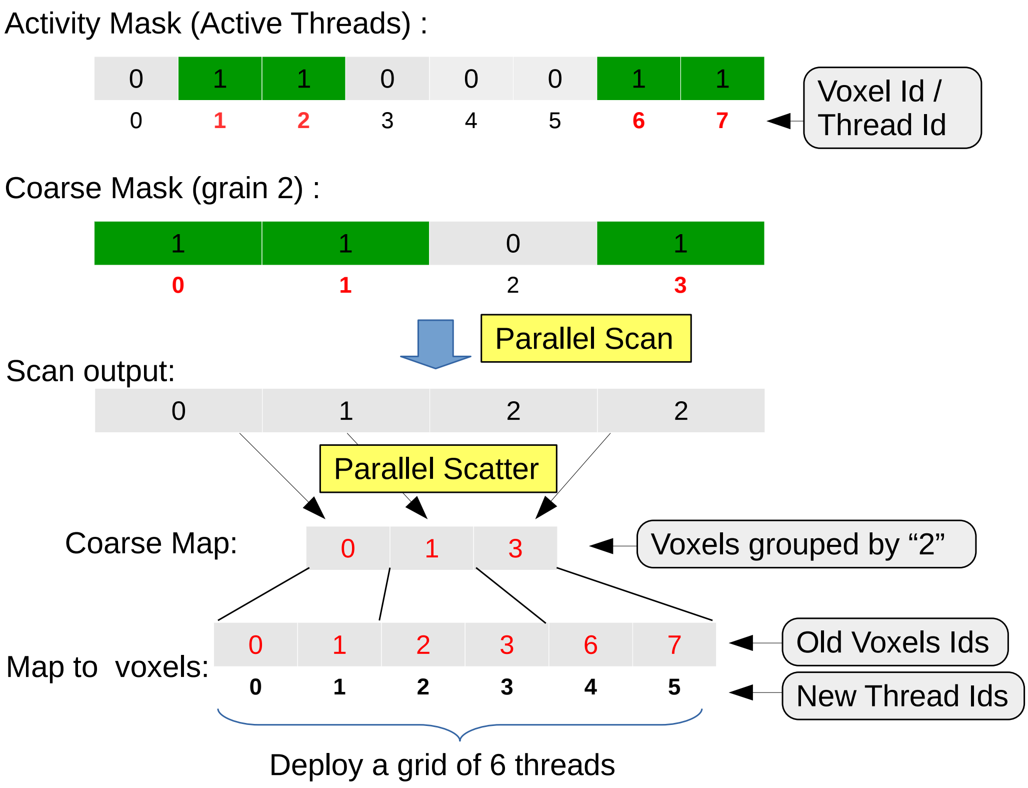
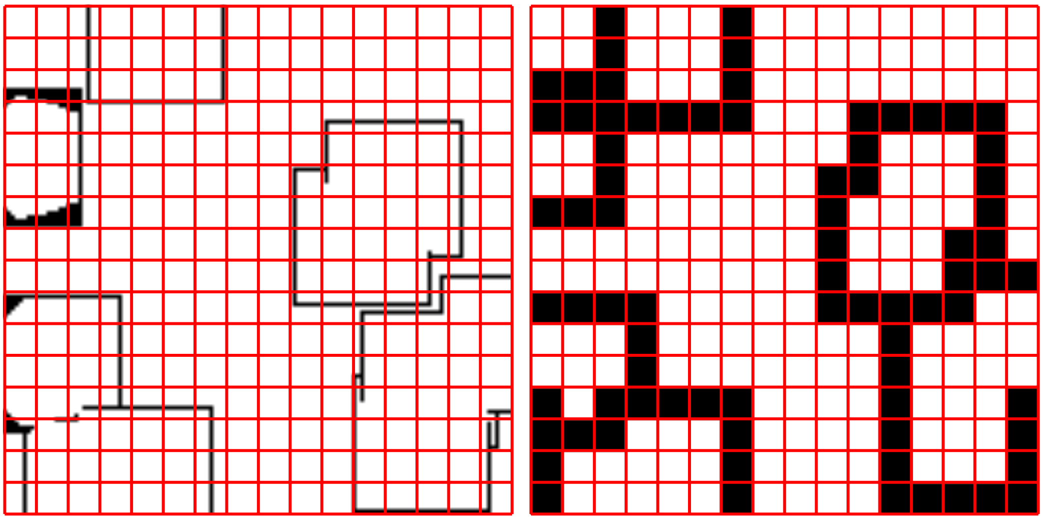


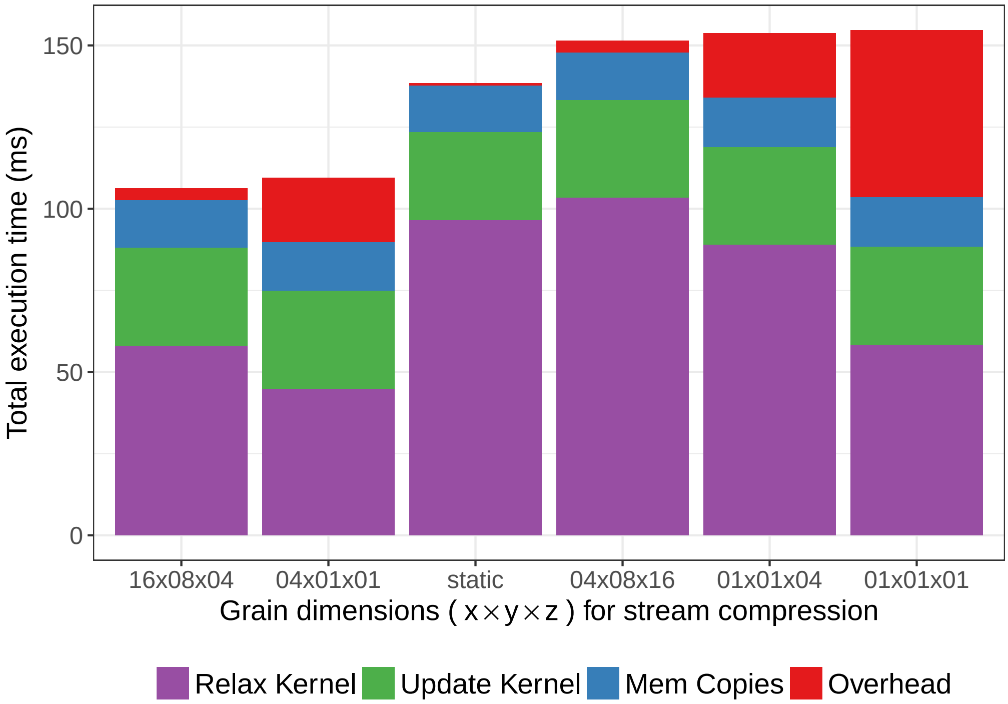
Figure 2.14: View in the 2D plan of the activity mask with stream compression at a coarse grain size.



(a) Original Activity Mask with subdivisions

(b) Produced Coarser Mask

Figure 2.15: Profiling of stream compression optimization.



Finer grains generate larger masks, therefore add more overhead for generating the map of active threads. A finer grain, however, results in a better compression, which reduces the amount of idle threads and useless thread-blocks. This positively impacts the time spent on the RELAXKERNEL procedure. See column $1 \times 1 \times 1$ in Figure 2.15. On the other hand, more compact thread-blocks will also increase the number of threads accessing non-coalesced memory locations. In our application the volume is stored linearly in memory, which means that neighbour voxels on the x dimension are stored contiguously in memory. Favoring a larger x grain-dimension increases the number of threads accessing the same memory segment, thus achieving a better memory throughput. This fact can be observed comparing grains of same sizes, but different shapes like $4 \times 1 \times 1$ and $1 \times 1 \times 4$.

Experiments with smaller volume sizes, like 32^3 and 64^3, showed worse total execution time than the base algorithm. Nevertheless, for the 128^3 volume, the technique of stream compression led to a 23.29% performance gain over the base implementation. The size of the volumes tested in this benchmark were limited by the amount of memory available by the GPU, used by auxiliary masks and indirect mappings structures. It was not possible to test the instance of 256^3, however we expect that *coarse stream compression* would represent an even larger speed-up because there would be more idles threads in such cases.

2.4.7 Conclusion

In this section we presented a GPU algorithm for computing the Voronoi diagram with generalized distance functions. Our method adapts a graph algorithm, for the SSSP problem, to compute the Voronoi diagram on a 3D grid of voxels. We have shown through an experimental evaluation that our base parallel implementation significantly speeds-up Voronoi computation. Additionally, we proposed a new optimization strategy called *Coarse Stream Compression* that allows to increase utilization of the GPU on large volumes. Different from previous approaches of stream compression, we have shown that we are able to trade-off the overhead of the method by carefully choosing a grain size.

Regarding our implementation, there is still room for optimization on the data representation in memory. Its current linear representation cannot benefit from the locality present in the neighborhood computation. A better memory layout, like Z-curves, could further enhance the performance.

The contributions presented in this chapter have direct application on physically based simulation algorithms where Voronoi diagrams are used to compute shape-functions. The GPU algorithm proposed is useful in scenarios where a fast computation of Voronoi diagrams is required, for instance when the shortest-path distance change because of on-line modification on the object's topology or on the material's property map (e.g. stiffness). Such situations are common, for instance, in interactive simulation of *tearing* and *cutting* (MANTEAUX et al., 2015).

2.5 Parallel Voronoi-based Interpolations

In this section we present another contribution of this thesis. We build on the first algorithm that we just presented and extend it to a classical application of Voronoi diagrams – the *Natural Neighbors Interpolation* method (NNI). We propose a parallel NNI implementation based on the GPU algorithm of Section 2.4. The NNI and derivatives are used in the context of meshless simulations to interpolate shape functions' weights from different DoFs over the object domain. Before detailing the parallel algorithm, we briefly introduce the NNI method.

2.5.1 The Natural Neighbor Interpolation Method

In applied mathematics, a classical application of the Voronoi diagram was brought to bear by Rodin Sibson in 1981 on the development of the *natural neighbor interpolation* method (SIBSON, 1981). Sibson's interpolation is a well-known method for interpolating irregular spatial data. It has applications in several different fields such as medical imaging, meteorological or geological modeling (BEUTEL et al., 2010), flow map reconstruction (BARAKAT; TRICOCHE, 2013) and scattered data visualization (PARK et al., 2006). Natural Neighbor based interpolations have also been used in the field of solid mechanics by Sukumar (2003) in the *Natural Element Method* (NEM) where Sibson's and non-Sibsonian (Laplace) interpolators are employed to perform crack simulations.

A lot of effort has already been dedicated to improve the performance of Sibson's interpolation, particularly when interpolating on a discrete grid in the Euclidean space - known as the *Discrete Sibson Interpolation*. A popular approach relies on the parallelization with Graphics Processing Units (GPUs) of the Discrete Voronoi diagram like on (BEUTEL et al., 2010) for Digital Elevation Model (DEM) construction.

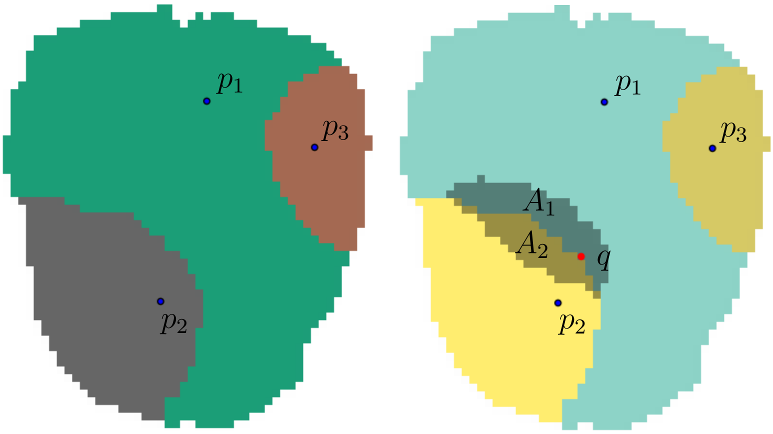
2.5.2 Voronoi Based Interpolation Methods

One of the most well-known interpolation methods based on the Voronoi diagram is the NNI method proposed by Sibson (1981). Consider the problem of finding neighbors in a set of non uniformly distributed data points. By taking the Voronoi tessellation induced by these points, the *natural neighbors* are the data points whose Voronoi cells share a common frontier.

We briefly explain below three different methods of scattered data interpolation that require the computation of Voronoi diagrams.

The Sibson interpolation is defined as a ratio of areas in 2D (volumes in 3D). We insert the query point q in the initial Voronoi tessellation of sample points (Figure 2.16(b)). The interpolating weight w_i of each data point is given by the ratio between the area A_i overlapping the neighbor Voronoi cell and the total area A_q of the newly inserted cell. In the example of Figure 2.16, the interpolation of a function f at the point q is given by:

Figure 2.16: **NNI: (a)** A Voronoi partitioning of the space is generated from sample data points (p_1, p_2, p_3); **(b)** Interpolated values can be queried at any point q by inserting a *query seed* (red point). The Voronoi cell of q is shown by the shaded area. A_1 and A_2 are the intersection areas between p_1's and p_2's Voronoi cell, respectively, with q's. The coeficients for interpolating values from p_1 and p_2 at q are the ratios of A_1 and A_2, respectively, by the shaded area.



(a) Initial Voronoi diagram (b) *NNI* Query

$$f(q) = \sum_{n=1,2} w_i * f(p_i) \ ,$$

where

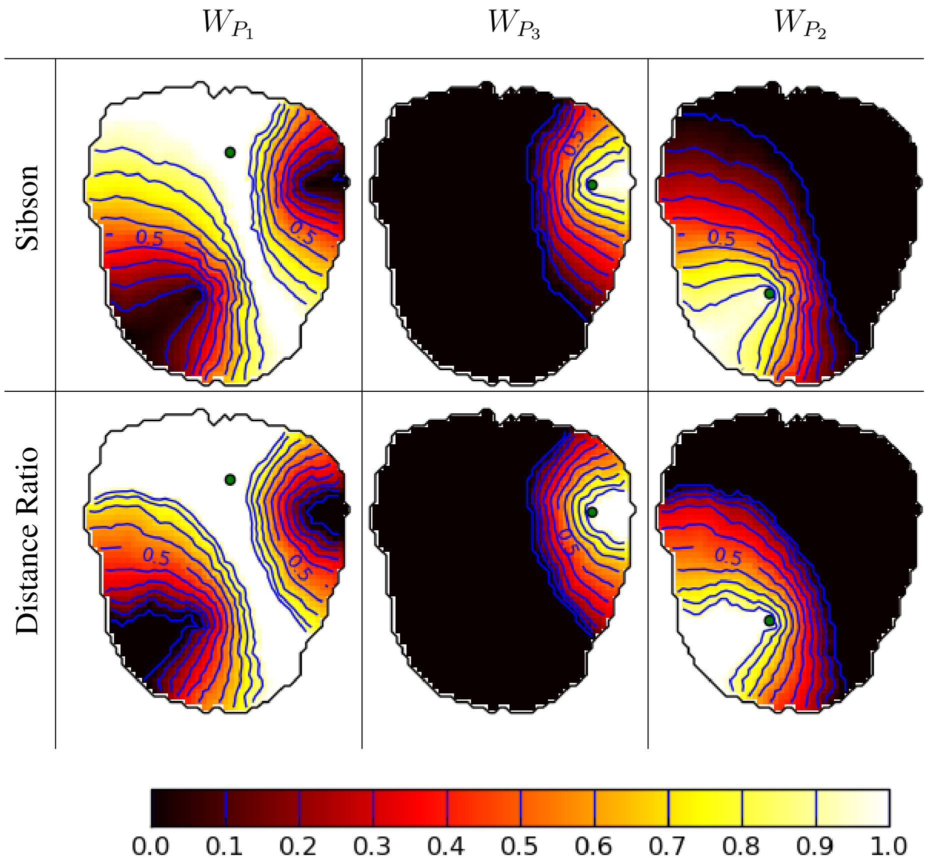
$$w_1 = \frac{A_1}{A_q} \qquad w_2 = \frac{A_2}{A_q} \ .$$

The Laplace (non-Sibsonian) interpolation ((BELIKOV et al., 1997) as cited in (SUKUMAR, 2003)) uses the same notion of natural neighbor but it computes the ratio between segments in 2D (areas in 3D). Instead of taking the area of the neighbor cells it uses the ratio between the length of the Voronoi frontier (line in 2D / facet in 3D) and the distance from q to its natural neighbor nodes p.

Both natural neighbors interpolation methods described above require the computation of a new Voronoi diagram for each query point q added to the input diagram of the data samples. This results in Q Voronoi diagram computations where Q is the interpolation resolution desired. To reduce the complexity in terms of number of Voronoi diagrams computed per query point, Faure et al. (2011) use an alternative interpolation method where the number of Voronoi diagrams computed is a constant factor of the amount of data samples S.

The Distance Ratio interpolation (FAURE et al., 2011) applies a particular scheme that computes the ratio between the distance from the query point q to the Voronoi bor-

Figure 2.17: Example comparing two methods for interpolation of shape-functions' weights ($W_{P_1}, W_{P_2}, W_{P_3}$) of three control frames (green dots) on the steak object. Interpolation method: Sibson interpolation (**top**), Distance Ratio (**bottom**). Weights are normalized starting from 1 at the frames location (Voronoi cell center) and decreases until they vanish outside of the support.



der d_{b_i} and the distance from q to the Voronoi seed d_{p_i}. Although less formal guaranties on the properties were presented for this interpolant, this algorithm is implemented on SOFA (SOFA, 2017) and shows good practical results, with the advantage of being more computationally efficient. Figure 2.17 presents a visual comparison of this method with Sibson's. The heatmaps show the interpolated weights relative to the three control frames over the steak object.

2.5.3 Parallel NNI Algorithm

In the classical natural neighbor interpolation, each query point q is inserted, one at a time, to the Voronoi diagram of initial data samples. For each seed added, the initial diagram is updated to generate the new Voronoi cell that will then be used to compute the interpolation.

As seen previously, in the experimental results of Section 2.4.4.2, the number of seeds computed in the Voronoi diagram has a big impact on the amount of parallelism that will be exposed. Computing a single Voronoi cell in parallel reduces the possibilities of parallelization. This situation is even worse when we consider to update an existing Voronoi diagram with the addition of a new seed (we refer to it as the *query seed*). In this case, only a limited region around the *query seed* would be recomputed (see shaded Voronoi cell in Figure 2.16(b)). Performing the interpolation simply as a sequence of (parallel) Voronoi computations on the GPU doesn't payoff the overheads of memory transfers and thread scheduling inherent to this architecture.

The evident strategy to generate interpolated values over a grid would be to perform multiple queries concurrently in parallel. This approach was used in (BEUTEL et al., 2010) to generate the interpolation over a regular grid using the assumption that every sample has a limited radius of influence. This allowed to decompose the domain in independent blocks where queries could be answered in parallel batches. Park et al. (2006) propose a more efficient implementation of Sibson's interpolation on raster images. Their method avoids the explicit construction of a new Voronoi diagram for each query point. Instead they use a Kd-tree structure to find the closest seed to the current query point and use this distance as a radius of influence to increment the interpolation weight. Once again, these techniques make assumptions that are valid for Euclidean space but not trivially generalized for geodesic (graph) distances.

Parallelization can still be implemented for NNI over graph spaces if we duplicate some data structures. More precisely, for each NNI query running in parallel, we copy the input distance map of shortest paths and insert a new seed at the queried coordinates with distance zero. The distance map is then updated using the same algorithm of Voronoi computation presented in Section 2.4.2 with the added difference that we update an extra counter for each Voronoi cell of the input diagram to track the number of voxels overlapping in each cell. The array of counters is then copied back to the CPU, where the ratio of areas is computed and the weights images generated. With this approach several queries can be grouped in a parallel batch and deployed for execution on the GPU. The batch size is a parameter that allows to control the trade-off between the amount of parallelism exposed and extra memory consumed. Note that for each query in the batch, the distance map and the counters of the overlapping voxels must be replicated.

Table 2.2: Comparison of running times of the parallel Sibson interpolation algorithm varying the number of queries per batch. Parallel NNI queries run in batches until a whole grid of $100 \times 40 \times 20$ voxels is computed.

Impl$^{\cdot\text{ion}}$	**Size**	**N**	**Mean$^{\text{(ms)}}$**	**SD**	**SEM**	**Sum$^{\text{(ms)}}$**	**SES**	$\frac{Tot}{80k}$
CPU	1	80k	1.004	0.230	0.001	80284.528	65.042	1.004
GPU	1	80k	9.746	1.210	0.004	779657.760	342.332	9.746
GPU	10	8k	12.414	1.424	0.016	99309.125	127.361	1.241
GPU	100	800	29.537	3.041	0.108	23629.871	86.005	0.295
GPU	150	534	34.807	3.278	0.142	18587.131	75.752	0.232

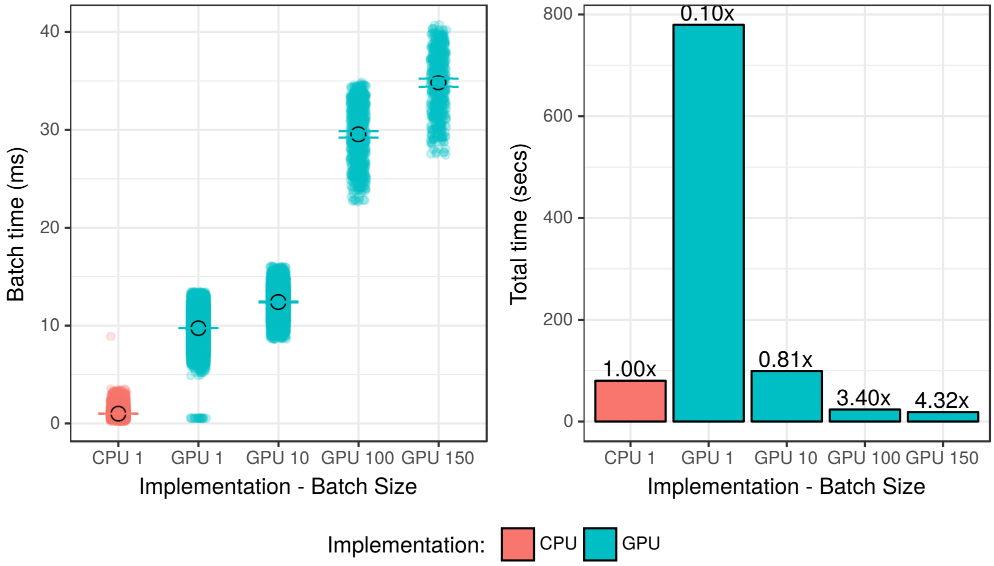
2.5.4 Performance Evaluation

This section evaluates the performance of our GPU implementation of parallel NNI queries on non-Euclidean spaces. The dataset used for this experiment consists of the *plate* volume with $(100 \times 40 \times 20)$ voxels with uniform topology and 20 data points (Voronoi seeds). The experiment performs a total of $80,000$ NNI queries, one for each voxel. Queries are grouped in batches of fixed size that are deployed sequentially to the GPU until the whole grid of voxels is computed. The program outputs a 4D image with $(100 \times 40 \times 20 \times 20)$ normalized *weights*, one for each voxel and data point. These benchmarks were performed on a GPU NVidia Tesla K40c, with 11.25 GB of global memory and 2880 CUDA cores.

Time was measured once and individually for each batch execution. In Table 2.2, we report the mean runtime, and standard deviation (SD), for batches of a given Size over a total of N batches required by a complete grid interpolation. We also report the total time for computing the full volume interpolation as the Sum of runtimes of each batch. For the sake of statistical comparison we compute the Standard Error of the Mean (SEM) and Standard Error of the Sum (SES), respectively, as $\frac{SD}{\sqrt{N}}$ and $SD\sqrt{N}$. Finally, the amortized time per voxel (i.e. Total time/ 80k) is shown in the last column of this same table.

A graphical comparison of runtimes is shown in Figure 2.18. The left-side plot presents the time of batch executions with 99.7% confidence interval $(3 \times SEM)$. The right-side plot compares the total runtimes and shows, on top of each bar, the speed-up relative to the sequential CPU version. In this plot, confidence intervals were negligible and therefore omitted. Notice the $10\times$ factor difference between the total runtime of the sequential CPU implementation and the GPU with a unitary batch (GPU1). This difference characterizes the approximated overhead of the GPU implementation. Indeed,

Figure 2.18: Parallel Sibson interpolation analysis. **Left:** average time per batch (Milliseconds) ; **Right:** total running time of the grid interpolation (Seconds).



when testing with a batch $10\times$ larger, the runtimes get much closer reaching a "speed-up" of 0.81, although it still represents a slow-down. Speed-up can be achieved at larger batch sizes like on the 100 and 150 batches.

2.5.5 Conclusions

Previously, in Section 2.4 we have revisited a classical geometrical structure – the Voronoi diagram – in the context of its non-usual application to meshless interactive simulations. These diagrams, have the particularity of been defined over a non-Euclidean space. The existing technique to compute these diagrams were not suited to the real-time requirements of the simulation application. We proposed an algorithm that uses the parallelism in GPUs to speed-up Voronoi diagrams computation. For the current image size used in the simulation (32^3), our implementation achieves near $17\times$ speed-up and can be computed in under 2 ms.

In section Section 2.5, we built on the parallel Voronoi diagram algorithm and extended it to compute the well-known natural neighbor interpolation. The proposed algorithm allows to compute the NNI of a set data samples on a discrete grid with non-Euclidean distances. Points in the grid (i.e. NNI queries) can be computed in parallel when grouped in batches. For sufficiently large batches, the parallel algorithm shows good performances with reported speed-up of $4.32\times$ on batches of size 150.

Both parallel algorithms presented have a valuable application in soft object simulation methods. They provide performance solutions to the physically based simulation community willing to employ Voronoi shape-functions in their meshless simulations.

In current simulations, the grid for interpolation is static and computed at initialization for fixed image resolution. Typically, during the simulation only a subset of the precomputed weights on the image will be actually used. Clearly, an interpolation over the full set of points in the grid represents a wasteful amount of computation. Precomputation was required to avoid doing NNI queries during the simulation itself. As the full grid is not usually needed, with our parallel batch algorithm we could afford the on-line computation of a subset of NNI queries. This represents a change in the way these simulations are currently designed. We expect that these parallel solutions can be used to support dynamic modifications of topologies during the simulation with on-demand recomputation of shape functions. Incorporating this work into SOFA, for experimenting with actual simulations, represents a significant programming effort that is left as future work.

Finally, recent work have proposed to also use Voronoi Shape-functions on grids with "extended" connectivity called *non-manifold grids* (MANTEAUX et al., 2015), which would allow to represent objects with more complex topologies in meshless frameworks. A possible extension of this work could consider the application of our parallel algorithms in these new domains.

3 A PROBLEM ON STREAMING DATA VISUALIZATION

3.1 Context

Advanced visualization tools are essential when it comes to big data analysis. First approaches focused on large static datasets stored in cloud infrastructures. However, on a worldwide scale, there are always active users on social networks or buyers on on-line retailers. Analyzing and visualizing these streams of data as they are generated is becoming essential. Twitter is a common example. The flow of tweets is continuous, and users want to be aware of the latest trends. This need is expected to further grow with the Internet of things (IoT) and the associated massive deployment of sensors that will generate extremely large and heterogeneous data streams. Over the past years, several in-memory big-data management systems have appeared in academia and industry. In-memory databases systems avoid the overheads related to traditional I/O disk-based systems and have made it possible to perform interactive data-analysis over large amounts of data. A vast literature of systems and research works address different aspects of these systems (ZHANG et al., 2015). Notably, algorithms in this area must deal with the limited storage size and a multiple level memory hierarchy of caches. Maintaining a right data layout that favors locality of access is a determinant factor for the performance of in-memory processing systems. Stream processing engines like Spark or Flink (ZAHARIA et al., 2013; CARBONE et al., 2015) support the concept of a *window*, which collects the latest events without a specific data organization. It is possible to trigger the analysis upon the occurrence of a given set of criteria (time, volume, specific event occurrence). After a window is updated, the system simply shifts the processing to the next batch of events. There is a need to go one step further to keep a live window continuously updated while having a fine grain data replacement policy to control the memory footprint. The challenge is the design of dynamic data structures to absorb high rate data streams, stash away the oldest data in order to stay within the allowed memory budget while enabling fast queries executions to update visual representations. A possible approach consists in extending data structures such as R-trees (GUTTMAN, 1984) used in databases like `SpatiaLite` (SPATIALITE, 2017) or `PostGis` (POSTGIS, 2017), or to develop dedicated frameworks like `Mercury` and `Venus` based on a pyramid structure (MAGDY et al., 2014; MAGDY et al., 2016).

In this work, we propose a novel self-organized cache-oblivious data structure, called *Packed-Memory Quadtrees (PMQ)*, for in-memory storage and indexation of fixed length records tagged with a spatiotemporal index. The PMQ combines a Packed Memory Array (BENDER et al., 2005) and a Morton indexed quadtree (GARGANTINI, 1982). We store the data in an array with a controlled density of gaps, i.e. empty slots. These slots guarantee that insertions can be performed with a low amortized number of data movements ($O(\log^2(N))$) while enabling efficient spatiotemporal queries. During insertions, parts of the array are rebalanced when required to respect density constraints, and the oldest data stashed away when reaching the memory budget. We sort data according to their Morton index ensuring a good spatial locality in the array, leading to efficient spatiotemporal queries. We experiment the PMQ for absorbing a stream of tweets and query the data structure to support different types of visual queries. The PMQ significantly outperform other approaches like the specialized `Kite` framework (MAGDY; MOKBEL, 2017; MAGDY et al., 2014; MAGDY et al., 2016) or the R-tree based geospatial databases like the in-memory `SpatiaLite` (SPATIALITE, 2017) and `PostGis` (POSTGIS, 2017). In summary, we contribute:

- a self-organized cache-oblivious data structure for storing and indexing streaming spatiotemporal datasets;
- algorithms to support visual queries and automatic alert detection over streaming data;
- comparison against state-of-the-art competing strategies.

3.2 Background

In this section we review background material on packed-memory arrays, Z-curves, and linear quadtrees used for the PMQ.

3.2.1 The Packed Memory Array (PMA)

Consider the following toy example: Given an array of <*key,value*> elements *sorted* by their keys (where the keys are not necessarily unique), a new element is to be inserted in this array in the correct position following the current order. The naive approach could be to insert it at the end of the array, leaving the array temporally in an unsorted state and performing a complete sort on this array. Another approach, apparently more efficient, could be to search the insertion position for the newcomer and shift all the following elements to make place for the insertion at the position found. These approaches are expensive in terms of memory movements because a single insertion of a new element on a dense array may cause the displacement of a large extent of memory. Alternatively, using a dynamic storage structure like a linked list would solve this problem at the expense of managing pointers to elements. However, a linked list would lose the capacity of direct indexing of elements as well as all the benefits of data locality when performing a sequential scan.

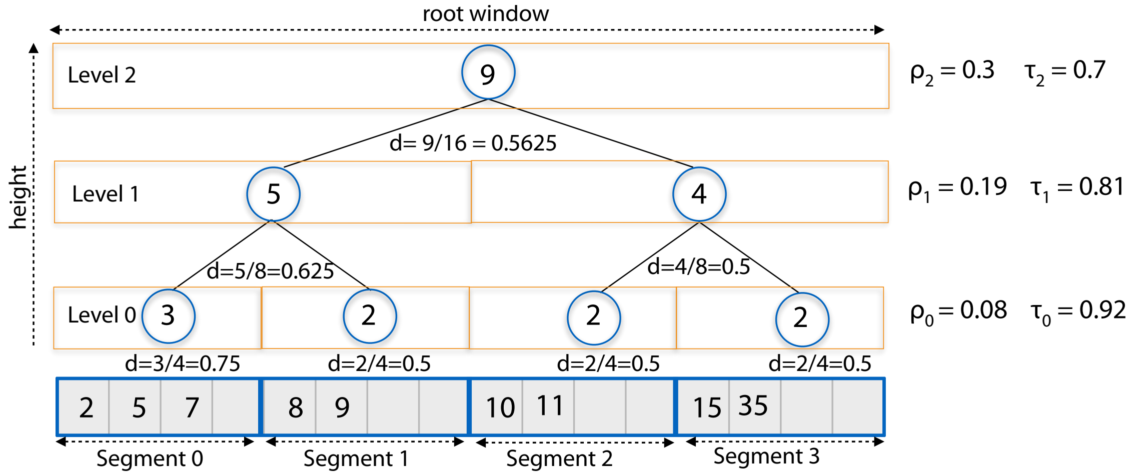
The Packed-Memory Array (PMA), first proposed by Itai et al. (1981) and later revised by Bender et al. (2005), is a data structure designed to handle modifications on sorted data. In summary, it tries to fulfill two objectives:

1. To provide efficient sequential access to sorted data.
2. To perform fast element insertion and suppression in a sorted array by having minimum memory movements.

To achieve these goals, the PMA manages a sparse array where "gaps" are kept inside the array to allow future insertions of new elements. A PMA stores N elements in an array of size $P = cN$, with $c > 1$, where the remaining $(P - N)$ positions are gaps maintained to speed-up the insertion operations. The idea behind the gaps is that each insertion will need to shift only a small and local amount of data to insert the new element in its correct position.

We give an overview of the PMA algorithm before further detailing it in the next sub-sections. Initially, gaps in the PMA are uniformly distributed in the array. After a few insertions, some sub-ranges of the array might get more populated (i.e. denser) than others. This unbalance in elements distribution negatively affects the insertion performance in the PMA as some ranges get closer to a dense array. When such situations are detected an operation called *Rebalance* takes place to redistribute the elements (and consequently the gaps) in a larger sub-range of the array. The extent of a *Rebalance* range is chosen according to thresholds of minimum and maximum densities allowed within this range. Small ranges have a high maximum threshold and a low minimum threshold, whereas large ranges have less variability between their maximum and minimum thresholds.

Figure 3.1: Example of a PMA with 4 segments. ρ_l and τ_l are the minimum and maximum allowed densities at each level of the PMA.



3.2.1.1 PMA Structure and Thresholds

We make a concise description of the sequential PMA algorithm of Bender et al. (2005) with the batch insertion scheme proposed by Durand et al. (2012), and we refer to these publications for more details.

The PMA array (size P counting the gaps) is divided into $\Theta(P/logP)$ consecutive *segments* of size $\Theta(logP)$. For convenience, the number of segments is chosen to be a power of 2. The array is stored in memory and keeps for each element an indexing *key* and its associated *value*. The segments are paired in a hierarchical fashion, creating a perfect binary tree structure where individual segments are the tree leaves and the root the full array. A group of segments corresponding to a tree node is called a *window*.

Windows have a bounded *density*: the ratio of the number of elements they contain over the total window capacity should always be in between the allowed density bound. The minimum and maximum density thresholds for a window at level l in the tree are respectively ρ_l and τ_l. Let $l = 0$ denote the lowest level of the tree (i.e a single segment) and $l = h$ be the root node (i.e. the full array). The density bounds are defined such that:

$$\rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0 \tag{3.1}$$

Thus, the larger a window, the more constraining its density bounds. It also implies that if a window of size i respects its density bounds, then all sub-windows also respect their density bounds. The minimum and maximum densities of windows at intermediate levels in the tree are linearly interpolated between the $[\rho_0, \rho_h]$ and $[\tau_h, \tau_0]$ thresholds as defined below:

$$\begin{aligned}\tau_l &= \tau_h + (\tau_0 - \tau_h)\frac{(h-l)}{h} &(3.2)\\ \rho_l &= \rho_h + (\rho_0 - \rho_h)\frac{(h-l)}{h} &(3.3)\end{aligned}$$

The upper and lower density thresholds, respectively, decrease and increase by $O(1/\log N)$. This $O(1/\log N)$ interval is fundamental to guarantee that an insertion or deletion requires $O(\log^2(N))$ amortized data movements. Figure 3.1 shows an example of PMA for 9 elements. The density thresholds are: $\rho_2 = 0.3$, $\rho_0 = 0.08$, $\tau_2 = 0.7$, $\tau_0 = 0.92$, and the values for ρ_1 and τ_1 as described in Equation 3.2 and Equation 3.3 above.

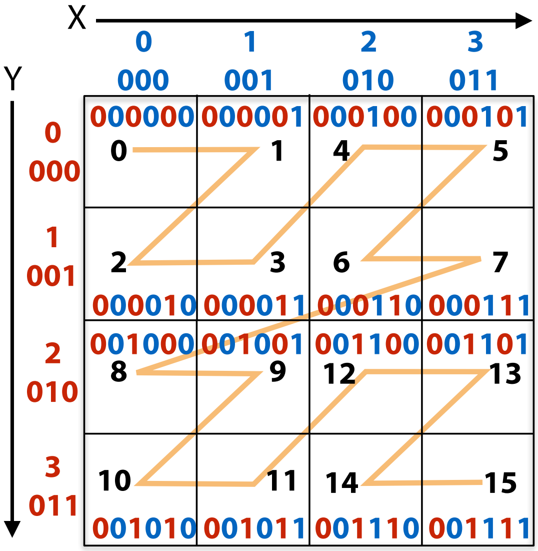
3.2.1.2 PMA Operations

Lets now consider a PMA filled with K elements: the K elements are correctly ordered and spread in the segments such that all windows respect their associated density thresholds. Now suppose that we need to insert I new elements stored in what we call the *insertion array* (Ins). The goal is to insert these new elements while keeping a PMA that respects all density thresholds. The insertion algorithm goes top-down.

First we consider the case where inserting the new elements does not cause the full array density to go over τ_h. Let p be the key of the first element of the right top window (the element 10 in Figure 3.1). We re-order the Ins array such that all elements smaller than p stay on the left side, while the others stay on the right. The left elements will go in the top left window of the PMA, the others in the top right window. We test for each top window their new densities against the corresponding thresholds, counting the elements to insert. If at least one top window does not respect the density thresholds, we *rebalance* the elements of the full array while including the new ones, i.e. we evenly redistribute all elements. Otherwise, if density thresholds are respected, the algorithm proceeds recursively on the left and right windows. In the best case *rebalances* are only required in individual segments at the bottom of the tree. If the density of the full array goes beyond τ_h counting the I new elements, we double its size and rebalance all elements. To make sure that after doubling the array size the new PMA is not below ρ_h, the top densities must respect:

$$2\rho_h < \tau_h \tag{3.4}$$

Figure 3.2: Z-curve ordering: each cell has a Morton code obtained by interleaving its X and Y-binary coordinates.



3.2.2 Z-curve

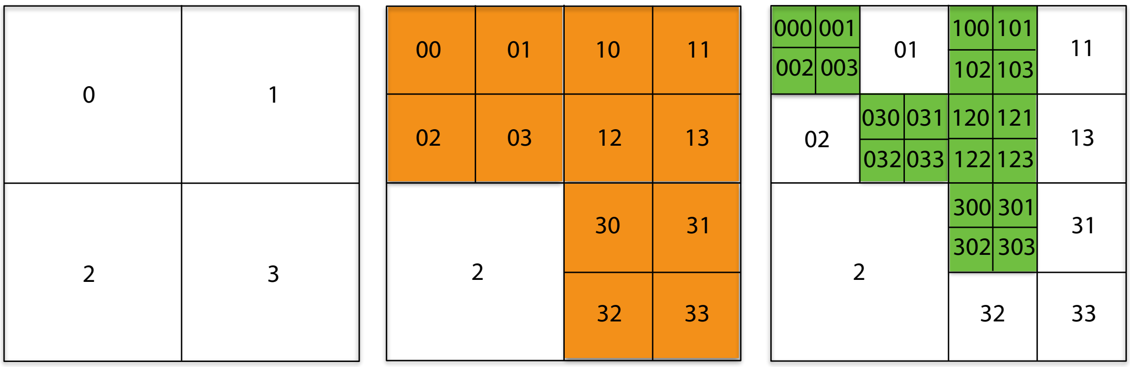
The Z-curve (or Morton curve) is a space-filling curve that maps multidimensional data into a one dimension code (the Morton code) while preserving some of the data locality. We give an example of the Z-curve in Figure 3.2. The Z-curve passes through all cells in this 4×4 grid, from the top-left corner to the bottom-right corner. The order the cells are visited corresponds to the Morton code of the cell (from 0 to 15). The Morton code can be computed using the integer coordinates of each cell by simply interleaving the binary representation of the X and Y-coordinates. Inside each cell, we display the interleaving of the X and Y binary codes used to produce the Morton code. Such ordering allows multidimensional data to be stored into any one-dimensional data structure while keeping a good spatial locality. This ordering is exploited, for example, in GPUs to store texture maps and thus increase spatial locality of reference.

3.2.3 Linear Quadtrees

Spatial-indexing techniques are commonly employed when dealing with spatial data (i.e. data with position information in space). Indexing structures based on quadtrees for instance, are widely used to speed-up spatial operations like neighborhood search and spatial queries.

Quadtrees are efficient data structures to represent geographical information (SAMET, 2005). In a quadtree, a rectangular 2D domain is recursively partitioned

Figure 3.3: Examples of linear quadtrees and codes of each cell.



into four cells (quadrants) and stored as a 4-way tree. Each one of the four quadrants receives an index from 0 to 3, and have corresponding sub-regions in the 2D domain. Each quadrant may be subject to further partitioning, resulting in an adaptive partitioning of the plane that is very efficient to answer geometric queries. Usually, the quadtree recursion terminates when a maximum depth is reached or when the leaf satisfies a threshold of maximum number of elements in it.

A suitable way to refer to nodes in the quadtree is the linear quadtree proposed by Gargantini (1982). In the linear quadtree, the code of each node aggregates a subsequent index for each level (e.g. code 123 corresponds to a quadrant following children 1, 2, and 3 respectively). We refer to this code as the *geohash* of a given node. We show these codes for all cells in the quadtree in Figure 3.3. In fact, the linear quadtree code is none other that the Morton code. Another property of this representation is that the different prefixes of a given code (for code 1203 would be 120, 12, and 1) represent the nodes on the upward path from the node 1203 to the root of the tree. Hence, this kind of data-structure is a *trie* or (a *prefix-tree*).

3.3 Related Work

Research in processing real-time streaming data spans over a wide range of related domains. One good example is real-time microblog processing like on Twitter streams. In building an efficient system to enable interactive exploration of microblogs data streams, we must deal with several challenges common to areas like in-memory big-data, stream processing, geo-spatial processing and information visualization. The real-time requirements combined with a large amount of data produced in microblogs meets the issues faced by the works on in-memory big-data processing systems and stream-processing community. At the same time, microblog posts often contain location information. As such, they can be treated as geospatial data and take benefit of several techniques used in geospatial databases like spatial indexes and spatial query languages. Some stream processing engines, like GeoInsight for MS SQL StreamInsight (KAZEMITABAR et al., 2010), are tailored for single-pass processing of the incoming data without the need to keep in memory a large window of events that would require an advanced data structure. Rather than extensively covering all these domains, we focus on the characteristics of the core data structures.

3.3.1 Adaptive Sorting Algorithms

The goal is the design of a data structure that can be dynamically updated to store streams of geospatial data while enabling the fast execution of spatiotemporal queries, such as the *top-k* query that ranks and returns only the k most relevant data matching predefined spatiotemporal criteria. One possible approach is to keep the data sorted in a (dense) array. To keep a good spatiotemporal locality, a classical approach is to rely on a space-filling curve for sorting the data in the array. Answering a range query is efficient. Data is ordered and stored continuously in memory with good locality. We benefit from the various processor optimization features for continuous data access (pages, cache, prefetching, coalesced data transfers, etc.). Using a dense array shows its limits during insertions. The cost of memory allocations can be reduced using an amortized scheme that doubles the size of the array every time it gets full. Inserting a single element takes on average $O(N)$ data movements, i.e. the number of elements to move to make room for the newly inserted element. However, elements often are inserted by batches in an already sorted array. In that case, one possible approach is to rely on adaptive sorting algorithms, a specific class of sorting algorithms able to take advantage of already sorted sequences (ESTIVILL-CASTRO; WOOD, 1992). Practically, the complexity of an adaptive algorithm ranges from $O(N)$ if the disorder is very limited up to the classical optimal bound $O(N \log(N))$. Publications on this topic mainly present theoretical results with a small number of experimental studies (COOK; KIM, 1980; MCGLINN, 1989). One exception is Timsort (PETERS, 2017), an adaptive sorting algorithm with known efficient implementations. We show experiments that compare our data structure to Timsort.

3.3.2 Tree-Based Indexes

Another classical approach is to rely on a tree of linked arrays. The B-tree (BAYER; MCCREIGHT, 1972) and its variations (BRODAL; FAGERBERG, 2003) is probably the most common data structure for databases. The UB-Tree is a B-tree for multidimensional data using space filling curves (RAMSAK et al., 2000). These structures are to our knowledge seldom used for in-memory storage with a high insertion rate. They are competitive when data access time is large enough compared to management overheads, often the case for on-disk storage. These data structures are cache-aware, i.e. to ensure cache efficiency they require a calibration according to the cache parameters of the target architecture.

The emergence of geospatial databases led to the development of a specialized tree called a R-tree (GUTTMAN, 1984), that associates a bounding box to each tree node. Several data processing and management tools have been extended to store geospatial data relying on R-trees or variations of R-trees like the *SpatiaLite* (SPATIALITE, 2017) extension for *SQLite* or *PostGis* (POSTGIS, 2017) for *PostgreSQL*. Our experiments include comparisons with both. Though such spatial libraries brought flexibility for applications in the context of traditional spatial databases, their algorithms are not adapted to swallow a large continuous stream of incoming data. Magdy et al. (2014) proposed an in-memory data structure to query and update real-time streams of tweets. Initially called *Mercury*, then *Venus* (MAGDY et al., 2016) and eventually *Kite* (MAGDY; MOKBEL, 2017) for the latest implementation (Kite is benchmarked as well in our experiments). They rely on a pyramid structure that decomposes the space into hierarchical levels. Periodically the pyramid is traversed to remove the oldest tweets to keep the memory footprint below a given budget. This idea to rely on bounding volume hierarchies is also popular in computer graphics for indexing 3D objects and accelerating collision detection (YOON; MANOCHA, 2006). One difficulty in these data structures is to ensure fast insertions while keeping the tree balanced. The data structure may also become too fragmented in memory leading to an increase of cache misses. The partitioning criteria is usually based on heuristics, but there is often no theoretical performance guarantees.

3.3.3 Cache-Oblivious Data Structures

Another direction that somehow lies in between the two formerly discussed approaches is to store all data into an array of size $\Theta(N)$ larger than the actual number of elements to store. The elements are spread in the array following a scheme that enables efficient insertions and deletions while keeping efficient range queries. Itai et al. (1981) were probably the first to propose such data structure. Bender et al. (2005) refined it, leading to the Packed Memory Array (PMA). The main idea is that by maintaining a controlled spread of gaps, insertions of new elements can be performed moving much less than $O(N)$ elements. The PMA guarantees that one element insertion only requires $O(\log^2(N))$ amortized element moves. This cost goes down to $O(\log(N)$ for random insertion patterns. Bender and Hu (2007) also proposed a more complex PMA, called adaptive PMA, that keeps this $O(\log(N))$ for specific insertion patterns like bulk insertions. The PMA is a *cache-oblivious* data structure (FRIGO et al., 1999), i.e. it is cache efficient without explicitly knowing the cache parameters. Such data structures are particularly interesting today as the memory hierarchy is getting deeper and more complex with different block sizes. Cache-oblivious data structures are seamlessly efficient in this context. Since a CPU and a GPU can share data structures, they become more complex. Bender et al. (2005),(2007) also proposed to store a B-tree on a PMA using a *van Emde Boas* layout, leading to a cache-oblivious B-tree. However, it leads to a complex data structure with no known practical implementation. The PMA also has very few known applications. Mali et al. (2013) use the PMA for dynamics graphs. Durand et al. (2012) rely on the PMA to search for neighbors in particle-based numerical simulations. They index particles in the PMA based on the Morton index computed from their 3D coordinates. They also propose an efficient insertion scheme for batches of elements, while Bender relies on single element insertions. In this work, we propose to extend the PMA for in-memory storage of streamed geospatial data.

3.3.4 Visual Analytics Data Structures

Several data structures were proposed recently for the visual analysis of big data. A common theme is the idea of pre-computing aggregations in *datacubes* proposed by Gray et al. (1997). Representative work include imMens (LIU et al., 2013), nanocubes (LINS et al., 2013), hashedcubes (PAHINS et al., 2017) and Gaussian Cubes (WANG et al., 2017).

The idea behind these systems is to offload records from tables in a database and create several pre-defined summaries called the *datacube*. To allow interactive exploration of the dataset, these summaries are optimized to fit in main memory and the aggregation is driven by the display resolution used in the visualization. Notice that these data structures therefore don't store the raw data, once a subset of keys (dimensions in the *datacube*) are fetched from the database, only the pre-computed summary results are available for exploration. In general, the summaries are a *count* on the aggregated keys.

While the goal it to support interactive exploration on *static* datasets, the main focus is to reduce the query latency and memory consumption to support larger datasets. Updates due to insertions and removals of records are not possible and construction times are usually too long to afford on-demand reconstruction of the aggregations (LINS et al., 2013; PAHINS et al., 2017).

The PMQ data structure extends the visual queries described in their work to streaming data. The main difference is that, instead of pre-computing aggregations, we maintain the raw data in a dynamic <*key-value*> store coupled with a spatial index. Additionally, we support true ad-hoc queries like aggregations that are performed on-the-fly over a time-window on the stream. We present an interface based on the PMQ that provides the same kind of heatmap visualization on a live stream instead of on a static set of records.

3.4 The Packed-Memory Quadtree

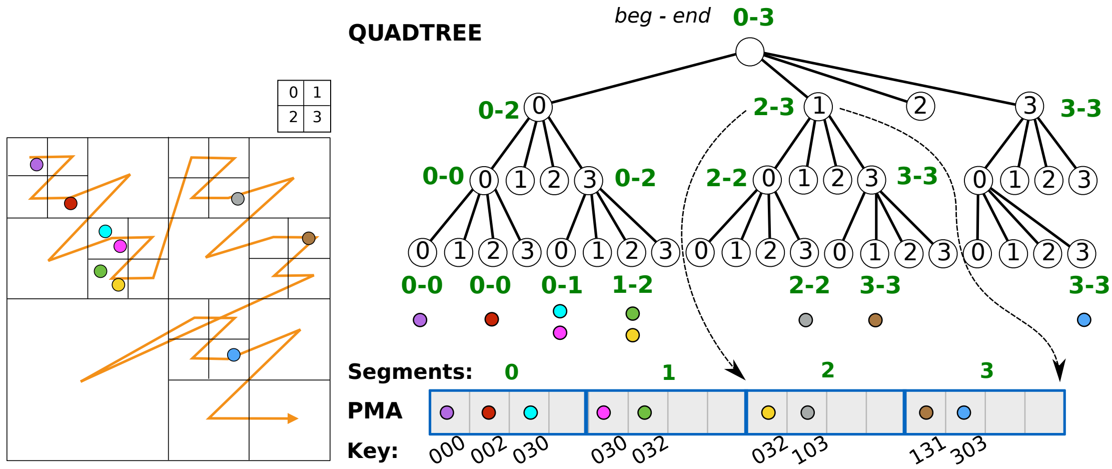
In this section, we introduce the PMQ, our proposal of a data structure for supporting spatiotemporal queries in streaming data.

3.4.1 Overview

The PMQ uses a PMA to store the streaming data, and a quadtree with its associated Z-ordering for indexing and sorting the data. Our proposal is inspired by the connection between quadtrees and the Z-ordering established by Bern et al. (1993). They show that the Z-ordering is equivalent to the ordering produced by a depth-first traversal in a quadtree. We combine this idea with the self-reorganization capabilities of the PMA to keep the stream of incoming data well sorted. The Z-ordering provides a one-dimension index used to sort the data into the PMA. It guarantees that the data records contained within the region of any quadtree node are stored contiguously in the PMA array, and thus support efficient range queries. We sort data in the PMA first by z-index and next by data timestamp.

We developed two versions of PMQ: an *explicit* (Section 3.4.3) and an *implicit*

Figure 3.4: **PMQ data structure:** Example of *Packed-Memory Quadtree* storing 9 elements spread in space as depicted in (**a**). The explicit PMQ has a quadtree as the one in (**b - top**). The elements are actually stored in the PMA array on (**b - bottom**), sorted according to their Morton index (key). Each quadtree node stores an index of the first and last PMA segment that contains elements under its sub-tree (noted as *beg-end* next to each node). A quadtree node is split down to the deepest level a soon as it contains at least one element. Notice that the quadtree cells are not necessarily aligned with segments. For instance the cell `030` (index build reading from root to leaf) contains two elements stored between segments 0 and 1 respectively. The *implicit* PMQ consists only in the PMA array (**b - bottom**) and search for a cell content is performed directly through a binary search for its Morton index.



(a) Z-order space-filling curve

(b) **Top:** Quadtree , **Bottom:** PMA

one (Section 3.4.2). The *explicit* PMQ combines a PMA with a pointer-based quadtree (Figure 3.4(b)). The quadtree creates an index that associates quadtree nodes to intervals in the PMA that store the corresponding data. Since quadtree cells and segments might not align, each quadtree node saves a *delimiter index* to the first and last PMA *segment* (*beg* and *end,* respectively) that contains its data. Because segments always contain empty gaps, we insert new data in the quadrant without having to change its indices. Of course, when an insertion violates the density thresholds, rebalancing is required and the delimiter indices need to be updated. The search for all elements inside a quadrant requires an extra cost of identifying the elements in the PMA between delimiter indices that do not belong to the expected cell. It suffices to do a sequential scan in the PMA between *beg* and *end.* Since the region defined between the delimiter indices may contain elements outside the desired quadrant, we compare the Morton codes of the quadrant with the keys in the PMA until we find the first element that is inside the quadrant of the query node. Similarly, we find the last valid element on the *end* segment.

The *implicit* PMQ does not store a quadtree. While this approach saves on the maintenance costs of the quadtree, it also makes the search more expensive. To find the elements contained in a given quadrant (aligned to the quadtree partitioning), we compute the Morton code Q of this quadrant and search in the PMA for the keys starting with the prefix Q. This search is facilitated by the z-curve ordering. We can compute the first and the last keys in this quadrant by padding ,respectively with zeros and ones, the least significant bits of the prefix code Q up to the keys' length (64 bits in our implementation). Because the keys are ordered in the PMA, we perform two binary searches for the first and last key to find the extremities of the range that contains the elements in the quadrant Q. For instance, in Figure 3.4 all the keys in quadrant `0` of the first level of the quadtree, start with the prefix `0xx` (keys: *000, 002, 030, 030, 032, 032*), and every key with this prefix is in quadrant `0`.

The PMQ is designed to support two main operations: *insertion* to integrate the data stream and *search* to answer the queries. Queries are detailed in Section 3.5. Only the insertion operations require writing into the PMA. Because we also need to comply with a given memory budget, the PMQ needs a protocol to flush the oldest data away, to a persistent storage, for instance, keeping only the most recent records. Instead of progressively removing data at each insertion, we adopt a lazy approach in which we only trigger memory stashing when an insertion is about to overflow the PMA (density at the root of the PMA goes over the τ_h threshold).

3.4.2 *Implicit* PMQ

We first detail the insertion operation for the *implicit* PMQ. We perform batch insertions as described by Magdy et al. (MAGDY et al., 2014; MAGDY et al., 2016). Consider a PMA of size P, containing N sorted elements stored in its array such that all density bounds are satisfied. The PMA that we use behaves exactly as presented in Section 3.2.1. Let Ins be the array containing a batch of I elements to insert in the PMQ, and a deletion condition $rm(x)$ that returns *true* if x can be removed. The algorithm (see Algorithm 3.1) starts at the topmost window of the PMA (the full array) by checking if the density would be violated after inserting the I new incoming elements ($N + I > \tau_h P$). Two possible paths follows, one where the PMA is close to full (Section 3.4.2.1) and the other where elements can be accommodated in the current available slots (Section 3.4.2.2).

3.4.2.1 Insertion with memory stashing

In the classical PMA, an overflow of the maximum density always requires the reallocation of a new array twice its size. In our PMQ, we first try to free some space by running a stashing procedure that checks and removes elements based on a predicate $(rm(x))$ prior to any redoubling operation. The PMQ is scanned counting the number of elements D satisfying the remove predicate $rm(x)$ (Line 6). We then check again the density bounds taking D into account $(N + I - D > \tau_h P)$. Note that it may happen that the number of elements to remove is such that the density of the array drops below the lower bound $(N + I - D < \rho_h P)$, case where the array is halved. In every case (when the array size is halved, doubled or left unchanged) the array content is rebalanced while performing the insertions and deletions, and evenly spreading elements into the array.

> **Note on remove predicate** $rm(x)$ **:** the remove predicate is used to limit the maximum amount of data that the PMQ keeps in memory. Often, the condition to remove elements is based on a difference of timestamp to keep only the records that arrived in the last T time units. Memory consumption can then be controlled by setting T based on the arrival rate of the data stream. In practice the PMQ self-stabilizes: it doubles its size until reaching a *steady* state where insertions and removal equilibrate.

3.4.2.2 Standard Recursive Insertion

In case where the density bound was already respected at the topmost window $(N+I < \tau_h P)$ (Line 5), we follow the insertion process presented in Section 3.2.1. We do not delete any element since we have the guarantee that the array contains sufficient empty gaps to insert the new elements. The algorithm enters a recursive procedure (Line 15 on Algorithm 3.1) starting at the root of the PMA, splitting the window in two and checking the densities of both sub-windows. Let us call N_l the number of valid elements in the left window and N_r the one on the right-hand side. We identify the first valid element p of the right window. We partition the elements of the arrays Ins such that all elements smaller than p are packed on the left array Ins_l while the remaining elements are on the right array Ins_r. We check the new expected density on each of the left and right windows against the threshold τ_j, $N_l + Ins_l < \tau_j P_l$ and $N_r + Ins_r < \tau_j P_r$, where j denotes the window level. If an overflow happens at this level, we rebalance the elements across both

Algorithm 3.1 PMQ algorithm for batch insertions (Ins) with supression condition ($rm(x)$).

```
procedure InsertBatch($PMA$,$Ins$,$rm()$)
    $P \leftarrow capacity(PMA)$
    $N \leftarrow size(PMA)$
    $I \leftarrow size(Ins)$
    if $N + I > \tau_h P$ then
        $D \leftarrow count\_if(P, rm())$
        if $N + I - D > \tau_h P$ then
            $double\_and\_remove(PMA, Ins, rm())$
        else if $N + I - D < \rho_h P$ then
            $halve\_and\_remove(PMA, Ins, rm())$
        else
            $remove\_and\_rebalance(PMA, Ins, rm())$
        end if
    else
        $recursive\_rebalance(PMA, Ins)$;
    end if
end procedure
```

windows while inserting the elements from Ins. Otherwise, the densities are respected for both windows and we recursively proceed on each sub-window of level $j - 1$. We have the guarantee that all sub-windows down to segments at level 0 satisfy their density bounds since densities are less constraining as the window size decreases

If the process goes all the way down to the segments without violating any threshold, the rebalance is directly performed by inserting the elements in the correct order. Note that when performing a rebalance or an insertion in a segment, we always keep the elements sorted based on their z-curve index and insertion timestamp. As the sorting in the rebalance procedure is stable, the only requirement is that the elements of the Ins array are ordered by arrival time (which is the natural order in a real-time stream). If no element from Ins needs to be inserted into the considered window, the recursion stops for this window.

The *implicit* PMQ has the PMA worst case complexity of $O(\log^2(N))$ amortized element moves per insertion (see (BENDER et al., 2005; BENDER; HU, 2007) for the proof). This algorithm is elegant as the reorganizations (rebalances) are automatically triggered when needed. No heuristic is needed to decide when to split a node or when to trigger a deletion as in (MAGDY et al., 2014; MAGDY et al., 2016). Memory allocations are only needed when doubling or halving the array.

3.4.3 *Explicit* PMQ

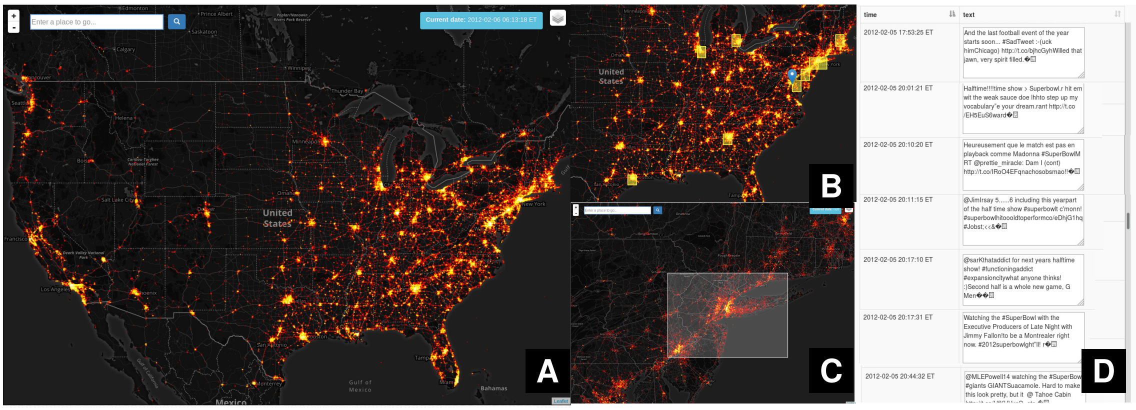
The PMA of the explicit PMQ is managed exactly like the one of the *implicit* PMQ presented before. However, the explicit PMQ has an additional quadtree structure that needs to be updated during insertions to address the correct segments in the PMA (Figure 3.4(b)). The advantage of the explicit PMQ is that rebalances often operate on small windows while large rebalances are less frequent ($O(\log^2(N)$ rebalanced elements per insertion on average). Therefore, we only need to perform local updates in the quadtree index. Keeping such index on a dense array is impracticable as any insertion array would invalidate all the following positions.

We present here how we update the quadtree. If a rebalance of the full PMA is required, we recompute the quadtree from scratch. Otherwise we update the existing quadtree as follows. First, for each new element, we build the corresponding sub-tree down to the deepest level by recursively splitting the quadrants (see Figure 3.4). If the leaf node does not exist, we allocate a new node and insert the element in the PMA. Whenever a rebalance happens in a window of the PMA, we mark as invalid all previous PMQ indexes referring to segments in this window. Therefore, the nodes in the corresponding sub-tree of the PMQ must have their delimiter indexes updated. We start by updating the leaves of this sub-tree. We scan the rebalanced window looking for the first and last appearance of the Morton code (i.e. the *key*) of each leaf node. We respectively update the delimiter indexes in each node. After the leaves were updated, these values are propagated back to the parents by return value of the recursive call. The *beg* and *end* indexes of the parent node are assigned with the corresponding delimiter indexes from its first and last child respectively.

3.5 PMQ Interface for Interactive Tweet Exploration

In this section we demonstrate the usage of our PMQ data structure in a tool for interactive exploration and visualization of a live stream of tweets. The system stores records of geo-located tweets with text and additional metadata information (like device and language) which can be queried through the visual interface. The interface displays a global view of the concentration of twitter posts by region using a heatmap representation and allows interactive zooming and selection on sub-areas (Figure 3.5). In the following sections, we describe three types of interface queries implemented and how they are processed by the PMQ data structure.

Figure 3.5: **PMQ interface:** A Twitter stream is consumed in real-time, indexed and stored in the Packed-Memory Quadtree data structure. **A** : A live heatmap is built to visualize the tweets in the current time window. **B** : Alerts are displayed on the interface indicating regions with high activity of Twitter posts at the moment. **C** : The interface allows to zoom into any region and issue queries on the current data. **D** : The actual text records can be retrieved from the Packed-Memory Quadtree to analyze what the tweets are saying in the region of interest.



time	text
2012-02-05 17:53:25 ET	And the last football event of the year starts soon... #SadTweet :-(uck himChicago) http://t.co/bjhcGyhWilled that jawn, very spirit filled.�▯
2012-02-05 20:01:21 ET	Halftime!!!!time show > Superbowl.r hit em wit the weak sauce doe lhhto step up my vocabulary”e your dream.rant http://t.co /EH5EuS6ward�▯
2012-02-05 20:10:20 ET	Heureusement que le match est pas en playback comme Madonna #SuperBowlM RT @prettie_miracle: Dam I (cont) http://t.co/IRoO4EFqnachosobsmao!!�▯
2012-02-05 20:11:15 ET	@JimIrsay 5......6 including this yearpart of the half time show #superbowlt c'monn! #superbowlhitoooldtoperformco/eDhjG1hq #Jobst;<<&�▯
2012-02-05 20:17:10 ET	@sarKthataddict for next years halftime show! #functioningaddict #expansioncitywhat anyone thinks! :)Second half is a whole new game, G Men��▯
2012-02-05 20:17:31 ET	Watching the #SuperBowl with the Executive Producers of Late Night with Jimmy Fallon!to be a Montrealer right now. #2012superbowlght”ll! r�▯
2012-02-05 20:44:32 ET	@MLEPowell14 watching the #SuperBow #giants GIANTSuacamole. Hard to make this look pretty, but it @ Tahoe Cabin

3.5.1 Heatmap Queries

The visual interface of our system relies on a heatmap view continuously updated based on the content stored in the PMQ (see Figure 3.5 **A**). We use the same tiling scheme of most map services on the World Wide Web. For instance in OpenStreetMaps[1], tiles are represented by coordinates (x, y, z). Coordinate z corresponds to the *zoom* level. For instance a tile with coordinates (0,0,0) represents the entire world map, whereas a tile at zoom 13 will map to the level of a village or town. Typically, each tile has resolution of 256×256 pixels. For a given tile with coordinates (x, y, z), a heatmap query on the PMQ must return an array with 256^2 aggregated *counts*, one for each pixel of the tile. The query algorithm traverses the PMQ quadtree down to level z and gets the quadrant indexed by (x, y). This quadrant maps to a range in the PMA. Because the tile is a square of side 2^8, the algorithm further refines this quadrant to depth $z+8$, which results in 256^2 quadrants. At this point we count, in the corresponding ranges of the PMA, the number of tweets grouped by quadrants into 256^2 bins.

Notice that we only *count* elements, since actually reading their values is not necessary. Therefore, instead of a normal scan on the array, the counting algorithm leverages an existing internal auxiliary data structure of the PMQ that keeps the count of elements per segment (implementation details are discussed later in Section 3.6).

[1] <http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames>

3.5.2 Range Queries

A range query is a spatial query that requests all elements stored in a given rectangular region (Figure 3.5 **C**). We define a range query by specifying the corners of a bounding box in the map. Given a range query, we have to access the PMQ to retrieve all records within the rectangular region. We return the result to the application for any post-processing of this information. In our interface, we currently just display a subset of the results (e.g. a fixed number of tweets – Figure 3.5 **D**). Unlike heatmaps, which queries the PMQ using a fixed resolution grid, the range query can define any arbitrary rectangular region and the result is a single list of elements contained in the area. Therefore, we need to find the *coarsest* quadrants in the quadtree that exactly match the bounding box of the range query. We use the traditional region quadtree intersecting algorithm to compute the minimal disjoint set of quadtree nodes contained in the query region (SAMET, 2005). Once the set of quadtree nodes is found, we search where its elements are stored in the PMQ. As explained in Section 3.4.1, in the *explicit* PMQ, this search is done by directly accessing the positions in the PMA between the delimiter indexes. In the case of the *implicit* PMQ, we compute the Morton codes for each node in the search set and perform binary searches for keys with prefix matching these Morton codes.

3.5.3 Top-K Queries

The third type of query combines the temporal ordering with the spatial dimension to find the *top-k* most relevant data according to a given spatiotemporal interval. It is processed like the range query but filters the candidate values in a temporary priority queue of size k.

Given a 2D point p, the top-k query finds the elements e with k lowest *bestScore* values according to a score function (Equation 3.7). The search space of the top-k queries can be reduced using the bounding parameters R and T, for spatial and time boundaries respectively. The parameter R defines a radius around p where records are going to be ranked. In the same way, parameter T limits the oldest timestamp to consider in the scoring function. The spatial and temporal scores are computed for the records returned by a range query centered at p and $2 \times R$ wide. Both scores are then normalized between $[0, 1]$ (Equation 3.5 and Equation 3.6) and combined in a final spatio-temporal score using the parameter α to balance whether spatial ($\alpha = 1$) or temporal ($\alpha = 0$) dimension is more relevant (Equation 3.7).

$$spatialScore(e, p) = \frac{distance(e.location, p)}{R} \tag{3.5}$$

$$temporalScore(e) = \frac{NOW - e.timestamp}{T} \tag{3.6}$$

$$\begin{aligned} bestScore(e, p) = \alpha \times spatialScore(e, p) \\ + (1 - \alpha) \times temporalScore(e) \end{aligned} \tag{3.7}$$

At the bottom level of the quadtree index, records (tweets) in the same cell (i.e. with the same Morton code) are ordered based on their timestamp. The top-k search uses the same refinement algorithm of range queries (Section 3.5.2) to find the records included inside the bounding box of radius R centered at p.

When scanning the PMQ range, elements that are within R and T are inserted in a priority queue of max-size k, ordered by the spatiotemploral score (Equation 3.7). Once the priority queue gets filled with the first *k* elements, we start to use the worst score W_s found so far to tighten the search boundaries $R' < R$ and $T' < T$. If $W_s < \alpha$ then $R' \leftarrow (W_s/\alpha) \times R$, and if $W_s < \alpha - 1$ then $T' \leftarrow (W_s/(\alpha - 1)) \times T$. At every new insertion on the priority queue, W_s is updated and boundaries are tightened. At the end of the scan, the queue contains the resulting elements of the top-k query.

3.6 Implementation Details

We implemented the PMQ in C++. Each element has a 64-bit *key* representing the spatial index plus a fixed-length *value* for storing additional information. We describe below our choices for the microblog content in the Twitter dataset. We use a quadtree with fixed depth of 25 levels of refinement, which requires a 50-bit length Morton code for the index. The choice of microblog content to consider depends on the application needs. If only the metadata is required, we use a 32-byte struct to store latitude, longitude, insertion timestamp, and identifiers of device, application and language. If the full tweet text is required, we use a 156-byte struct to store latitude, longitude, timestamp and a 140-bytes string.

The segments have a fixed size of eight elements. We depart from the original PMA that have segments of size $\Theta(logN)$ because we found no significant performance benefit in increasing the segment size with the array size. Fixed-size segments also make it easier to maintain segments aligned with the cache lines, a parameter that does not change across the cache hierarchy and that is often equal among a large family of processors. The PMQ is still oblivious to the cache sizes. Elements are packed at the left of a segment while leaving empty gaps on the right, as in Figure 3.1. We experienced a significant performance difference when scanning the full array compared to having gaps spread at any place in segments, which requires checking the *valid* elements inside a given region. Compact segments lead to a more regular access pattern that is friendly to low-level optimizations, such as prefetching.

In addition to the PMA array, we use an auxiliary array to store the number of valid elements in each window. This array starts at the segment level and moves up to the full array. We update it during insertions and deletions. The auxiliary array is used to estimate window densities or to speed up queries. We use this array when counting the valid elements in a given interval (in heatmap queries for example).

Before we rebalance a given window of the PMA, we copy the data to a temporary array. An in-place implementation is possible that leverages the existing empty gaps in the segments, as well as the space that becomes available in the insertion array once we merge the elements into the PMA. This implementation is more delicate but enables us to save memory space. The PMQ density thresholds were set to $\rho_2 = 0.3$, $\rho_0 = 0.08$, $\tau_2 = 0.7$ and $\tau_0 = 0.92$ which led to good performance. We found the PMQ performance not too sensitive to these parameters except at extreme values.

3.7 Performance Evaluation

In this section we conduct a series of benchmarks to evaluate the performance of the PMQ. We start by a preliminary comparison on Section 3.7.2 to confirm our hypothesis that standard database solutions together with their extensions to geospatial data are not adapted to stream processing. Following these preliminary conclusions, database solutions were judged not competitive and therefore removed from the further benchmark comparison presented from Section 3.7.3 to Section 3.7.5

The preliminary analysis also let us have a first insight about the scalability and bottlenecks of the PMQ implementation. This led us to propose the more "light weight" version of the PMQ - the *implicit PMQ* (described on Section 3.4.2) which was then added to the comparative analysis.

3.7.1 Experimental protocol

The dataset used in our benchmarks consists of geolocated tweets collected with the Twitter API between November 2011 and June 2012 over the United States. The dataset has a total of 210.6 millions tweets. We simulate an incoming stream of tweets by grouping them into batches of fixed size and iteratively inserting it into the PMQ. In the experiments with variation of the tweet arrival rate, we modify the streaming by changing the size of the batches inserted at each step of the simulation.

Time is measured for each batch insertion. Depending on the data structure, time is broken down into two operation, *index update* and *container insert*. The total insertion time is, therefore, the sum of both running times except for the *implicit PMQ,* which does not have an index structure. In this case, the total insertion time is equal to the *container insert* operation.

The benchmarks were run in a dedicated Linux machine with an Intel® Core™ i7-4790 CPU @ 3.60GHz with 24GB of main memory. Code is compiled with `GCC v5.4` and `-O3` compilation flags, and does not explicitly use any parallel processing capability.

3.7.2 Standard Database Solutions

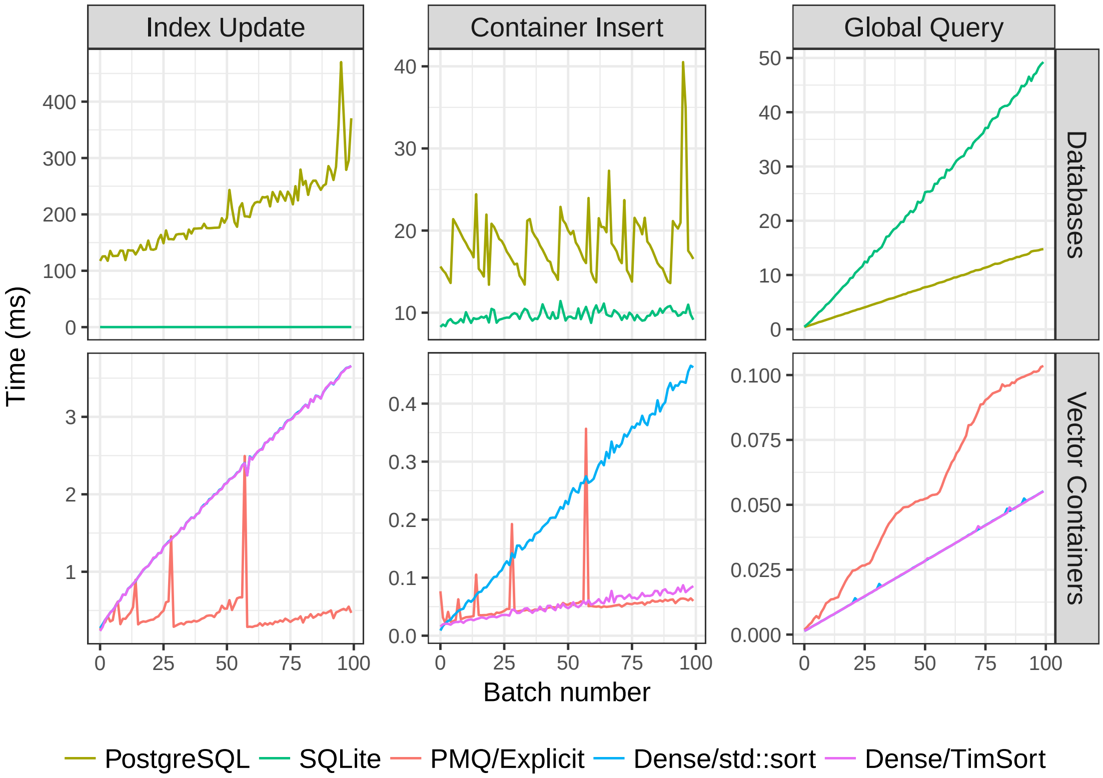
In this section we start by conducting a set of experiments to evaluate three different storage solutions for spatial data, namely:

Spatial databases: Opensource databases with geospatial library extensions – SQLite + SpatiaLite and PostgreSQL + PostGIS. SQLite uses in-memory storage, while PostgreSQL uses disk.

Dense vectors with quadtree index: Pointer-based quadtree index on a dense C++ `std::vector`. Spatial ordering of elements in the container is maintained using two sorting algorithms – the C++ `std::sort()` implementation from `GNU GCC libstdc++`, and the `C++ TimSort` adaptive sorting algorithm (GORO, 2017).

Explicit PMQ: The Packed-Memory Quadtree with an *explicit pointer-based* index described in Section 3.4.3.

Figure 3.6: Performance comparison of spatial data management solutions. **top row:** Standard geospatial databases cannot handle real-time insertions. **bottom row:** In-memory containers based on dense or sparse (PMQ) vectors indexed with pointer-based quadtrees.



This set of benchmarks gives an insight about the scalability of insertion and query operations of the solutions above. The different data structures are initially empty and increase their size as elements arrive in batches of 100 tweets. We measure the insertion time of each batch, which includes the time for updating the index and physically storing the data into the storage container. We also measure the latency for accessing data from the storage. After each batch insertion, we query all elements indexed by the data structure (Figure 3.6 *right*).

Insertion time has two major operations: *Index Update* - to insert the key of the new elements in the indexing structure (Figure 3.6 *left*); and *container insert* - to insert and reorder the records in the data structure (Figure 3.6 *middle*). As can be seen, even with a small number of elements, the database solutions have poor scalability. While PostgreSQL uses disk storage, it spends most of the time optimizing index and physically reordering elements on disk. SQLite, on the other hand, seems to have a less efficient indexing strategy than PostgreSQL. It spends less time on indexing and insertion operations, but pays a significant cost to access the data (Figure 3.6 *right*), even if stored in-memory.

None of the database solutions are suited to the real-time latency requirements of update and read operations.

For both in-memory solutions, insertion latency is dominated by the *index update* operation, which is one order of magnitude larger than *container insert*. The spikes on the PMQ benchmarks correspond to doubling the array size when the PMQ reaches the maximum density. However, the sparse storage of PMQ allows to reduce the time on *index update* when compared to the dense vectors (Figure 3.6 *bottom-left*). Finally, notice that the PMQ pays a small overhead compared to dense storage scheme when querying elements (Figure 3.6 *bottom-right*). This is an expected overhead caused by the extra management and memory transfers required by the empty slots (gaps) in the PMA.

3.7.3 PMQ Insertions

In the previous experiments, we observed that the index maintenance (*index update* operation) on quadtrees was significantly slower than the actual data insertion (*container insert*). This observation inspired the creation of the implicit PMQ, which trades-off the high cost of quadtree maintenance by slightly slower queries based on range searches.

Insertion on the PMQ is analyzed by breaking it down into two operations:

- *Index Update* - responsible for inserting the key in the quadtree index.
- *Container Insert* - responsible for storing the records *ordered* into the storage array.

We simulated the insertion of 1 million tweets inserted in batches of 100 elements at each step of the simulation. Figure 3.7 presents the insertion time of the *Explicit* PMQ and its dense-vectors counterparts. As shown, the total insertion time is dominated by *index update*, which is several orders of magnitude larger than *container insert* in all of the three methods. We note that the *index update* operation in dense arrays is much slower than on the PMQ (Figure 3.7 *left*). One explanation for this low performance is the fact that every insertion in a sorted dense array invalidates most of the pointers in the index. In practice, this is equivalent to a full index reconstruction. On the other hand, a sparse array like the PMA has more flexibility while performing local updates, which greatly reduces the average insertion time. However, we observe that worst-case insertions happen on the PMQ (clearly seen in the spikes in the PMQ benchmark in Figure 3.7). These insertions require global rebalance operations that resize the PMA, thus invalidating the full index.

With the *Implicit* setup, insertions on the PMQ and Dense container do not per-

Figure 3.7: **Insertions with *Explicit* index:** total insertion time accounts for the *index update* plus *container insert* operations. 10^6 elements are inserted by batches of size 100.

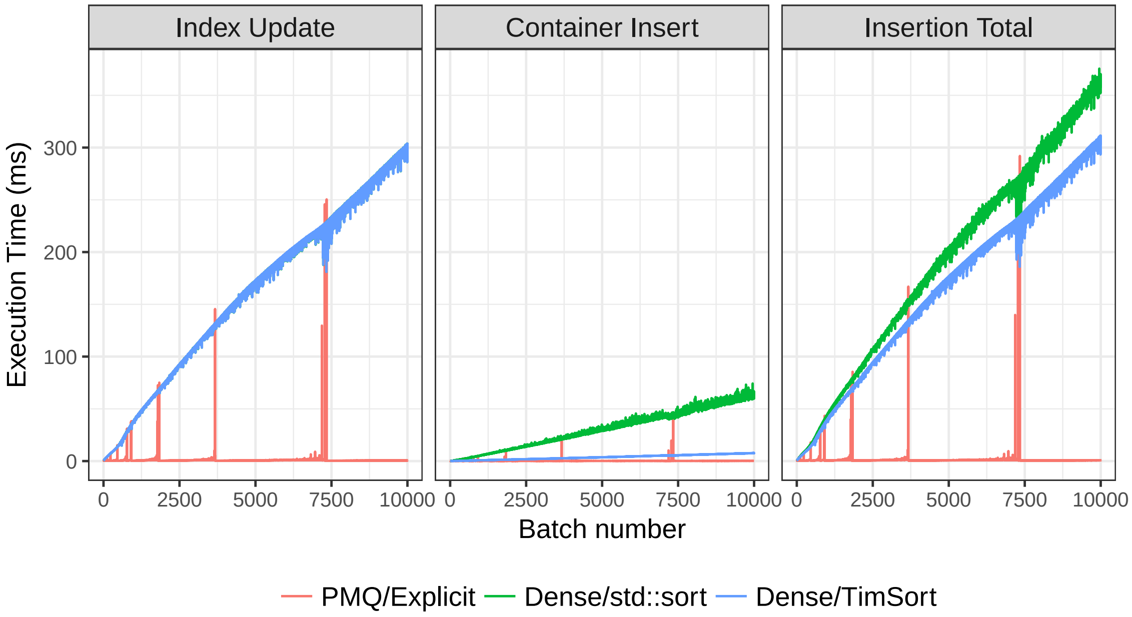


Table 3.1: **Insertions with *Implicit* index:** total insertion time is reduced to the *container insert* operation (Figure 3.7 *middle*). Mean batch insertion time of 10^4 batches.

Sorting Algorithm	**Mean (ms)**	**Max. (ms)**	**Total Time (ms)**
PMQ / Implicit	0.154	41.560	1543.454
Dense / TimSort	3.725	7.866	37251.072
Dense / std::sort	30.320	74.240	303187.010

form the costly *index update* operations. Therefore, the performance is dominated by the *container insert* alone, which depends on the algorithm used to maintain elements sorted after a batch insertion. Table 3.1 summarizes the insertion time in *implicit* setups with different order maintenance strategies. The values presented actually correspond to the time spent on the *container insert* operation (Figure 3.7 *middle*). As shown by the mean batch insertion time on Table 3.1, the *implicit* PMQ outperforms the dense array solutions by at least one order of magnitude.

3.7.4 Temporal Deletions

In streaming data, the amount of records is in unbounded by definition. Therefore the data structures must be able to evict elements when the storage capacity is depleted. The PMQ supports a lazy deletion protocol. Removals work with the concept of a time horizon T which removes elements arrived more than T time units in the past. We note that removal happens only periodically at the rebalance operations, details of this algorithm are explained in Section 3.4.2.

In the set of experiments that follows, we perform insertions on the *implicit* PMQ under a regime of fixed time horizon. At each timestep one batch is inserted. For convenience we stipulate that a simulation timestep is equivalent to one second. This way, an insertion rate of 1000 Tweets/sec means a simulation with one batch of 1000 tweets inserted by timestep. The timestamp of every tweet is set to be the Id of the batch by which it was inserted. We evaluate the scalability of the data structure varying two parameters, time horizon T (Section 3.7.4.1) and insertion rate (Section 3.7.4.2).

3.7.4.1 Time Horizon Variation

In Table 3.2 we simulate a twitter insertion rate of 1000 tweets per second. We present the average insertion time in the PMQ after reaching the steady phase, i.e after elapsed T time steps of the simulation. At this point, whenever the max capacity of the PMQ is reached, tweets older that T time units are removed. Note that during this steady phase, tweet removals neither lead to halving nor doubling the PMQ size. In this phase, the amount of tweets in the PMQ grows from $Elts_{min}$ to $Elts_{max}$ as shown in Table 3.2. The Max column corresponds to the insertion operation in which the capacity of the container is reached and consequently triggers the rebalance procedure with elements removal. Although this is a slow procedure, taking up to one second, it happens rarely

Table 3.2: **Insertion time varying time horizon**: Batches of 1K elements are inserted in an *implicit* PMQ. The amount of element in the container varies from Elts_{min} to Elts_{max}.

T	**Elts_{min}**	**Elts_{max}**	**Mean(ms)**	**99th pctl.**	**Max(ms)**
$3h$	$10.8 * 10^6$	$11.74 * 10^6$	1.209	1.066	265.613
$6h$	$21.6 * 10^6$	$23.48 * 10^6$	1.310	1.134	554.971
$9h$	$32.4 * 10^6$	$46.97 * 10^6$	1.278	1.587	1007.040
$12h$	$43.2 * 10^6$	$46.97 * 10^6$	1.423	1.321	1045.950

Table 3.3: **PMQ remove procedure:** duration of the removal operation (ms); capacity of the PMQ (number of slots); number batch insertions between removals (steady period).

Batch size	250	500	1000	2000	4000	6000	8000
Time (ms)	136.97	259.33	554.97	1034.65	2092.26	3961.14	4194.38
PMQ capacity	8×2^{20}	8×2^{21}	8×2^{22}	8×2^{23}	8×2^{24}	8×2^{25}	8×2^{25}
# Batch ins.$^{\text{tions}}$	1889	1889	1889	1889	1889	9718	1889

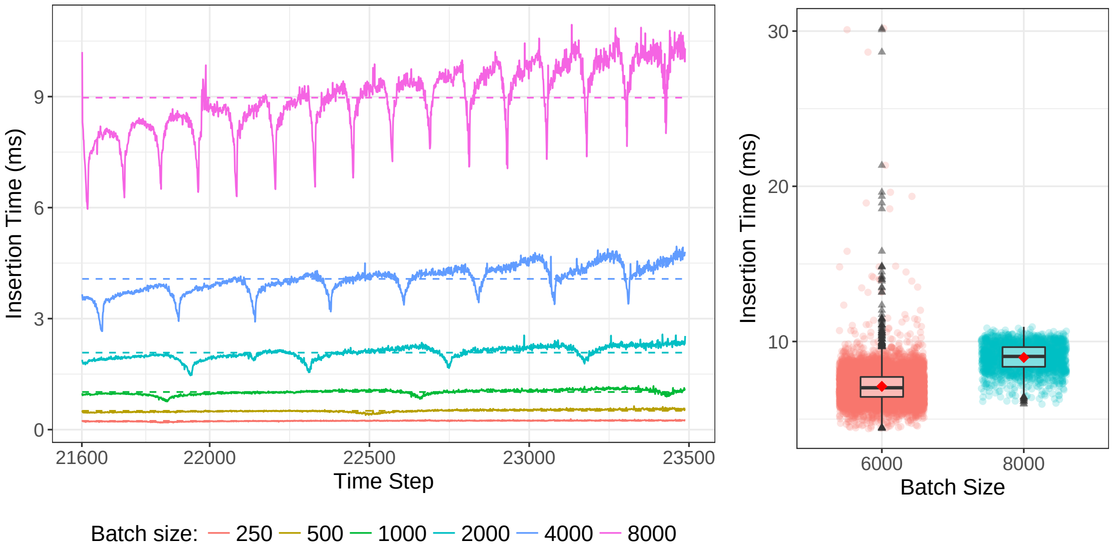
enough as shown by the 99th-percentiles. Additionally, notice that the mean execution time is much smaller than the target 1-second time step meaning that there is enough slack in the system to support an insertion rate of 1000 tweets/sec.

3.7.4.2 Variable Insertion Rate

When the time horizon is kept fixed, the size of the storage used in the PMQ will depend on the arrival rate of the stream. We simulate a scenario of a PMQ with a fixed time horizon of 6 hours (i.e. $T = 21600$ time steps) and show how the insertion time varies with the increase of the insertion rate. For each experiment we doubled the batch size between 250 and 8000 elements. Figure 3.8(a) shows the insertion time variation after the PMQ reaches the steady phase. In this picture, after the timestep $t = 21600$, removal operations will be triggered periodically every 1889 timesteps. The variation shown in this picture represents a period of a cyclic behavior, where the average time is presented by the horizontal dashed lines. The amount of elements in the PMQ increases from time step $t_0 = 21600$ to $t_1 = 23489$ by an amount of *Batch size* per time step. At $t_1 + 1$ the removal procedure takes place and all the elements inserted before $t_{old} = (t_1 + 1 - t_0)$ are removed. Notice that during the period between t_0 and t_1 the PMQ capacity stays constant. Table 3.3 presents, for each configuration of batch size tested, the execution time of the removal operation and the actual size of the PMQ array (capacity) and number of batch inserted between two removal operations.

The capacity of the PMQ increases exponentially, only doubling when the cur-

Figure 3.8: Insertion performance during steady phase of the PMQ



(a) Different PMQ size

(b) Same PMQ size.

rent capacity is reached. We separately compare the insertions using two different batch sizes (6000 and 8000) that result in same PMQ capacity. The PMQ with smaller insertion rate (6000 Tweets/timestep) will last a longer "steady" period between two removal operations (see *# batch insertions* on Table 3.3). The boxplot on Figure 3.8(b) summarize the distribution of the observed insertion times at each timestep during the steady period. The central rectangle spans the interquartile range (IQR). The red diamond in the center presents the average insertion time. The triangles denotes the "suspected outliers" which are in fact insertions happening a few time steps before the removal procedure is triggered, when the PMQ is almost full. These operations are expected to be slower than the average because the array is becoming denser and thus insertions will spend more time rebalancing elements over wider ranges of segments. Notice that the presence of outliers is not problematic, as the average time is very small, slower operations (even the removal ones) are amortized in the next batch insertions. In practice the system is very efficient and supports insertion rates well above the streaming rates of real systems. For instance the current average number of Twitter posts per *second*, worldwide, is 6000 tweets (STATS, 2017). The PMQ supports this amount of insertions under 8 ms, which represents a theoretical rate of 750.000 tweets per second.

3.7.5 Range Queries

In this section we analyze the range query performance. The user interface enables standard map navigation, like vertical and horizontal panning and zooming. A heatmap is overlayed on top of the map and shows the aggregated count of tweets per pixel. Additionally, a selection tool allows to query rectangular areas in the map to retrieve the twitter post for further inspection.

Range queries performance is influenced by two factors:

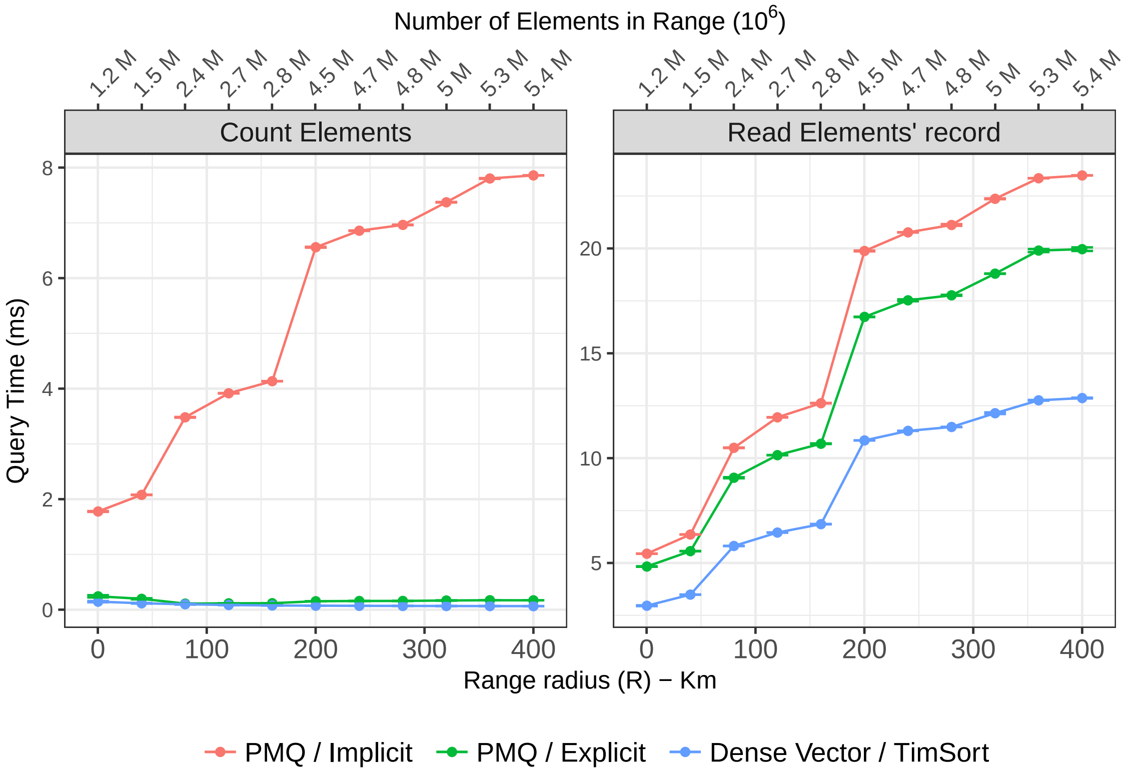
- The size and location of the selected area: for a given selected rectangle the algorithm must find the intersecting quadrants with the quadtree.
- The amount of data currently lying inside the selected area: each quadrant identified in the quadtree translates directly to a sequential scan on PMQ array. More populated areas will have longer scans.

To analyze the performance, we run a sample query centered at the coordinates of the Central Park, NY, and varied the radius of the selection from 1 to 400 KM (as illustrated in the usecase of Figure 3.12). At the time of the queries execution, the PMQ contained 21.6 M of tweets. We repeated this query three times and reported the mean execution time with standard deviation on Figure 3.9. This amount of repetitions showed to be sufficient for our analysis and no more repetitions were required.

As expected, the *implicit* PMQ search pays a small overhead compared to the other approaches using an explicit quadtree index. Notice that the difference between implicit/explicit is larger in the case of the *count* operation than on the *read* one (respectively, left and right plots of Figure 3.9). This is due to the fact that *count* operations in the PMA are done using the same internal structure used for controlling the density thresholds of the PMA segments. This operation is relatively fast and therefore the cost of this type of query is dominated by the quadtree refinement algorithm. The refinement is faster done with the *Explicit* index than with the *Implicit* PMQ algorithm, which requires several binary searches in the array to find the quadrants.

In the *read* operation, the time used to scan sequences of records from memory plays a significant role in performance, consequently the three approaches have a more similar running time. As expected, the *Dense Vector/TimSort* implementation is the fastest one because it does not have the overhead of scanning empty gaps like on the PMA. Nonetheless, the time for reading the actual tweets with the PMQ lies below 30 ms acceptable for real-time analysis.

Figure 3.9: **Range Query:** performance comparison between dense vector approach vs sparse vector (PMQ). R = radius of the square region being searched.



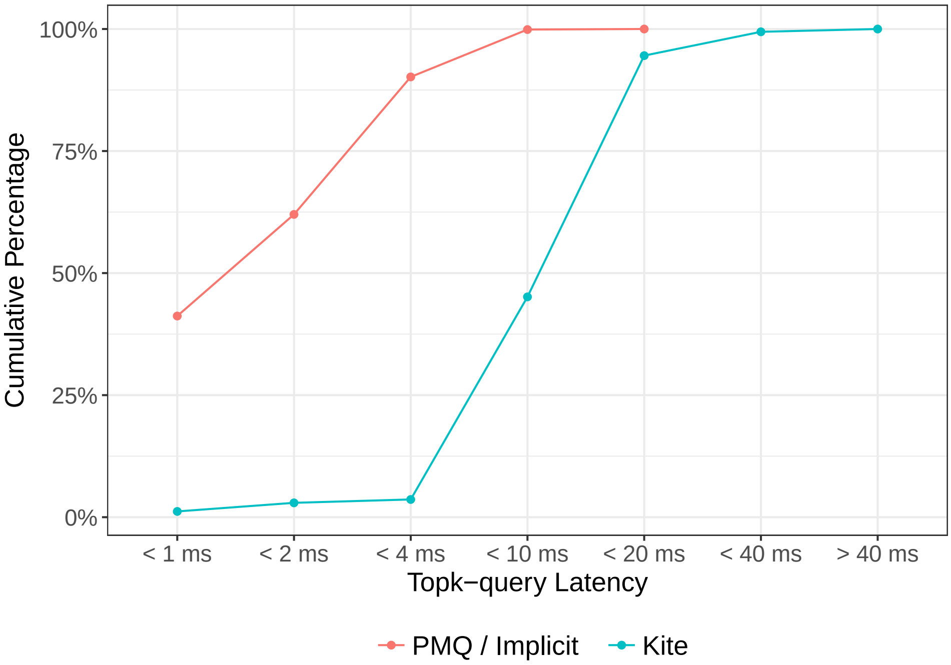
3.7.6 Top-K Performance

We compare the performance of top-k queries implemented on top of our implicit PMQ against the *Kite* framework (MAGDY; MOKBEL, 2017). We generated $10K$ top-k queries from the check-in locations of the Brightkite social network. This dataset of check-ins is publicly available in Leskovec's Stanford Large Network Dataset Collection (LESKOVEC; KREVL, 2017). The queries we generate correspond to users, in a giving location, trying to find the most relevant tweets nearby.

At the moment of execution of the top-k queries, there were $10M$ tweets stored in PMQ. For each query, we measured the latency of accessing the storage array and computing the top-k elements. The top-k ranking function (Section 3.5.3) used the default parameters values $K = 100$, $R = 30\ km$ and $T = 10000$ seconds. Temporal and spatial scores in the PMQ were balanced with $\alpha = 0.2$. *Kite* does not offer a balancing parameter for the temporal and spatial dimensions, their ranking function consider all the elements within radius r and score them according to the temporal dimension.

In Figure 3.10 we summarize the execution time of the 10.000 different queries executed. Queries were divided in bins of a cummulative histogram, which shows the

Figure 3.10: **Top-K Queries:** cumulative percentages of query latency for K = 100 , R= 30 km and T = 10000 seconds. We compare the search performance of the implicit PMQ against the *Kite* framework.



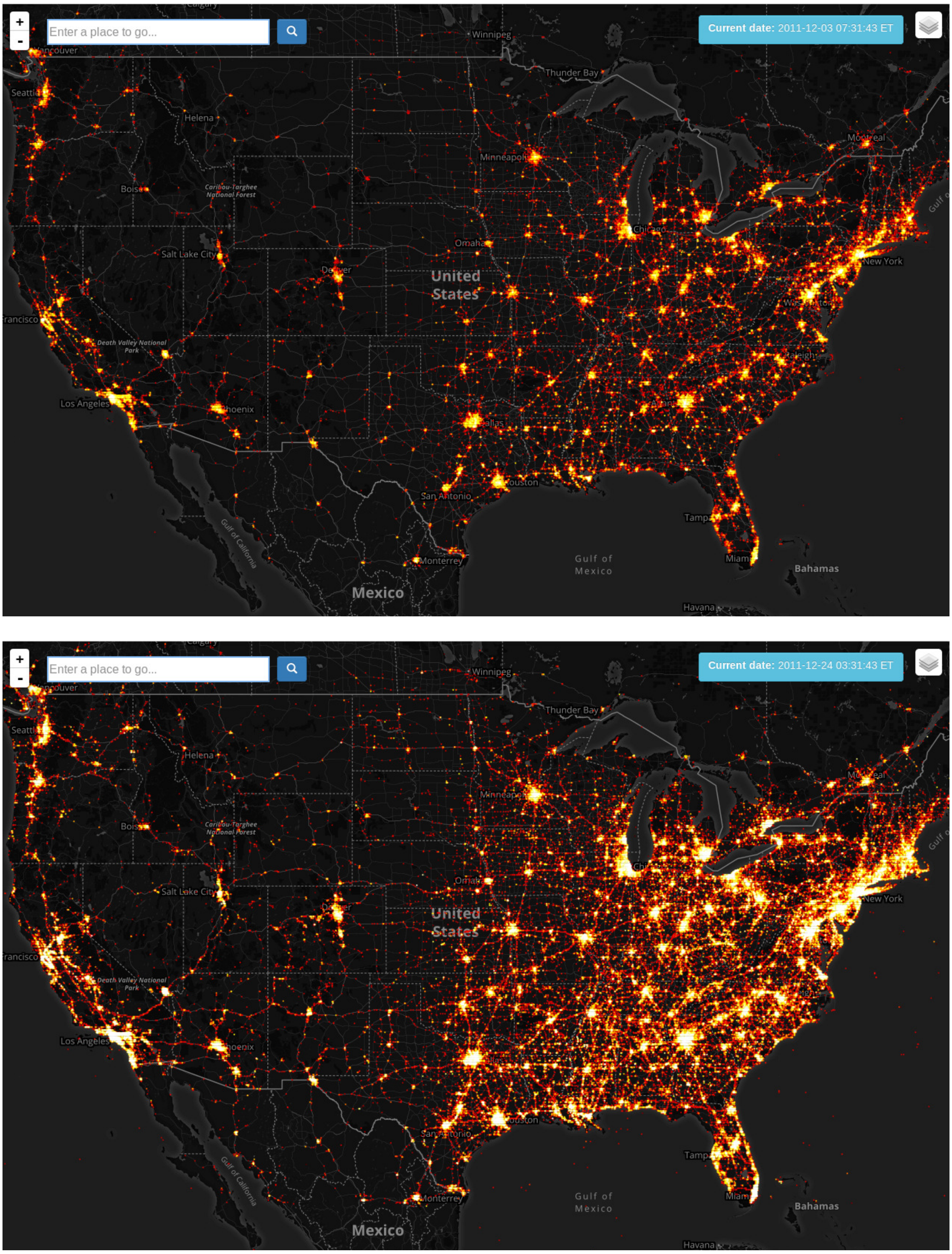
amount of queries answered under a given latency limit. The PMQ is able to answer 90% of the queries in less than 4 ms while *Kite* can only process 3% of them. *Kite* uses a regular grid as a spatial index. Since a grid of fixed resolution cannot adapt to the distribution the data, *Kite* does not perform well under common scenarios of streaming datasets.

3.8 Closing Remarks on Performance

Packed Memory Arrays have a complex performance behavior to be analyzed. Usually, its performance is measured in terms of amortized running times. However, one of their main characteristic is that its operations have variable running time (by a constant amount) and depends on the relative occupation of the storage array. Nonetheless, we have show that average insertion time is well suited for the real-time constraints.

In a real streaming system, the insertion rate is usually subjected to some variations and burst of tweets are likely to happen. In the case of an increase of the stream rate, the PMQ will naturally resize itself. Alternatively, to keep a same storage size, the time horizon (T) could be parameterized proportionally to the stream rate.

Figure 3.11: **Dynamic heatmap:** The heatmap on the interface is updated dynamically as the stream of tweets is received. With an average insertion rate of $1000\ tweets/sec$ we show the heatmap at different timesteps, when the PMQ contains $1M$ (top) and $10M$ (bottom) elements.



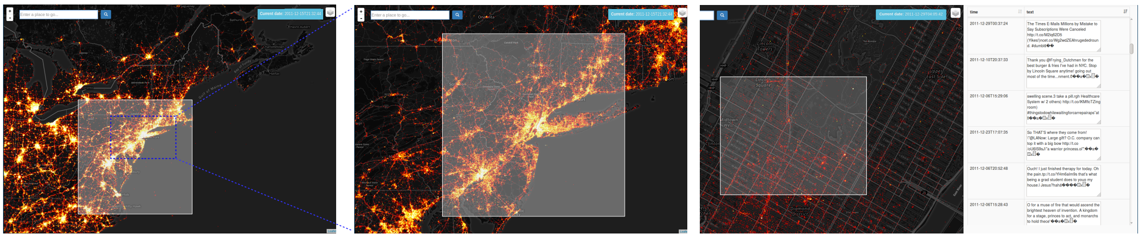
Enter a place to go...

Current date: 2011-12-03 07:31:43 ET

Enter a place to go...

Current date: 2011-12-24 03:31:43 ET

Figure 3.12: **Range queries:** Heatmap zoom and range queries are used to explore the latest streamed tweets. The in-memory storage of PMQ provides fast access to the actual tweets' content allowing real-time user interaction even on large range queries (R = radius of the selected area).



time	text
2011-12-29T00:37:24	The Times E-Mails Millions by Mistake to Say Subscriptions Were Canceled http://t.co/M2iq62D5 (Yikes!)ncet.co/Wg2wdZEAhrugededround. #dumbl$��
2011-12-10T20:37:33	Thank you @Frying_Dutchmen for the best burger & fries I've had in NYC. Stop by Lincoln Square anytime! going out most of the time...nment.0��a�□x□�
2011-12-06T15:29:06	swelling scene.3 take a pill.rgh Healthcare System w/ 2 others) http://t.co/IKMftcTZing room) #thingstodowhilewaitingforcarrepairaps"at ll��a�□x□�
2011-12-23T17:07:35	So THAT'S where they come from! \"@LANow: Large gift? O.C. company can top it with a big bow http://t.co /oU6IS9sJ\"a warrior princess.ol".��a�□x□�
2011-12-06T20:52:48	Ouch! I just finished therapy for today. Oh the pain.tp://t.co/YHm6alm9s that's what being a grad student does to youo my house.l Jesus?hahā����□x□�
2011-12-06T15:28:43	O for a muse of fire that would ascend the brightest heaven of invention. A kingdom for a stage, princes to act, and monarchs to hold thece"��a�□x□�

a) R = 400 km b) R = 200 km c) R = 1.6 km

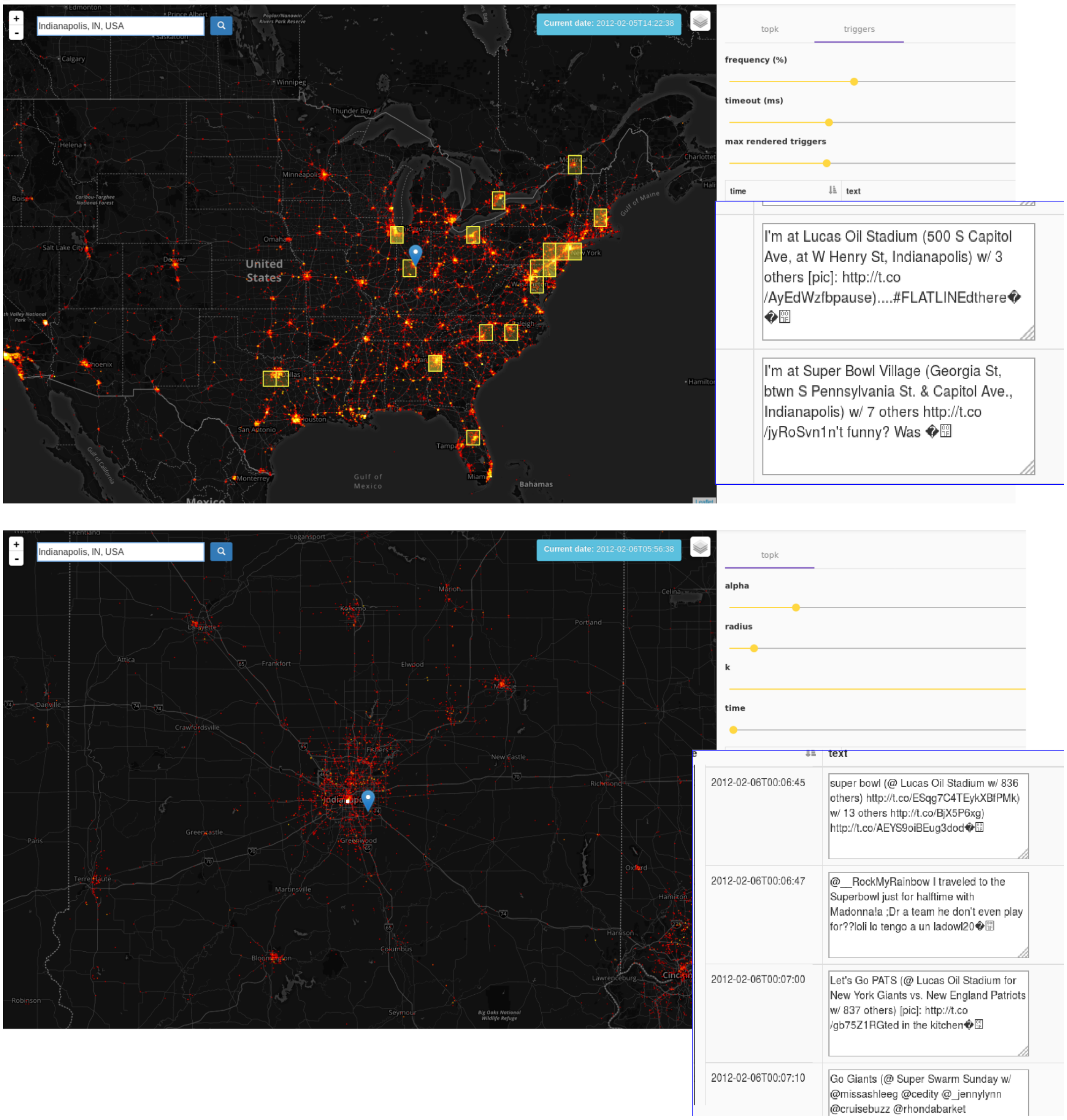
3.9 Use Cases of Interactive Exploration

We present an example of the interactive exploration of tweets enabled by the PMQ and its user interface. The PMQ allows to keep in main memory several hours of the last arrived tweets, filling the gap between stream processing engines working only on a small window of the stream, and classical solutions based on a persistent storage. The heatmap enables to display the aggregated count of tweets posted over the last hours.

In Figure 3.11 we simulate a system consuming a stream of 1000 tweets per second with a time horizon of 6 hours. The PMQ is able to index each new batch of 1000 tweets and keep all the elements sorted in less than 1.5 ms on average. The slowest performance peaks, caused by the removal procedure, take about 1 second as shown previously on Table 3.2. Notice that the PMQ and the user interface are implemented in two separate process, therefore the interaction fluidity is not affected by the longer removal operations. The PMQ performance is good enough for keeping up with the stream update rate and supporting the visual queries.

The interface allows the user to zoom into the heatmap or to perform range queries. We display the tweets inside the selected areas in a separate text area next to the map. Figure 3.12 shows the combined use of heatmaps and range queries at different zoom levels over New York. The user can interactively zoom until finding the desired information. Additionally, we implemented a simple pre-processing of batches, before its insertion on the PMQ, to trigger alerts in regions with a high percentage of tweets arrivals. Alerts indicate areas with high tweet activity (Figure 3.13). Once an alert is triggered, the user can further investigate it by interactively exploring the last received tweets. During exploration, one can also perform top-k queries to retrieve the top-most relevant tweets at a given point.

Figure 3.13: **Top-k queries:** Our system consumes an incoming stream of tweets and displays them in a dynamic heatmap. Triggers configured by the user show alerts (yellow squares) on regions with a high rate of Twitter posts. We zoom into the region of interest and filter tweets using a top-k query. The interface shows elevated tweet activities at several major US cities during the Super Bowl 2012. Zooming into Indianapolis and performing a top-k query retrieve the most relevant tweets in the area.



Indianapolis, IN, USA

Current date: 2012-02-05T14:22:38

time	text
	I'm at Lucas Oil Stadium (500 S Capitol Ave, at W Henry St, Indianapolis) w/ 3 others [pic]: http://t.co /AyEdWzfbpause)....#FLATLINEdthere� �⍰
	I'm at Super Bowl Village (Georgia St, btwn S Pennsylvania St. & Capitol Ave., Indianapolis) w/ 7 others http://t.co /jyRoSvn1n't funny? Was �⍰

Indianapolis, IN, USA

Current date: 2012-02-06T05:56:38

time	text
2012-02-06T00:06:45	super bowl (@ Lucas Oil Stadium w/ 836 others) http://t.co/ESqg7C4TEykXBfPMk) w/ 13 others http://t.co/BjX5P6xg) http://t.co/AEYS9oiBEug3dod�⍰
2012-02-06T00:06:47	@__RockMyRainbow I traveled to the Superbowl just for halftime with Madonnala ;Dr a team he don't even play for??loli lo tengo a un ladowl20�⍰
2012-02-06T00:07:00	Let's Go PATS (@ Lucas Oil Stadium for New York Giants vs. New England Patriots w/ 837 others) [pic]: http://t.co /gb75Z1RGted in the kitchen�⍰
2012-02-06T00:07:10	Go Giants (@ Super Swarm Sunday w/ @missashleeg @cedity @_jennylynn @cruisebuzz @rhondabarket

For instance, on February 5th of 2012 at 14:22 UTC, the system indicates a high tweet activity over Indianapolis. Zooming into the alert zone and using range queries, we observe that many people are at the Lucas Oil Stadium commenting about the Super Bowl game. We set a top-k query at the stadium and configure on-the-fly (T, R and α) to follow the most relevant tweets nearby. The filtered feed displayed on the right panel of the interface shows tweets with information like the teams playing (New York Giants Vs New England Patriots), or about the half-time show of Madonna (Figure 3.13).

3.10 Conclusion

We introduced PMQ, a new data structure to keep sorted the latest streamed data that can fit in a controlled memory budget. The PMQ reorganizes itself when needed with a low amortized number of data movements per insertion ($O(\log^2(N))$). Amongst the two versions we proposed, the implicit and explicit PMQ, the implicit one proved to have the best performance tradeoff between insertion and search times. Experiments demonstrated that the PMQ enables to explore several hours of the latest tweets interactively.

The PMQ can maintain a significant amount of data in memory, filling the gap between stream processing engines working only on small windows of received stream, and more classical persistent storage solutions. One direction for improvement would be to combine in-memory and persistent storage in a multi-level PMQ. The lazy stashing protocol might not adapt to some needs, as old data may stay a *long time* (up to the next top rebalance) before removal. We plan to develop a more reactive protocol for such situations. The current implementation uses the Mongoose library (MONGOOSE, 2017) for handling multiple HTTP requests concurrently, but the PMQ was not designed to support read (queries) and write (insertions) operations concurrently. To sort this out, we impose that every operation must acquire a thread lock before accessing or modifying the PMQ. All requests are thus performed sequentially limiting the volume of transactions the PMQ can support. Algorithms for supporting concurrent operations in the PMA were proposed in (BENDER et al., 2005). This work, however, is mostly focused on the correctness aspects of the concurrency management. To our knowledge, no implementation or practical experiments were conduced. The support of concurrent and parallel operations in the PMQ will be addressed in future works.

4 FINAL REMARKS

Along the development of this thesis, we approached several application-specific problems that derived into a more general one. In the quest of developing new efficient parallel algorithms for physically based simulations we approached the problem of computing a Voronoi diagram on a graph with grid-like topology (Chapter 2). We further applied this algorithm to the computation of natural neighbor interpolation in the scope of the Sofa project. A main characteristic of these algorithms is related to locality aspects of their computation. The Voronoi diagram computation is based on local distance computation from several seed points in the space. While these problems have good solutions on static scenarios (FAURE et al., 2011), these solutions can hardly be used on dynamic data were the locality relationship is constantly changing.

In another seemly unrelated field, big data and stream processing, the same problem is present: maintaining locality on dynamically changing data. The application problem, in this case, relates to the interactive exploration of spatial data, more precisely, a stream of geo-localized feed from Twitter (Chapter 3).

Despite the different context of these problems, a common concern is to identify the data structure that best handles dynamic modifications on data without compromising locality. The *Packed Memory Array* is a data structure that targets this issue. Although most literature about the PMA have proved that it shows good complexity bounds, a few have actually performed experimental performance evaluations. Additionally, with the continuous expansion of computing architecture with multiple processing units, there is the need to parallelize these solutions to leverage the benefits of the current hardware.

4.1 Bibliographic Production

During the research work of this thesis, we made an effort for publishing into high-venue conferences and journals. Regarding the specific contributions to field of physically based simulations, we presented:

- Application of the parallel Graph Voronoi to compute the natural neighbor interpolation method (TOSS et al., 2016):

  > Toss, J., Raffin, B., & Comba, J. (2016). Parallel Voronoi Computation for Physics-Based Simulations. Computing in Science and Engineering, 18(3), 88–94. <http://doi.org/10.1109/MCSE.2016.52>

- The parallel computation of the Graph Voronoi Diagram (TOSS et al., 2014):

  > Toss, J., Comba, J. L. D., & Raffin, B. (2014). Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions. In

27th SIBGRAPI - Conference on Graphics, Patterns and Images (SIBGRAPI) (pp. 212–219). <http://doi.org/10.1109/SIBGRAPI.2014.1>

Another journal paper about the *Packed-Memory Quadtree* for streaming data was submitted early this year but not accepted for publication in the *IEEE Transaction on Visualization and Computer Graphics*. Reviews however were rather constructive. According to the reviewers, rejection was justified because the contribution presented where more related to the databases community that to the visualization one. Following this feedback, we are trying a submission targeting the *Very Large Data Bases* conference (*PVLDB'2018*).

REFERENCES

BARAKAT, S. S.; TRICOCHE, X. Adaptive refinement of the flow map using sparse samples. **IEEE Transactions on Visualization and Computer Graphics**, v. 19, n. 12, p. 2753–2762, 2013. ISSN 10772626. 41

BAYER, R.; MCCREIGHT, E. Organization and Maintenance of Large Ordered Indexes. **Acta Informatica**, v. 1, p. 173–189, 1972. 56

BELIKOV, V. V. et al. The non-Sibsonian interpolation: A new method of interpolation of the values of a function on an arbitrary set of points. **Computational mathematics and mathematical physics**, Maik Nauka/Interperiodica, v. 37, n. 1, p. 9–15, 1997. 42

BENDER, M. A. et al. Cache-Oblivious B-Trees. **SIAM Journal on Computing**, IEEE Comput. Soc, v. 35, n. 2, p. 341–358, jan 2005. ISSN 0097-5397. 49, 50, 51, 57, 62

BENDER, M. A. et al. Cache-oblivious streaming B-trees. In: NINETEENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES - SPAA '07 **Procedings...** New York, New York, USA: ACM Press, 2007. p. 81. ISBN 9781595936677. 57

BENDER, M. A. et al. Concurrent cache-oblivious b-trees. In: SEVENTEENTH ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES **Procedings...** New York, NY, USA: ACM, 2005. (SPAA '05), p. 228–237. ISBN 1-58113-986-1. 81

BENDER, M. A.; HU, H. An adaptive packed-memory array. **ACM Transactions on Database Systems**, ACM, New York, NY, USA, v. 32, n. 4, p. 26–es, nov 2007. ISSN 03625915. 57, 62

BERG, M. de et al. Voronoi Diagrams. In: **Computational Geometry: Algorithms and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. chp. 7, p. 147–171. ISBN 978-3-540-77973-5. 20

BERN, M. et al. Parallel construction of quadtrees and quality triangulations. In: ALGORITHMS AND DATA STRUCTURES: THIRD WORKSHOP, WADS '93 **Procedings...** Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 188–199. ISBN 978-3-540-47918-5. 58

BEUTEL, A. et al. Natural neighbor interpolation based grid DEM construction using a GPU. In: 18TH SIGSPATIAL INTERNATIONAL CONFERENCE ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS - GIS '10 **Procedings...** [S.l.: s.n.], 2010. p. 172. ISBN 9781450304283. 41, 44

BRODAL, G. S.; FAGERBERG, R. Lower Bounds for External Memory Dictionaries. In: FOURTEENTH ANNUAL ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS **Procedings...** Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. (SODA '03), p. 546–554. ISBN 0-89871-538-5. 56

CARBONE, P. et al. Apache Flink: Unified Stream and Batch Processing in a Single Engine. **Data Engineering**, v. 36, n. 4, p. 28–38, 2015. 48

COOK, C. R.; KIM, D. J. Best sorting algorithm for nearly sorted lists. **Commun. ACM**, ACM, New York, NY, USA, v. 23, n. 11, p. 620–624, 1980. ISSN 0001-0782. 55

CORMEN, T. H. et al. Dijstrka's algorithm. In: **Introduction to Algorithms, Third Edition**. [S.l.]: MIT Press, 2009. chp. 24.3, p. 658–662. ISBN 9780262033848. 26

COURTECUISSE, H. **Nouvelles architectures parallèles pour simulations interactives médicales**. Thesis (THESE) — Université des Sciences et Thechnologies de Lille, dec 2011. 18

CRAUSER, A. et al. A parallelization of Dijkstra's shortest path algorithm. **Mathematical Foundations of Computer Science 1998**, Springer Berlin Heidelberg, p. 722–731, 1998. 26

DURAND, M. et al. A Packed Memory Array to Keep Moving Particles Sorted. In: 9TH WORKSHOP ON VIRTUAL REALITY INTERACTION AND PHYSICAL SIMULATION (VRIPHYS) **Procedings...** [S.l.: s.n.], 2012. 51, 57

EDMONDS, N. et al. Single-Source Shortest Paths With the Parallel Boost Graph Library. **The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ**, p. 1–20, 2006. 26, 27

ERWIG, M. The graph Voronoi diagram with applications. **Networks**, v. 36, n. 3, p. 156–163, oct 2000. ISSN 0028-3045. 21

ESTIVILL-CASTRO, V.; WOOD, D. A survey of adaptive sorting algorithms. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 24, n. 4, p. 441–476, dec 1992. ISSN 03600300. 55

FAURE, F. et al. SOFA: A Multi-Model Framework for Interactive Physical Simulation. In: PAYAN, Y. (Ed.). **Soft Tissue Biomechanical Modeling for Computer Assisted Surgery**. [S.l.]: Springer, 2012. p. 283–321. ISBN 978-3-642-29013-8. 32

FAURE, F. et al. Sparse meshless models of complex deformable solids. In: ACM SIGGRAPH 2011 PAPERS ON - SIGGRAPH '11 **Procedings...** New York, New York, USA: ACM Press, 2011. p. 1. ISBN 9781450309431. 18, 19, 23, 28, 29, 32, 42, 82

FRIGO, M. et al. Cache-Oblivious Algorithms. In: 40TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE **Procedings...** Washington, DC, USA: IEEE Computer Society, 1999. (FOCS '99), p. 285—-. ISBN 0-7695-0409-4. 57

GARGANTINI, I. An Effective Way to Represent Quadtrees. **Communications of the ACM**, ACM, New York, NY, USA, v. 25, n. 12, p. 905–910, dec 1982. ISSN 00010782. 49, 54

GILLES, B. et al. Frame-based elastic models. **ACM Transactions on Graphics**, v. 30, n. 2, p. 1–12, 2011. ISSN 07300301. 21

GORO, F. **C++ timsort**. 2017. <https://github.com/gfx/cpp-TimSort/>. [accessed on: 10-Jul-2017]. 68

GRAY, J. et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. **Data Mining and Knowledge Discovery**, Kluwer Academic Publishers, v. 1, n. 1, p. 29–53, 1997. ISSN 1384-5810. 57

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. **ACM SIGMOD Record**, ACM, v. 14, n. 2, p. 47, 1984. ISSN 01635808. 48, 56

HARISH, P.; NARAYANAN, P. J. Accelerating Large Graph Algorithms on the GPU Using CUDA. In: **High Performance Computing – HiPC 2007**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 197–208. 26, 27

HARISH, P. et al. **Large Graph Algorithms for Massively Multithreaded Architectures**. [S.l.], 2009. 26, 27, 28, 36

HOBEROCK, J.; BELL, N. **Thrust Parallel Algorithms Library**. 2011. <http://thrust.github.io/>. [accessed on: 10-Jul-2017]. 36

HOBEROCK, J. et al. Stream compaction for deferred shading. In: 1ST ACM CONFERENCE ON HIGH PERFORMANCE GRAPHICS - HPG '09 **Procedings...** New York, New York, USA: ACM Press, 2009. v. 1, n. 212, p. 173–180. ISBN 9781605586038. 36

HOFF, K. E. et al. Fast computation of generalized Voronoi diagrams using graphics hardware. In: 26TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES - SIGGRAPH '99 **Procedings...** New York, NY, USA: ACM Press, 1999. (SIGGRAPH '99), p. 277–286. ISBN 0201485605. 20, 24

HURTADO, F. et al. The weighted farthest color Voronoi diagram on trees and graphs. **Computational Geometry**, v. 27, n. 1, p. 13–26, jan 2004. ISSN 09257721. 21

ITAI, A. et al. A Sparse Table Implementation of Priority Queues. In: 8TH COLLOQUIUM ON AUTOMATA, LANGUAGES AND PROGRAMMING **Procedings...** [S.l.: s.n.], 1981. v. 115, n. 3, p. 417–431. ISBN 3-540-10843-2. 50, 57

KAZEMITABAR, S. J. et al. Geospatial stream query processing using Microsoft SQL Server StreamInsight. **Proceedings of the VLDB Endowment**, v. 3, n. 1-2, p. 1537–1540, sep 2010. ISSN 21508097. 55

KUMAR, S. et al. A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA. **2011 2nd International Conference on Computer and Communication Technology (ICCCT-2011)**, IEEE, p. 635–639, sep 2011. 26, 27

LARGILLIERE, F. et al. Real-time control of soft-robots using asynchronous finite element modeling. **2015 IEEE International Conference on Robotics and Automation (ICRA)**, IEEE, p. 2550–2555, may 2015. ISSN 10504729. 20

LESKOVEC, J.; KREVL, A. **SNAP Datasets: Stanford large network dataset collection**. 2017. <http://snap.stanford.edu/data>. [accessed on 10-Jul-2017]. 76

LINS, L. et al. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. **IEEE Transactions on Visualization and Computer Graphics**, IEEE, v. 19, n. 12, p. 2456–2465, dec 2013. ISSN 1077-2626. 57, 58

LIU, Z. et al. imMens: Real-time Visual Querying of Big Data. **Computer Graphics Forum**, Eurographics Association, v. 32, n. 3pt4, p. 421–430, jun 2013. ISSN 01677055. 57

MADDURI, K. et al. Parallel shortest path algorithms for solving large-scale instances. In: 9TH DIMACS IMPLEMENTATION CHALLENGE – THE SHORTEST PATH PROBLEM **Procedings...** DIMACS Center, Rutgers University, Piscataway, NJ: [s.n.], 2006. p. 1–39. 26, 27

MAGDY, A.; MOKBEL, M. **Kite System**. 2017. <http://kite.cs.umn.edu/>. [accessed on 10-Jul-2017]. 49, 56, 76

MAGDY, A. et al. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. **2014 IEEE 30th International Conference on Data Engineering**, IEEE, p. 172–183, mar 2014. ISSN 10844627. 48, 49, 56, 60, 62

MAGDY, A. et al. Venus: Scalable Real-Time Spatial Queries on Microblogs with Adaptive Load Shedding. **IEEE Transactions on Knowledge and Data Engineering**, v. 28, n. 2, p. 356–370, feb 2016. ISSN 10414347. 48, 49, 56, 60, 62

MALI, G. et al. A New Dynamic Graph Structure for Large-Scale Transportation Networks. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.: s.n.], 2013. v. 7878 LNCS, n. 288094, p. 312–323. ISBN 9783642382321. 57

MANTEAUX, P.-L. et al. Interactive detailed cutting of thin sheets. In: 8TH ACM SIGGRAPH CONFERENCE ON MOTION IN GAMES - SA '15 **Procedings...** New York, New York, USA: ACM Press, 2015. p. 125–132. ISBN 9781450339919. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2822013.2822018>. 40, 47

MCCOOL, M. et al. **Structured parallel programming: patterns for efficient computation**. [S.l.]: Elsevier, 2012. 14

MCGLINN, R. J. A parallel version of Cook and Kim's algorithm for presorted lists. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 19, n. 10, p. 917–930, 1989. ISSN 0038-0644. 55

MONGOOSE. **Mongoose - a networking library**. 2017. <https://docs.cesanta.com/mongoose/6.5/>. [accessed on 10-Jul-2017]. 81

NEALEN, A. et al. Physically based deformable models in computer graphics. **Computer Graphics Forum**, v. 25, n. 4, p. 808–836, 2006. 18

NVIDIA. **Cuda C programing Guide**. 2017. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>. [accessed on 11-Ago-2017]. 27

ORTEGA-ARRANZ, H. et al. A new GPU-based approach to the Shortest Path problem. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING & SIMULATION (HPCS) **Procedings...** [S.l.]: IEEE, 2013. p. 505–511. ISBN 978-1-4799-0838-7. 27

PAHINS, C. A. L. et al. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 671–680, jan 2017. ISSN 1077-2626. 57, 58

PARK, S. W. et al. Discrete sibson interpolation. **IEEE Transactions on Visualization and Computer Graphics**, v. 12, n. 2, p. 243–253, mar 2006. ISSN 10772626. 41, 44

PAULUS, C. et al. Augmented Reality during Cutting and Tearing of Deformable Objects To cite this version : Augmented Reality during Cutting and Tearing of Deformable Objects. **Mixed and Augmented Reality (ISMAR), 2015 IEEE International Sympo**, 2015. 20

PETERS, T. **TimSort**. 2017. <http://svn.python.org/projects/python/trunk/Objects/listsort.txt/>. [accessed on 10-Jul-2017]. 55

PETIT, B. et al. Virtualization gate. In: ACM SIGGRAPH 2009 EMERGING TECHNOLOGIES ON - SIGGRAPH '09 **Procedings...** New York, New York, USA: ACM Press, 2009. p. 1–1. ISBN 9781605588339. 20

POSTGIS. **Spatial and Geographic objects for PostgreSQL**. 2017. <http://postgis.net/>. [accessed on 10-Jul-2017]. 48, 49, 56

RAMSAK, F. et al. Integrating the UB-Tree into a Database System Kernel. **Proceedings of the 26th International Conference on Very Large Data Bases**, v. 2000, p. 263–272, 2000. 56

REED, D. A.; DONGARRA, J. Exascale computing and big data. **Communications of the ACM**, v. 58, n. 7, p. 56–68, jun 2015. ISSN 00010782. 15

REEM, D. On the possibility of simple parallel computing of Voronoi diagrams and Delaunay graphs. **CoRR**, abs/1212.1, p. 1–31, dec 2012. 25

RODRIGUEZ, A. et al. Real-time simulation of hydraulic components for interactive control of soft robots. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (ICRA) **Procedings...** Singapour, Singapore: IEEE, 2017. p. 4953–4958. ISBN 978-1-5090-4633-1. ISSN 10504729. 20

RONG, G. et al. GPU-assisted computation of centroidal Voronoi tessellation. **IEEE Transactions on Visualization and Computer Graphics**, v. 17, n. 3, p. 345–356, 2011. 20, 24, 25

RONG, G.; TAN, T.-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES - SI3D '06 **Procedings...** New York, New York, USA: ACM Press, 2006. p. 109. ISBN 159593295X. 20, 25

SAKAKI, T. et al. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors. In: 19TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB - WWW '10 **Procedings...** New York, New York, USA: [s.n.], 2010. p. 851–860. ISBN 978-1-60558-799-8. ISSN 1605587990. 16

SAMET, H. **Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0123694469. 53, 65

SIBSON, R. A Brief Description of Natural Neighbor Interpolation. In: **Interpreting Multivariate Data**. [S.l.: s.n.], 1981. v. 21, p. 21–36. ISBN 0471280399. 41

SOFA, P. **Simulaton Open Framework Architecture**. 2017. <http://www.sofa-framework.org/>. [accessed on 10-Jul-2017]. 20, 32, 43

SPATIALITE. **Spatial and geographic objects for SQLite**. 2017. <https://www.gaia-gis.it/fossil/libspatialite/index/>. [accessed on 10-Jul-2017]. 48, 49, 56

STATS, I. L. **Twitter Usage Statistics**. 2017. <http://www.internetlivestats.com/twitter-statistics/>. [accessed on 10-Jul-2017]. 74

SUKUMAR, N. Voronoi cell finite difference method for the diffusion operator on arbitrary unstructured grids. **International Journal for Numerical Methods in Engineering**, v. 57, n. 1, p. 1–34, may 2003. ISSN 00295981. 41, 42

TOSS, J. et al. Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions. In: 27TH SIBGRAPI - CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI) **Procedings...** [S.l.]: IEEE, 2014. p. 212–219. ISBN 978-1-4799-4258-9. ISSN 15301834. 82

TOSS, J. et al. Parallel Voronoi Computation for Physics-Based Simulations. **Computing in Science and Engineering**, v. 18, n. 3, p. 88–94, may 2016. ISSN 15219615. 82

WANG, Z. et al. Gaussian Cubes: Real-Time Modeling for Visual Exploration of Large Multidimensional Datasets. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 681–690, jan 2017. ISSN 1077-2626. 57

WEBER, O. et al. Parallel algorithms for approximation of distance maps on parametric surfaces. **ACM Transactions on Graphics**, v. 27, n. 4, p. 1–16, oct 2008. ISSN 07300301. 25

YOON, S.-E.; MANOCHA, D. Cache-Efficient Layouts of Bounding Volume Hierarchies. **Computer Graphics Forum**, v. 25, n. 3, p. 507–516, sep 2006. ISSN 0167-7055. 56

ZAHARIA, M. et al. Discretized streams: Fault-tolerant streaming computation at scale. In: TWENTY-FOURTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES **Procedings...** New York, NY, USA: ACM, 2013. (SOSP '13), p. 423–438. ISBN 978-1-4503-2388-8. 48

ZHANG, H. et al. In-Memory Big Data Management and Processing: A Survey. **IEEE Transactions on Knowledge and Data Engineering**, v. 27, n. 7, p. 1920–1948, jul 2015. ISSN 1041-4347. 15, 48