Universitat de Girona

# GPU PARALLEL ALGORITHMS FOR REPORTING MOVEMENT BEHAVIOUR PATTERNS IN SPATIOTEMPORAL DATABASES

**Nacho VALLADARES CERECEDA**

**Dipòsit legal: Gi. 1117-2013**
http://hdl.handle.net/10803/119544


PhD Thesis

# GPU PARALLEL ALGORITHMS FOR REPORTING MOVEMENT BEHAVIOUR PATTERNS IN SPATIOTEMPORAL DATABASES

Nacho Valladares Cereceda

2013


PhD Thesis

# GPU PARALLEL ALGORITHMS FOR REPORTING MOVEMENT BEHAVIOUR PATTERNS IN SPATIOTEMPORAL DATABASES

Nacho Valladares Cereceda
2013

Doctoral Programme in Technology
Computer Science

PhD Advisors
Dr. J. Antoni Sellarès and Dra. Marta Fort

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor at the University of Girona. Doctoral Programme in Technology.

Universitat de Girona

El Dr. J. Antoni Sellarès, professor titular del departament d'Informàtica, Matemàtica Aplicada i Estadística (IMAE) de la Universitat de Girona, i la Dra. Marta Fort professora lectora del departament d'Informàtica, Matemàtica Aplicada i Estadística (IMAE) de la Universitat de Girona,

CERTIFIQUEN:

Que aquest treball, titulat *GPU parallel algorithms for reporting movement behaviour patterns in spatiotemporal databases*, que presenta Nacho Valladares Cereceda per a l'obtenció del títol de doctor, ha estat realitzat sota la nostra direcció i que compleix els requeriments per poder optar a Menció internacional.

Girona, Juny de 2013

*GPU parallel algorithms for reporting movement behaviour patterns in spatiotemporal databases*,
Nacho Valladares Cereceda, PhD in Technology, Universitat de Girona, Informàtica, Matemàtica Aplicada i Estadística (IMAE), June 2013

*A mis padres y hermanos.*
*Porque **siempre** os tengo a mi lado...*

# ACKNOWLEDGMENTS

Primerament agrair als meus directors de Tesi en J. Antoni Sellarès i la Marta Fort el seu recolzament, ajuda i ganes de fer les coses bé. El vostre perfeccionisme acaba enganxant. La frase 'funciona, però es pot fer millor'? l'he sentit més cops dels que m'agradaria reconèixer i ens ha portat a algun desacord, però al final... sempre era millor. He après més amb aquests 4 anys amb vosaltres que tot el que portava fins llavors, i gràcies a això, crec que ja no 'estic tant verd'. Treballar amb vosaltres, és (casi sempre) divertit. Tant de bo, que algun dia ens puguem seguir divertint.

La meva família, en especial els meus pares, també els vull agrair la confiança que mostren sempre en mi. Faci el que faci, prengui la decisió que prengui. Per això els he dedicat al ells la tesi. L'han viscuda casi tant com jo.

A la meva parella, la Lara, que mereix un reconeixement a part. La seva capacitat d'escoltar-me les batalletes de despatx és inacabable. M'agrada que li agradi formar part de les meves coses i això sempre m'ajuda. Igualment, Joan, Margarita, Pau i Rosa perquè sempre us heu preocupat i interessat per la meva tesi.

Menció especial a la gent de EPI. Jordi, Victoriano, Laura, Israel i Marité. M'heu donat totes les facilitats que heu pogut i és una cosa que sempre he valorat de treballar amb vosaltres. Gracies.

Vull agrair al grup Virvig-GGG la beca de la TIN2010-20590-C02-02 que em van donar per poder tirar endavant amb aquesta tesi. També als seus membres, en especial a en Gus. La teva empenta i alegria sistemàtica es contagiosa i la passió que poses a tot és admirable. Ets molt gran.

No vull oblidar-me d'en Narcís Madern. Companys de despatx als meus inicis i als seus finals. Em vas ajudar molt quan estava verd que sempre és complicat. Igualment, a altres companys de despatx com la Tere, Fran y Raissel. Ser companys de despatx implica compartir moltes coses. M'agrada haver-les compartit.

Per últim agrair tota aquella gent que des de la distància fan que les coses siguin més senzilles. La gent del cafè del P4, els companys de tenis... però per damunt de tot, als meus amics de tota la vida Melli, Narcís, Bruna i Dani, perquè han viscut tots els minuts d'aquesta tesi. A cop de cervesa, sí, però mereixen un reconeixement.

# PUBLICATIONS

This thesis has given as a result several publications in journals and conferences in Computational Geometry:

Journals

- **Computing and visualizing popular places**
  M. Fort, J.A. Sellarès and N. Valladares
  Knowledge and Information Systems. DOI: http://dx.doi.org/10.1007/s10115-013-0639-5
- **Finding extremal sets on the GPU**
  M. Fort, J.A. Sellarès and N. Valladares
  Journal of Parallel and Distributed Computing (Accepted under minor revision. May 2013).
- **A parallel GPU-based approach for reporting flock patterns**
  M. Fort, J.A. Sellarès and N. Valladares
  International Journal of Geographical Information Science (Submitted April 2013).
- **A GPU approach to subtrajectory clustering using the Fréchet distance**
  J. Gudmundsson and N. Valladares
  IEEE Transactions on Parallel and Distributed Systems (Submitted April 2013).
- **Intersecting two families of sets on the GPU**
  M. Fort, J.A. Sellarès and N. Valladares
  Concurrency and Computation: Practice and Experience (Submitted December 2012).

Conferences

- **Reporting flock patterns on the GPU**
  M. Fort, J.A. Sellarès and N. Valladares
  *XV Spanish Meeting on Computational Geometry. Sevilla, June 26-28, 2013.*
- ***GPU approach to subtrajectory clustering using the Fréchet distance***
  *J. Gudmundsson and N. Valladares*
  20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. *Redondo Beach, California. November 2012.*
- ***Computing Popularity Maps with Graphics Hardware***
  *M. Fort, J.A. Sellarès and N. Valladares*
  27th European Workshop on Computational Geometry (EuroCG'11). *Morschach, Switzerland, March 28-30, 2011.*
- ***Computing Popular Places using Graphics Processors***
  *M. Fort, J.A. Sellarès and N. Valladares*
  *In* Proc. SSTDM'10. In cooperation with IEEE ICDM'10. *IEEE Computer Society. Sydney, December 2010.*
- ***Reporting flock patterns via GPU***
  *M. Fort, J.A. Sellarès and N. Valladares*
  *Proceedings 19th Annual Fall Workshop on Computational Geometry; Boston, USA, pp 85-86, 2009*

# CONTENTS

# RESUM

La informació que es pot extreure del moviment d'un objecte és un element clau en moltes activitats. En àmbits científics i tecnològics, l'obtenció d'aquesta informació marca la diferència en la millora dels seus processos i estudis. Amb les noves tecnologies com els GPSs o els telèfons mòbils, avui dia es poden generar conjunts de dades de persones, animals, vehicles o qualsevol objecte mòbil, normalment com a mostres de punts que representen una posició en l'espai en un determinat instant del temps. En la majoria dels casos, aquests conjunts de dades tenen volum molt gran i poden arribar a contenir estructures molt complexes. És per això que és necessari desenvolupar algoritmes eficients de mineria de dades i tècniques analítiques de visualització per extreure informació útil i rellevant d'aquests conjunts. Existeixen una sèrie de patrons de moviment que poden tenir lloc dins d'aquests conjunts de dades, mitjançant els quals, podem extreure'n informació important. Degut a la mida de l'entrada de les dades, aquesta informació pot ser molt difícil, a vegades impossible, de detectar sense utilitzar els algoritmes adients.

En els últims anys, les Unitats de Processament Gràfic (GPU) han demostrat ratis de càlcul molt alts i els seus fabricants han posat a disposició dels usuaris eines per explotar les seves capacitats de càlcul paral·lel. En aquest sentit, les GPUs s'han convertir una alternativa interesant a les tradicionals Unitats de Processament de Càlcul (CPU). Executar algoritmes, o part d'ells, a la GPU pot aportar molts beneficis a nivell d'eficiència. Tot i que la seva capacitat de càlcul redueix considerablement els temps de càlcul, és la capacitat de les GPU per realitzar càlculs en paral·lel el que les converteix en una bona alternativa. Aquesta, avui dia, s'utilitza en molts algoritmes i aplicacions.

En aquesta tesi tractem i resolem varis problemes relacionats amb el càlcul de patrons de moviment en bases de dades espai-temporals, dissenyant i implementant algoritmes paral·lels utilitzant GPUs. Primer, proposem un algoritme que utilitza els processos gràfics de la GPU per reportar el patró 'Llocs Populars', on introduïm el concepte de mapa de popularitat. Aquest, és una mesura de quantes vegades un objecte mòbil ha visitat cada punt. Després estudiem el problema de reportar tots els grups de subtrajectories d'una trajectòria, en altres paraules, busquem corbes similars dins d'una trajectòria. Per mesurar la similitud entre corbes hem triat la distancia de Fréchet. Mostrem com els actuals algorismes seqüencials es poden modificar utilitzant tècniques paral·leles que, juntament amb la potència de les GPUs, mostren millores de rendiment molt significatives. Finalment resolem el problema del patró 'Ramat'. Un 'Ramat' es refereix a un subconjunt prou gran d'entitats que es mouen properes durant un cert interval temporal. Presentem un algorisme paral·lel, per executar-se en una GPU, per reportar patrons 'Ramat' maximals. Amb aquest objectiu, presentem dos algorismes per resoldre dos problemes relacionats amb el patró 'Ramat': El problema de trobar tots els conjunts maximals de una família, i el problema de intersecar dos famílies de conjunts. Proposem algorismes paral·lels per resoldre els dos problemes que després s'utilitzen per reportar patrons 'Ramat'.

## RESUMEN

La información que se puede extraer del movimiento de un objeto es un elemento clave en muchas actividades. En ámbitos científicos y tecnológicos, la obtención de esta información marca la diferencia en la mejora de sus procesos y estudios. Con las nuevas tecnologías como los GPSs o los teléfonos móviles, hoy día se pueden generar conjuntos de datos de personas, animales, vehículos o cualquier objeto móvil, normalmente como muestras de puntos que representan una posición en el espacio en un determinado instante de tiempo. En la mayoría de los casos, estos conjuntos de datos tienen un volumen muy grande y pueden llegar a contener estructuras muy complejas. Es por esto que es necesario desarrollar algoritmos eficientes de minería de datos y técnicas analíticas de visualización para extraer información útil y relevante de estos conjuntos. Existen una serie de patrones de movimiento que pueden tener lugar dentro de estos conjuntos de datos, mediante los cuales, podemos extraer información importante. Debido al tamaño de la entrada de los datos, esta información puede ser muy difícil, a veces imposible, de detectar sin usar los algoritmos apropiados.

En los últimos años, las Unidades de Procesamiento Gráfico (GPU), han demostrado ratios de cálculo muy altos y sus fabricantes han puesto a disposición de los usuarios, herramientas para explotar sus capacidades de cálculo paralelo. En este sentido, las GPUs se han convertido en una alternativa interesante a las tradicionales Unidades de Proceso de Cálculo (CPU). Ejecutar algoritmos, o parte de ellos, en la GPU puede aportar muchos beneficios a nivel de eficiencia. Aunque su capacidad de cálculo reduce considerablemente los tiempos de cálculo, es la capacidad de las GPU para realizar cálculos en paralelo lo que realmente las convierte en una alternativa. Estas, hoy día, se usan en muchos algoritmos y aplicaciones.

En esta tesis tratamos y resolvemos varios problemas relacionados con el cálculo de patrones de movimiento en bases de datos espacio-temporales, diseñando e implementando algoritmos paralelos usando GPUs. Primero, proponemos un algoritmo que usa los procesos gráficos de la GPU para reportar el patrón 'Lugares Populares', donde introducimos el concepto de mapa de popularidad. Este, es una medida de cuantas veces un objeto móvil ha visitado cada punto. Luego, estudiamos el problema de reportar todos los grupos de subtrayectorias de una trayectoria, en otras palabras, buscamos curvas similares dentro de una trayectoria. Para medir la similitud entre curvas usamos la distancia de Fréchet. Mostramos como los actuales algoritmos secuenciales se pueden modificar usando técnicas paralelas que, juntamente con la potencia de las GPUs, muestran mejoras de rendimiento muy significativas. Finalmente resolvemos el problema del patrón 'Rebaño'. Un 'Rebaño' se refiere a un subconjunto suficientemente grande de entidades que se mueven cercanas durante un intervalo temporal. Presentamos un algoritmo paralelo, para ser ejecutado en una GPU, para reportar patrones 'Rebaño' maximales. Con este objetivo, presentamos dos algoritmos para resolver dos problemas relacionados con el patrón 'Rebaño': El problema de encontrar todos los conjuntos maximales de una familia y el problema de intersecar dos familias de conjuntos. Proponemos algoritmos paralelos para resolver los dos problemas que después se usan para reportar patrones 'Rebaño'.

ABSTRACT

Mobility is a key element of many processes and activities, and the understanding of movement is important in many areas of science and technology. With the recent advances in technologies for mobile devices, like GPS and mobile phones, we are able to generate data sets of people, animals, vehicles and other moving objects, normally available as sample points, representing a position in space in a certain instant of time. In most cases, moving object data sets are rather large in volume and complex in the structure of movement patterns they record. It is necessary to develop efficient data mining algorithms and visual analytic techniques in order to extract useful and relevant information from movement data sets. There exist a well known collection of spatiotemporal patterns which can take place inside these data sets and, by detecting them, we can reveal important information. Due to the size of the input, this information can be very difficult, and sometimes impossible, to detect without using the appropriate algorithms.

The increasing programmability and high computational rates of Graphics Processing Units (GPU) make them attractive as an alternative to CPUs for general-purpose computing. There exist some benefits like executing algorithms, or a part of them, in the GPU, releasing work from CPU. Although these tasks distribution considerably increases the algorithms efficiency, the most important feature of GPUs is their inherent parallelism that has been exploited in several algorithms and applications.

In this thesis we treat and solve various problems related to movement pattern detection by designing and implementing parallel algorithms using the GPU. We first propose a GPU pipeline based algorithm to report the 'Popular places' pattern. We introduce the popularity map, that is, a measure of how many times the moving objects of a set have visited each point. Then, we study the problem of reporting all subtrajectory clusters of a trajectory, in other words, we look for similar curves within a trajectory. To measure similarity between curves we choose the Fréchet distance. We show how the existing sequential algorithm can be modified exploiting parallel algorithms together with the GPU computational power showing substantial speed-ups. Finally we solve the 'Flock pattern'. A flock refers to a large enough subset of entities that move close to each other for a given time interval. We present a parallel approach, to be run on a GPU, for reporting maximal flocks. To this aim, we present two algorithms to solve two problems related with the 'Flock pattern': finding the maximal sets of a family and intersecting two families of sets. The GPU parallel algorithms proposed to solve these two problems are later used for reporting flock patterns.

# 1 INTRODUCTION

*The best way to predict the future*
*is to implement it.*
*– **Alan Kay***

Mobility is a key element of many processes and activities, and the understanding of movement is important in many areas of science and technology such as meteorology, biology, sociology, transportation engineering, etc. With the recent advances in technologies for mobile devices, like GPS and mobile phones, we are able to generate data sets of people, animals, vehicles and other moving objects, normally available as sample points, representing a position in space in a certain instant of time.

In most cases, moving object data sets are rather large in volume and complex in the structure of movement patterns they record. Furthermore, the amount of data has been exploding in size. For example, five years ago animals where tracked using GPS's with a frequency of one per week, now a single football game generates approximately 2.5 million coordinates in 3D, and the frequency and accuracy will increase rapidly in the future. Problems that earlier could be solved using brute-force do now require highly sophisticated algorithms and data structures. Therefore, it is necessary to develop efficient data mining algorithms and visual analytic techniques in order to extract useful and relevant information, regularities and structure from massive movement data sets. Movement patterns in such data refer to events and episodes expressed by a set of entities. Only formalized patterns are detectable by algorithms. Hence, movement patterns are modeled as any arrangement of sub-trajectories that can be sufficiently defined and formalized.

There exist a well known collection of spatiotemporal patterns based on location and motion direction, which can occur for a subset of the moving point objects at a given time step or time interval. Exact and approximate algorithms, based on techniques from Computational Geometry, are used to compute these patterns. The information that can be extracted opens up a wide range of areas where such fundamental tools are crucial in developing powerful applications.

## 1.1 MOTIVATION

The observation of behavioral patterns is crucial to animal behavior science. So far, individual and group patterns are rather directly observed than derived from racking data. However, there are more and more projects that collect animal movement by equipping them with GPS-GSMcollars. For instance, since 2003 the positions of 25 elks in Sweden are obtained every 30 minutes. Other researchers attached small GPS loggers to racing pigeons and tracked their positions every second during a pigeons journey. It is even possible to track the positions of insects, e.g. butteries or bees. Analyzing movement patterns of animals can help to understand their behavior in many different aspects. Scientists can learn about places that are popular for individual animals, or spots that are frequented by many animals. It is possible to investigate social interactions, ultimately revealing the social structure within a group of animals. A major focus lies on the investigation of leading and following behavior in socially interacting animals, such as in a flock of sheep or a pack of wolves. On a larger scale, animal movement data reflects very well the seasonal or permanent migration behavior. In the animation industry, software agents implement movement patterns in order to realistically mimic the behavior of animal groups.

Movement data of people can be collected and used in several ways. For instance, using mobile phones that communicate with a base station is one way to gather data about the approximate

locations of people. Traffic-monitoring devices such as cameras can deliver data of the movement of vehicles. This data could be used for urban planning, e.g. to plan where to build new roads or where to extend public transport. The detection of movement patterns can furthermore be used to optimize the design of location-based-services. The services offered to a moving user could not only be dependent on the actual position, but also on the estimated current activity, which may be derived from a detected movement pattern.

Patterns are used for traffic management in order to detect undesirable or even dangerous constellations of moving entities, such as traffic jams or airplane course conflicts. Traffic management applications may require basic moving object database queries, but also more sophisticated movement patterns involving not just location but also speed, movement direction and other activity parameters.

Surveillance and intelligence services might have access to more detailed data sets capturing the movement of people, e.g. credit card usage, video surveillance camera footage or maybe even GPS data. Apart from analyzing the movement data of a suspect to help prevent further crime, it is an important task to analyze the entire data set to identify suspicious behavior in the first place. This leads to define 'normal behavior' and then search the data for any outliers. Some specific activities and the corresponding movement patterns of the involved moving entities express predefined signatures that can be automatically detected in spatiotemporal or footage data. One example is that fishing boats in the sea have to report their location in fixed intervals. This is important for the coast guards in case of an emergency, but the data can also be used to identify illegal fishing in certain areas. Movement patterns have furthermore attracted huge interests in the field of spatial intelligence and disaster management. Batty et al. [1] investigated local pedestrian movement in the context of disaster evacuation where movement patterns such as congestion or crowding are key safety issues.

Professional sports are also influenced by technology. For example, some of the major tennis tournaments provide 3D reconstructions of every single point played, tracking the players and the balls. It is furthermore known that, e.g. football coaches routinely analyze match video archives to learn about an opponent's behaviors and strategies. Making use of tracking technology, the movement of the players and the ball can be described by 23 trajectories over the length of the match. Researchers were able to develop a model that is based on the interactions between the players and the ball. This model can be used to quantitatively express the performance of players, and more general, it might lead to an improved overall strategy.

In medical image field we have seen some recent applications and papers where fiber brain tracking has been increasing its importance in diagnosing. In many cases fiber tracking can be seen as a group of entities where they are moving in a 3D space. Hence, pattern recognition to compare different type of brain takes sense viewed from a geometric point of view.

In addition, relating movement patterns with the underlying geography helps us to understand where, when and ultimately why the entities move the way they do. This knowledge is essential to substantiate decision making in public and private sectors. For example: in traffic management the traffic status at the level of street network can be used for route recommendations in case of jams, incident detection, urban planning (where to build new roads) or transport planning (where to extend public transport); in tourism development it can be used to understand and manage visitor flows between various attraction locations and to better planning touristic infrastructures.

A scientist studying the immigration of birds may know how different species group together for a given season. In that sense, the flock pattern, defines a group of entities moving close together for a given time interval. They also present a natural definition of the leadership pattern "one object is leading others", which is based on behavioral patterns discussed in the behavioral ecology literature. A wildlife biologist may be interested in identifying sets of antelopes heading for some location at certain time. The time would indicate the beginning of the mating season, the selected set of entities the ready to mate individuals, and the spot might be the mating area. This is a convergence pattern. It is primarily spatial, that means the entities head for an area but may reach it at different times. On the other hand the wildlife biologist and the antelopes may share the vital interest to identify entities that head for some location and actually meet there at some

time extrapolating their current motion. Thus, the pattern encounter includes considerations about speed, excluding entities heading for a meeting range but not arriving there at a particular time with the others.

There is wide research on data mining of moving objects, in particular, on the discovery of similar trajectories or clusters. Trajectories for moving points are also referred to as (geo)spatial lifelines. In general the input is a set of n moving point objects whose locations are known at consecutive time steps, that is, the path of each moving object is a polygonal line that can self-intersect, and the output is the event produced by the movement of the entities.

In Figure 1 we can see an example of trajectories where patterns are highlighted. A flock of three entities over five time-steps, a periodic pattern where an entity shows the same spatiotemporal pattern with some periodicity, a meeting place where three entities meet for four time steps, and finally, a frequently visited location which is a region where a single entity spends a lot of time.

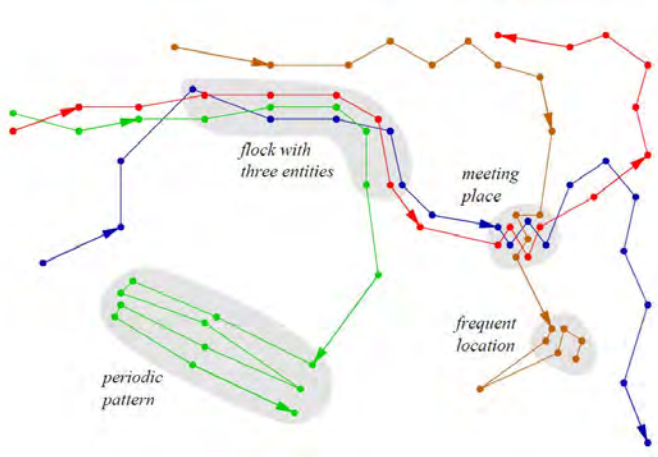



Figure 1: Illustrasting the trajectories of four entities moving over 20 time steps [2]

Several approaches are given to discover patterns. Each one defines a new algorithm or geometric interpretation, studding the variables involved. One of the first ones is the REMO framework (RElative MOtion). It was developed by Laube and Imfeld [1] to define similar behavior in groups of entities. To this end, they define a collection of spatiotemporal patterns based on similar direction of motion or change of direction. Laube et al. [3] extended the framework by not only including direction of motion, but also location itself. They defined several spatiotemporal patterns, including flock, leadership, convergence, and encounter, which can occur for a subset of the entities at a given time step or time interval. This framework established the basis for the most common movement patterns. Those were later redefined by several authors and they proposed new algorithms to compute them. In 2007, van Kreveld, Gudmunsson and Speckmann defined these patterns in [4] based on the same ideas as in the the REMO framework, but from a more computational perspective.

The study of patterns has increasingly focused on a pre-defined pattern, which means developing more accurate definitions of each pattern and more efficient algorithms to detect the patterns. For example, in 2008 Benkert et al. [5] described a new algorithm to detect the flock pattern and the same year Andersson et al. [6] presented a definition for the leadership pattern together with efficient algorithm to detect them.

Even though these are the most common movement patterns there are many other important patterns. In 2008 Dogde et al. [7] suggested a possible classification of pattern according to the information that they record. This classification opens a wide range of research of new patterns and algorithms to detect them.

Movement pattern computation is useful in huge trajectory data sets. Despite Computational Geometry techniques improve the running times, data sets of movement are so large that although this algorithms and structures are quite good, sometimes do not reach the expected time.

The increasing programmability and high computational rates of Graphics Processing Units (GPU) make them attractive as an alternative to CPUs for general-purpose computing. There exist some benefits like executing algorithms, or a part of them, in the GPU, releasing work from CPU. Although this tasks distribution considerably increases the algorithms efficiency the most important feature of GPUs is their inherent parallelism that has been exploited in several algorithms and applications [8, 9, 10, 11, 12, 13, 14, 15, 16].

As a result of technical advancements in graphics cards, some areas of 3D graphics programming have become quite complex and manufactures have provided programmers with tools to exploit the GPU parallel capabilities. There are mainly two kinds of GPU programming languages: shader languages such as Cg (C for graphics), and general purpose languages such c-CUDA (c for Compute Unified Device Architecture) or OpenCL (Open Computing Language). Programming with shader languages allows to utilize the rendering and the visualization hardware features directly. General purpose languages give access to the GPU CUDA architecture avoiding a graphics context.

Cg is a high-level shading language developed by Nvidia for programming vertex and pixel shaders. Some Computational Geometry techniques like Voronoi Diagrams have been implemented using GPU technology, demonstrating the high computational rates of the graphics hardware [17].

Nvidia CUDA is a general purpose parallel computing architecture that leverages the parallel compute engine in Nvidia graphics processing units. CUDA allows the user to think problems in a more natural way, without need of using pixels, fragments, or textures, and also taking benefit of the GPU parallelism. This, in the computational geometry field, is a great improvement.

## 1.2 CONTRIBUTIONS

Throughout our research work we treated and solved various problems related to movement pattern detection. We focus on the design and implementation of parallel algorithms using the GPU to report movement patterns in spatiotemporal databases. Next, we enumerate the different contributions of this thesis according to the different problems we solved.

**Popular places pattern.** We study the popular places pattern, that is, locations that are visited by many moving objects. We consider two criteria, strong and weak, to establish either, the exact number of times that an object has visited a place during its complete trajectory, or whether it has visited the place, or not. To solve the problem of reporting popular places, we introduce the popularity map, that is, a measure of how many times the moving objects of a set have visited that point. We propose different algorithms to efficiently compute and visualize popular places, the so-called popular regions and their schematization, by taking advantage of the parallel computing capabilities of the graphics processing units.

**Subtrajectory clustering pattern.** We study the problem of reporting all subtrajectory clusters of the trajectory. In other words, we look for similar curves within the a trajectory. To measure similarity between curves we choose the Fréchet distance. We show how the existing sequential algorithm can be modified exploiting parallel algorithms together with the GPU computational power showing substantial speed-ups. This is to the best of our knowledge not only the first GPU implementation of a subtrajectory clustering algorithm but also the first implementation using the continuous Fréchet distance, instead of the discrete Fréchet distance.

**Flock pattern.** A flock refers to a large enough subset of entities that move close to each other for a given time interval. Real-life examples are a set of cars moving in a region, a troop of tourists visiting a destination or a group of migrating animals. We present a parallel approach, to be run on a Graphics Processing Unit, for reporting maximal flocks.

**Maximal sets in a family.** The extremal sets of a family $\mathcal{F}$ of sets consist of all sets of $\mathcal{F}$ that are maximal or minimal with respect to the partial order induced by the subset relation in $\mathcal{F}$. We

present efficient parallel GPU-based algorithms, designed under CUDA architecture, for finding the extremal sets of a family $\mathcal{F}$ of sets.

**Family sets intersection.** The problem of intersecting two families of sets $\mathcal{F}$ and $\mathcal{F}'$ is to find the family $\mathcal{I}$ of all the sets which are the intersection of some set in $\mathcal{F}$ and some other set in $\mathcal{F}'$. We present an efficient parallel GPU-based approach, designed under CUDA architecture, to solve the problem.

All the contributions have experimental of the implementation of the algorithms.

## 1.3 OVERVIEW OF THE THESIS

The rest of the thesis is organized as follows:

*Chapter 2:* **Graphics Process Unit**
We introduce the Graphics Process Unit and the GPU programming model. We explain the GPU rendering pipeline and how can it be modified using shaders. We also describe the CUDA programming model.

*Chapter 3:* **Popular places pattern**
We present an algorithm to solve the continuous case of the Popular places pattern. We introduce the weak and strong criteria, that is, different criteria to measure the popularity of a point. Different approaches to compute the pattern are presented depending on the used criteria. We present the Popularity Map, and we experimental results tested with real and synthetic data sets. We also provide the time complexity of the presented algorithms.

*Chapter 4:* **Subtrajectory clustering pattern**
In this chapter, we first introduce the Fréchet distance, a measure to determine the similarity between curves. Then we propose a GPU parallel algorithm to solve the subtrajectory Clustering using the Fréchet distance as a measure of similarity. The complexity analysis of the algorithm and the experimental results show the scalability of the algorithm.

*Chapter 5:* **Maximal sets in a family**
We propose a GPU algorithm to solve the problem of finding the maximals sets of a given family. The algorithm presented in this Chapter is then used in Chapter 7 to report Flock patterns.

*Chapter 6:* **Family sets intersection**
In this Chapter we present a GPU parallel algorithm to solve the problem of intersecting two families of sets. To the best of our knowledge, this is the first time that the problem is addressed, either using the GPU or the CPU. The algorithm presented in this Chapter is then used in Chapter 7 to report the Flock pattern.

*Chapter 7:* **Flock pattern**
In this Chapter we present a GPU algorithm, based on the algorithms presented in Chapter 5 and in Chapter 6, to report Flock patterns. We show the experimental results obtained with the implementation of our algorithm.

*Chapter 8:* **Conclusions**
Finally, we provide the conclusions of the thesis.

# 2

# GRAPHICS PROCESS UNIT: A GENERAL PURPOSE COMPUTER

*Simple things should be simple, complex things should be possible.*
***– Bill Langley***

During the main chapters of the thesis we will show a number of techniques that aim to solve problems related to movement pattern detection in spatiotemporal databases using GPU parallel algorithms. For that purpose, we have taken advantage of the features that Graphics Processors Units (GPU's) make available to speed up our algorithms, taking advantage of the parallelism in the programable units.

In GPU programming there exists mainly two kinds of GPU programming languages: shader languages such as Cg (C for graphics), and general purpose languages such CUDA (Compute Unified Device Architecture) or OpenCL (Open Computing Language). Programming with shader languages allows to utilize the rendering and the visualization hardware features directly. General purpose languages give access to the GPU architecture avoiding a graphics context.

In this thesis we use both kinds of languages depending on the problem to solve. Next, in section 2.1 we present the current GPU pipeline and its programmable stages and then in Section 2.2 we present the CUDA architecture.

## 2.1 GRAPHICS HARDWARE PIPELINE

The graphics hardware pipeline, can be defined as a sequence of stages operating in parallel in a fixed order. Each stage of the pipeline receives an input from the prior stage and then it sends an output to the subsequent stage. In Figure 2 we show the graphics hardware pipeline with all the stages and programmable units. The stages that are rounded and colored in light blue are the programmable stages, making that way a flexible and adaptable pipeline to many scenarios. Following we explain each stage in more detail.

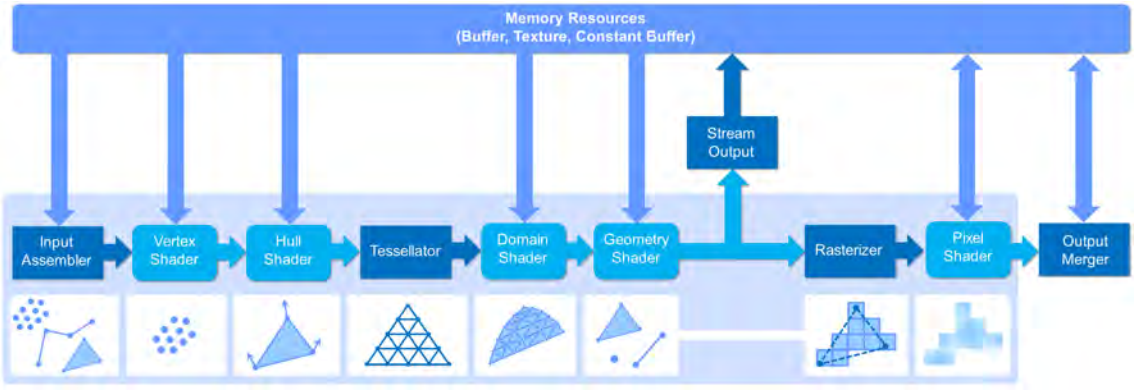



Figure 2: Graphics pipeline with all the stages and programable parts

1. **Input-Assembler Stage.** The input-assembler stage is the responsible for supplying data (triangles, lines and points) to the whole graphics pipeline from the userdefined buffers. This stage can assemble vertices into several different primitive types, such as line lists, triangle strisp, etc. New primitive types like a triangle list with adjacency have been added

to support the geometry shader stage. Another purpose of the input-assembler stage is to attach values generated by the system to help make shaders more efficient. These, system-generated values are text strings that are also called semantics, that allow to each shader stage to process only the data not yet processed. As can bee seen in the pipeline diagram (see Figure 22), once the input-assembler stage reads data from memory, assembles the data into primitives and attaches the system-generated values, the data is output to the vertex shader stage.

2. **Vertex-Shader Stage.** The vertex-shader stage is the responsible to process the vertices coming from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. One important feature from vertex shaders is that they always operate on a single input vertex, so for each input vertex they produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. That means that if our vertex shader has no special operation to perform, we have to supply at least a pass-through vertex shader. The vertex-shader stage can consume two system generated values from the input assembler: VertexID and InstanceID. Vertex shaders are ran on all vertices, including the ones supplied for adjacency purposes and can access texture information where screen-space derivates are not required.

3. **Hull-shader, tessellator, and domain-shader stages.** The hull-shader stage, the tessellator stage and the domain-shader stage comprise what is known as the tessellation stages (see Figure 22). Tessellation stages convert low-detail subdivision surfaces into higher-detail primitives on the GPU, allowing to save a lot of memory and bandwidth (data flow from CPU to GPU) when rendering a high detailed surface. It can also be used to support level-of-details (LOD) techniques in real-time. First, the hull-shader stage (programmable shader) produce a geometry patch corresponding to each input patch (quad, triangle or line). Then, the tesselator stage (fixed function pipeline) generates a sampling pattern of the domain that represents the geometry patch and generate a set of smaller objects (points, lines or triangles) that connect these samples. Finally, the domain shader (programmable shader) calculates the vertex position that corresponds to each domain sample.

4. **Geometry-Shader Stage.** The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). This is one of the main differences from the vertexshader stage, which operates on a single vertex. In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at most additional three vertices for a triangle or additional two vertices for a line. The geometry shader also supports limited geometry amplification and de-amplification, which means that given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives. The output from the geometry shader may feed the rasterizer and/or to a vertex buffer in memory through the stream output stage. This output data is done in one vertex at a time by appending vertices to an output stream object. The topology of the streams is determined by a fixed declaration, choosing one of PointStream, LineStream, or TriangleStream as the output for the geomtry stage. As in the previous vertex-shader stage, the geometry shader can perform load and texture sampling operations where screen-space derivatives are not required.

5. **Stream-Output Stage.** The stream-output stage streams primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU. There are two ways to feed stream-output data into the pipeline: First, stream-output data can be fed back into the input-assembler stage and second, stream-output data can be read by programmable shaders using loading functions.

6. **Rasterizer Stage.** The rasterization stage is the one that converts vector information, which is composed of shapes or primitives, into a raster image composed of a set of pixels. During this process each of the primitives (points, lines and triangles) are converted into pixels

and all the attributes associated to each vertex are interpolated across each primitive. In this step several other actions are taken into account: the clipping of vertices to the view frustum, the perspective correction, the mapping of the primitives to the 2D viewport and the determination of when a pixel shader is invoked. Culling of degenerate geometry, as the one giving adjacency information for the geometry shader stage, will happen in this stage.

7. **Pixel-Shader Stage.** The pixel-shader stage is the responsible to enable the generation of shading techniques in a per-pixel basis (i.e., per-pixel lighting and image post processing). Usually, a pixel shader program uses constant variables, texture data and the interpolated per-vertex values coming from previous stages to provide a per-pixel output data such as the color.
8. **Output-Merger Stage.** The output-merger stage combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

## 2.2 CUDA

The GPU is especially well-suited to address problems that can be expressed as data parallel computations. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores.

CUDA's parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions. A hierarchy of thread groups, shared memories, and barrier synchronization. That are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel. Such a decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved on any of the available processor cores.

### 2.2.1 *Programming model*

The basis of the CUDA programming model are threads. Threads are lightweight processes which are easy to create and to synchronize. The user writes programs called kernels and its execution is scheduled according to the distribution of threads. Threads are organized into blocks and blocks are organized into a grid. The grid and the block dimensions are defined according to the needs of the problem at its blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as shown in Figure 3 a).

All threads of the same kernel are executed in parallel and, typically, each one computes a result element. The GPU contains a number of multiprocessors consisting of a set of independent processors and each processor is responsible for executing one thread.

As we shown in Figure 3 b), the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

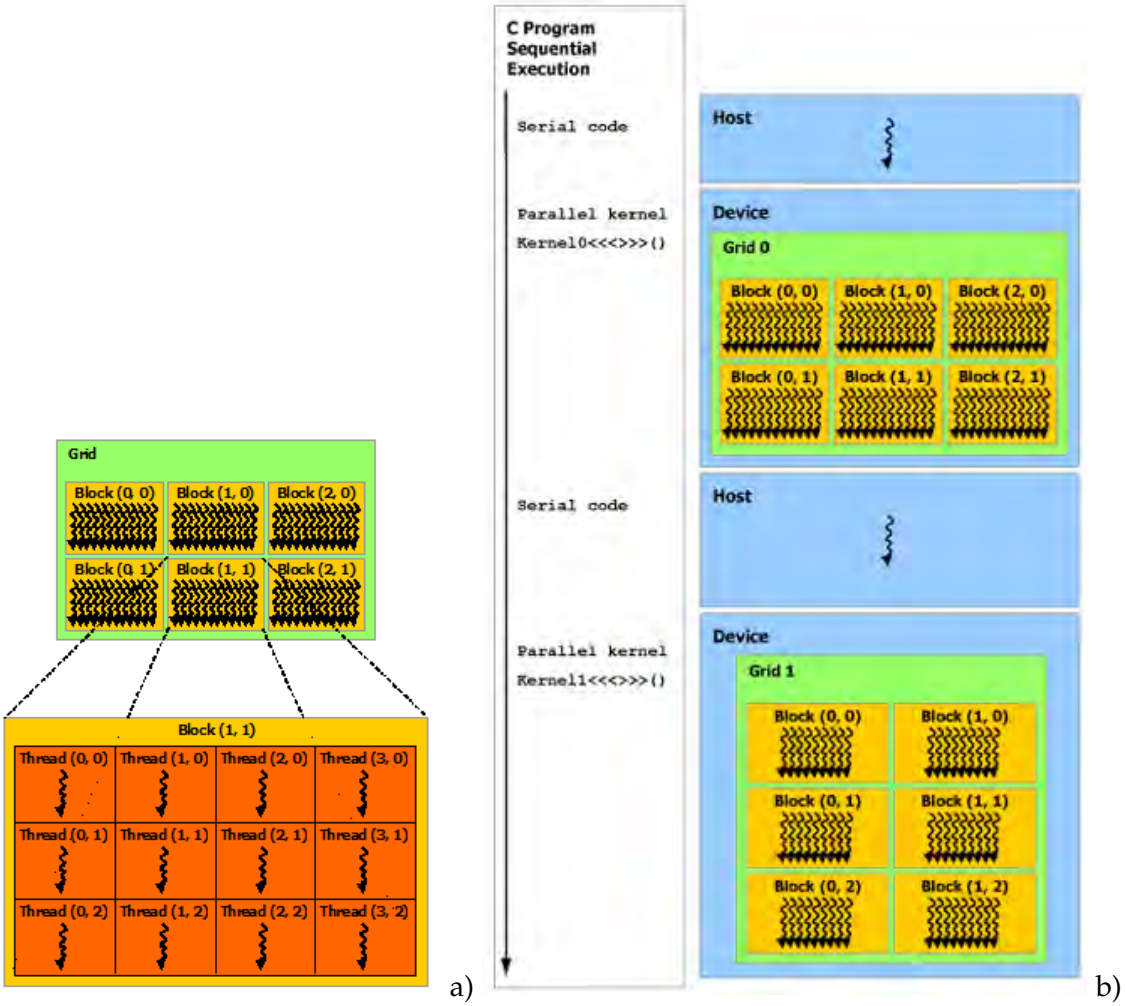



Figure 3: a) Grid of Thread Blocks. b) Heterogeneous Programming

### 2.2.2 *Memory model*

CUDA threads may access data from multiple memory spaces during their execution as shown in Figure 4. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

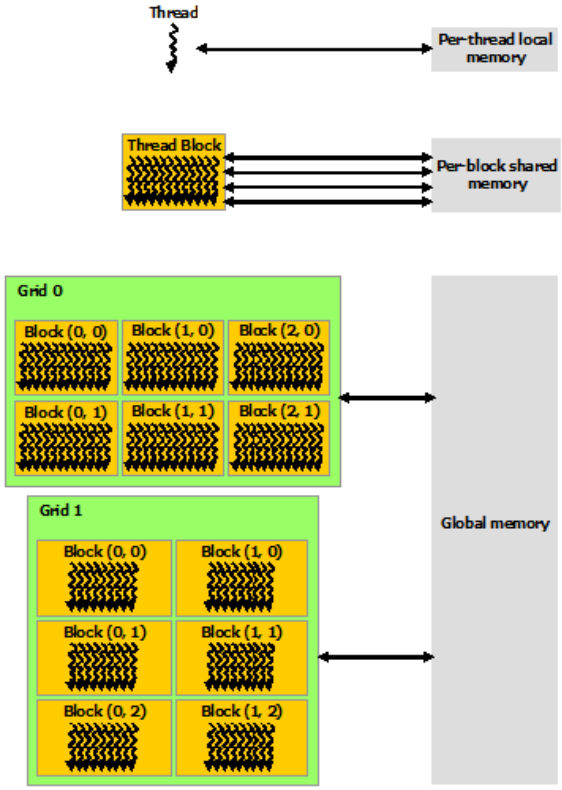



Figure 4: Memory Hierarchy

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

### 2.2.3 *Atomic functions*

CUDA concurrent memory access is a powerful feature which, in some situations, can entail simultaneous memory reads and writes causing unexpected results. This is given by the so-called thread 'race condition'. For a group of threads that are executed in parallel the thread order execution may differ between executions, even when executing exactly the same code. This follows from the fact that there is no guarantee that two or more threads reading and writing in the same position of memory will produce the same result for different executions. This is a typical scenario in parallel computing. CUDA provides a set of operations called 'atomic functions' to solve this issue.

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. This ensures the correct computation but atomic operations are very slow compared to non-atomic operations.

# 3

# POPULAR PLACES

*Real programmers can write assembly code in any language.*
***– Larry Wall***

In this Chapter we are interested in the problem of detecting popular places among trajectory paths, that is, locations that are visited by many entities. The popular place problem only reveals, without taking into account the temporal dimension of the movement, how many times a place has been visited by the entities. The detection of popular places has multiple applications in real life. In traffic analysis, for example, when we are interested in determining noise or pollution levels in highly transited areas. In tourism management, to determine locations, in a historical town, which are most frequently visited by tourists. In marketing, to ensure the effectiveness of an advertisement in a mall by determining how many people have seen it. Depending on the application, it is useful to know the exact number of times that an entity has visited the place (strong criterion) or, simply to know if it has visited the place or not (weak criterion). In the traffic analysis example, the total number of times that a car has been in a place needs to be counted when looking for the strong popular places. On the other hand, in the tourism management example, it does not matter whether a tourist has been in the place once or more than once, we are simply interested in how many different tourists have been there, in this case we are dealing with a weak popular place. In the marketing example, both criteria could be applied. Thus, we can consider that the more times you see an advertisement, the more effective it becomes i.e., considering the strong criterion, or we could take the weak criterion into consideration if we are simply interested in maximizing the number of different potential clients.

Fig. 5, which is commented on further in Section 3.3.1, presents a more detailed example. Fig. 5(a) shows the trajectories of 10,000 ants looking for food in a squared region, according to a NetLogo [18] simulation. In Fig. 5(b) and (c) the colored points are popular points according to the weak and strong criterion, respectively. They are colored from yellow to red, according to their popularity. The white areas are the not popular ones, and the red ones are the most popular or most visited ones. The red areas in Fig. 5 (b), have been visited by the biggest number of different ants during the simulation. Meanwhile, the areas in Fig. 5 (c) have received the greatest number of visits. Notice that the yellow area in Fig. 5 (c) is bigger than that in Fig. 5 (b) meaning that the same ant has gone through the same points more than once, thus increasing the number of visits from the weak to the strong criterion. Therefore, when an ant explores an area as it looks for food, it goes through the same points more than once. In both images, we can identify a circle in the center and which corresponds to the anthill. However, it only becomes one of the most popular places with the weak criterion. The food locations are also visible for both criteria, they are the red region to the north-west of the anthill, the red weak region to the south-west, and the blue east region. Notice that, while no information can be obtained from the original trajectories in Fig. 5(a), once the popular places problem is solved, useful and interesting information can be extracted from Fig. 5(b) and (c), subsequently demonstrating that the popular places is an interesting problem to solve.

Popular places were first studied by Benkert et al. in [19]. They looked for regions that had been visited by many entities, according to their trajectories. The authors model trajectory paths as 2D polygonal lines defined by trajectory points, without considering their temporal aspect. Two different models are presented: a discrete model, where only the vertices of the trajectory paths are considered, and a continuous model, where the entire trajectory paths are taken into account, along with the vertices and the line segments joining the vertices of two consecutive time-steps

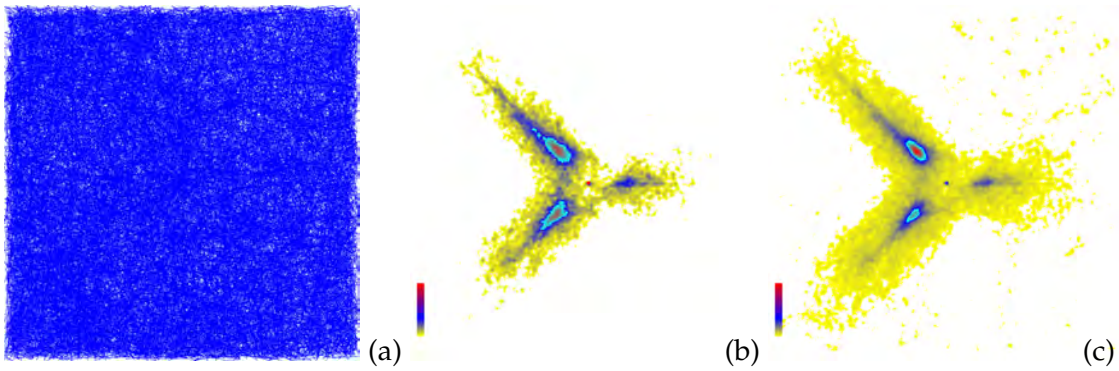



Figure 5: (a) the trajectory paths followed by the ants. (b) and (c) the popular regions under the weak and strong criterion coloured according to popularity. White regions are not popular, yellow regions have little popularity, and the red regions are the most popular ones.

(see Fig. 6). According to their definition, given a set $T$ of $n$ trajectory paths, each with $\tau$ vertices, $r > 0$, a real value modeling proximity and $k > 0$, an integer value modeling popularity, a point $p$ is an $(r, k)$-popular point in the discrete model if the axis aligned square $S(p, r)$ of center $p$ and side length $r$ contain the vertices from at least $k$ different trajectory paths of $T$. In Fig. 6 (a), point $p$ is $(r, 2)$-popular because $S(p, r)$ contains vertices of two different trajectories, while $q$ is $(r, 1)$-popular, because only the vertices of one trajectory are contained in $S(q, r)$. In the continuous model, $p$ is an $(r, k)$-popular point if there are at least $k$ different trajectories paths of $T$ that intersect the square $S(p, r)$. Accordingly, in Fig. 6 (a), points $p$ and $q$ are $(r, 2)$-popular points because the two trajectories intersect either $S(p, r)$ or $S(q, r)$. After providing these definitions, Benkert et al. define the $(r, k)$-popular regions as the maximal connected set of $(r, k)$-popular points and show that these regions are polygons. Observe that, the complexity in solving the continuous case is bigger than that for the discrete case as it is harder to handle because, it considers all the points defining the trajectories instead of only its vertices. Moreover, the popular places obtained under the continuous model always contain those obtained in the discrete one. Finally in [19], the authors present algorithms to compute the collection of $(r, k)$-popular regions, $\mathcal{P}_{r,k}(T)$. These algorithms take, for the discrete and continuous model, $O(\tau n \log \tau n)$ and $O(\tau^2 n^2)$ time, respectively, and need $O(\tau n)$ and $O(\tau n + V)$ space, where $\tau n$ represents the size of $T$ and $V$ denotes the total number of vertices of $\mathcal{P}_{r,k}(T)$. They also argue that it is likely that the best algorithm to solve the continuous model of the problem requires $\Omega(\tau^2 n^2)$ time in the worst case. They do not provide any results of the implementation of their algorithms.

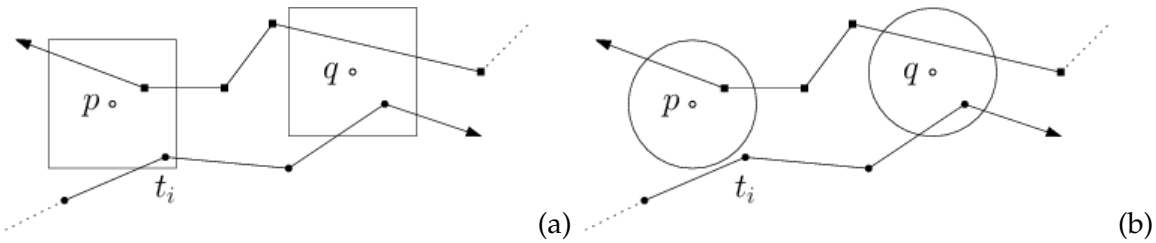



Figure 6: Discrete and continuous model examples using a square in (a) and a disk in (b).

Benkert et al. [19] end their paper by remarking that another natural way of defining a popular place is by using a disk instead of a square, and that more sophisticated techniques are needed to handle this new problem. Note that, when proximity is measured by Euclidean distance, the results obtained using a disk are more accurate that those obtained using a square, as can be seen in Fig. 6, where point $p$ has popularity 1 for the case of a disk and 2 for a square.

In this Chapter, we will focus on solving the latter problem, that of detecting popular regions in the continuous model, when proximity is determined by a disk $D(p, r)$ of center $p$ and radius

$r$, instead of the square $S(p, r)$. Moreover, to satisfy the needs of each particular application, we introduce weak and strong criteria for counting the number of intersections of a trajectory with the disk $D(p, r)$.

At times, to reduce the complexity of the popular regions and focus on the relevant information, we may be interested in obtaining a schematization by simplifying the geometry of the regions. To this end, we use the skeleton [20] to obtain a good schematization of the popular regions. The visualization methods proposed improving the understanding of the entities movement, revealing complicated structures that would be hard, if not impossible, to capture otherwise.

In this Chapter, we present algorithms for computing and visualizing popular places, in the continuous model and when proximity is determined by a disk, for both the weak and strong criteria. As we will see in Subsection 3.1.1, computing the collection of $(r, k)$-popular regions, $\mathcal{P}_{r,k}(T)$, in the worst case is likely to require quadratic time in the size of the set $T$ of trajectories. Thus, by working towards practical solutions and to obtain good running times, we use, GPU parallel computing capabilities together with Computational Geometry techniques to solve the problem. The proposed GPU algorithm is based on a discretization of the space. Although our proposed approach means reported solutions are approximated, this is acceptable as the data of moving entities are also approximated. We also describe here, the way of visualizing the popular regions obtained, and their schematization. To facilitate the analysis of trajectory data, our implementation allows constraints to be added. In the input, the trajectories are shortened by considering only those time steps contained in a selected time interval and in the output, a minimum area or length can be required in the popular regions or their schematization, respectively. Finally, we present and discuss experimental results obtained with the implementation of our algorithms.

## 3.1 BASIC DEFINITIONS

Given $E$ a set of $n$ moving entities $E = \{e_0, \ldots, e_{n-1}\}$, the **trajectory path** $t_i$ is a sequence of $\tau$ points in the plane $t_i : p_0^i, \ldots, p_{\tau-1}^i$, where $p_j^i$ denotes the position of entity $e_i$ at time $j$ with $0 \leqslant j \leqslant \tau - 1$ (Fig. 7a). We assume that the movement of an entity $e_i$ from its position $p_j^i$ to its position $p_{j+1}^i$ is described by the straight-line segment joining $p_j^i$ and $p_{j+1}^i$. Consequently, the trajectory path $t_i$ is described by the polygonal line (polyline, for short) which may self-intersect and whose vertices are the trajectory points $t_i : p_0^i, \ldots, p_{\tau-1}^i$.

A **sub-trajectory path** $s_i$ of trajectory path $t_i$ is a trajectory path $s_i : q_0^i, \ldots, q_{\xi-1}^i$, such that $j \in [0, \cdots, \tau - \xi]$ with point $q_0^i$ located on edge $p_j^i p_{j+1}^i$ of $t_i$, $q_1^i = p_{j+1}^i, \ldots, q_{\xi-2}^i = p_{j+\xi-2}^i$, and $q_{\xi-1}^i$ located on edge $p_{j+\xi-2}^i p_{j+\xi-1}^i$ of $t_i$ exists (Fig. 7b).

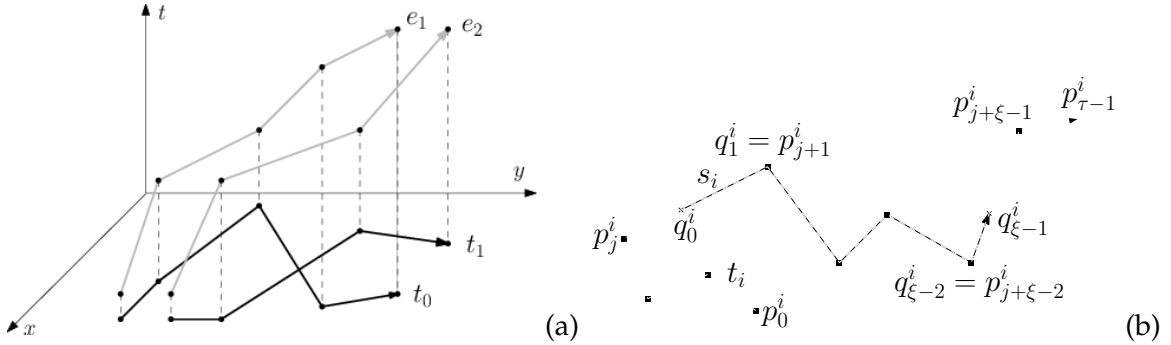



Figure 7: (a) Trajectories of two entities and their corresponding trajectory paths. (b) A trajectory path $t_i$ (dashed polyline) and a possible sub-trajectory path $s_i$ of $t_i$.

In this Chapter, we focus on the problem of detecting popular places among a set $T = \{t_0, \ldots, t_{n-1}\}$ of $n$ trajectory paths in the continuous case. In order to model the proximity between a point $p$ and the trajectory paths, a disk $D(p, r)$ of center $p$ and radius $r$ is used. Moreover, to satisfy the needs of each particular application, we introduce weak and strong criteria for counting the number of

intersections of a trajectory path with the disk $D(p, r)$ (Fig. 8).

**Weak criterion**. The number of intersections between the trajectory path $t$ and the disk $D(p, r)$ is 1 when $t$ intersects the disk, and 0 otherwise.

**Strong criterion**. The number of intersections between a trajectory path $t$ and the disk $D(p, r)$ is the number of maximal sub-trajectory paths contained in $D(p, r)$.

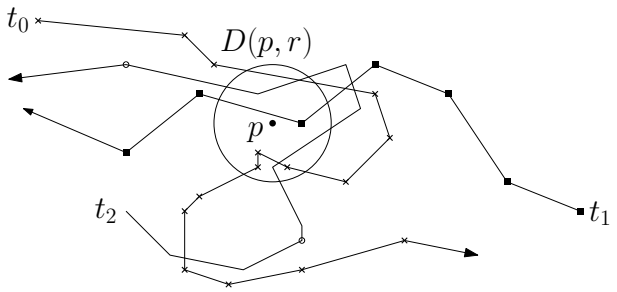



Figure 8: The number of intersections between $D(p, r)$ and the set of trajectory paths is 3 for the weak criterion, and 5 for the strong criterion.

The following definitions are valid for both strong and weak criteria.

Let $T$ be a set of $n$ trajectory paths, $r > 0$ be a real value and $k > 0$ an integer. A point $p$ is an $(r, k)$-**popular point** if the total number of intersections between the disk $D(p, r)$ of center $p$ and radius $r$ and the trajectory paths of $T$, is at least $k$.

An $(r, k)$-**popular region** is a maximal connected set of $(r, k)$-popular points. An $(r, k)$-popular region is bounded by straight-line segments and circular arcs. We denote $\mathcal{P}_{r,k}(T)$ the collection of $(r, k)$-popular regions (see Fig. 9).

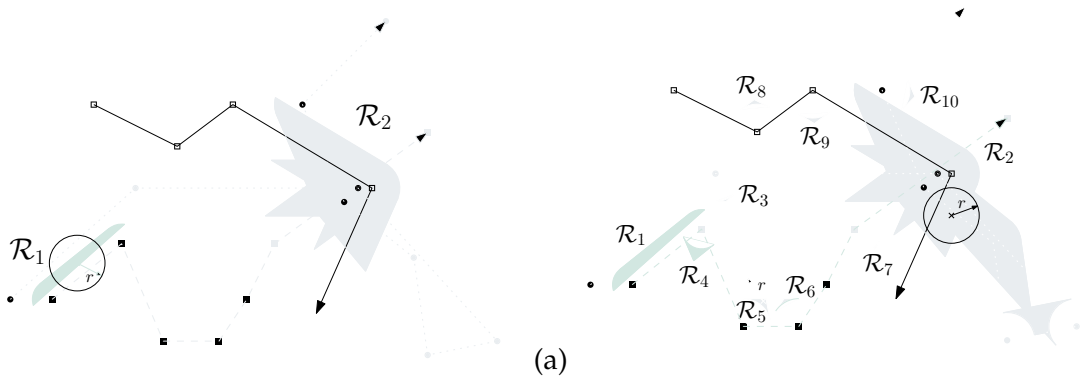



Figure 9: Example with three trajectory paths. (a) when we use the weak criterion and (b) the strong criterion. Note that, points marked with crosses are $(r, 2)$-popular points. The disk in (a) is intersected by two trajectory paths and in (b) by two sub-trajectory paths, at least. Finally, in a) we have $\mathcal{P}_{r,2}(T) = \{\mathcal{R}_1, \mathcal{R}_2\}$ and in b) $\mathcal{P}_{r,2}(T) = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_{10}\}$.

For a given parameter $r > 0$, the **popularity of a point** $p$ is the total number of intersections between the trajectory paths of $T$ and the disk $D(p, r)$ of center $p$ and radius $r$.

The **popularity map**, $\mathcal{M}_r(T)$, is the partition of the plane in maximal connected regions so that all points of a region have the same popularity. Notice that, the set of points of a given popularity may be composed of several independent connected regions. Popularity maps are structures from which it is easy to report popular regions.

### 3.1.1 *Hardness of computation of popular regions*

[19] show that a special case in the problem of computing $P_{r,k}(T)$ in the continuous model, when proximity is determined by a square and belongs to the 3-SUM-hard class [21]. It is not difficult to see that this also holds true, for both weak and strong criteria, when proximity is determined by a disk. Consequently, using the same argument of [19], we can conclude that the best algorithm to compute $P_{r,k}(T)$ in the continuous model when proximity is determined by a disk, for both the weak and the strong criteria, is likely to require $\Omega(\tau^2 n^2)$ time in the worst case, where $\tau n$ is the size of the set $T$ of trajectories. This motivated us to explore an efficient alternative GPU parallel approach for solving the problem.

## 3.2 COMPUTING POPULARITY MAPS

In this section, we explain how to compute the popularity map $\mathcal{M}_r(T)$ from an arrangement of regions related to the trajectory paths in $T$. The regions conforming the arrangement depend on the, weak or strong, counting criterion used.

Finding the popularity map under the weak and strong criterion lead to two different, but similar strategies. Both of them can be adapted to solve both problems; however, each one uses specific properties of the obtained arrangement, making each strategy the most suitable for the considered criterion as is shown in Section 3.6.

### 3.2.1 *Computing popularity maps under the weak criterion*

For each edge $e_j^i = p_j^i p_{j+1}^i$ of the polyline representing trajectory path $t_i$, we consider the offset region $O_r(e_j^i)$ obtained by sweeping along $e_j^i$ with a disk of radius $r$ such that the center moves on $e_j^i$. We can represent the region $O_r(e_j^i)$ as a rectangle and two semidisks, as shown in Fig. 10(a). We denote $O_r(t_i)$ the union of the regions $O_r(e_j^i)$, $0 \leqslant j \leqslant \tau - 1$, determined by the edges of the trajectory path $t_i$ (Fig. 10(b).

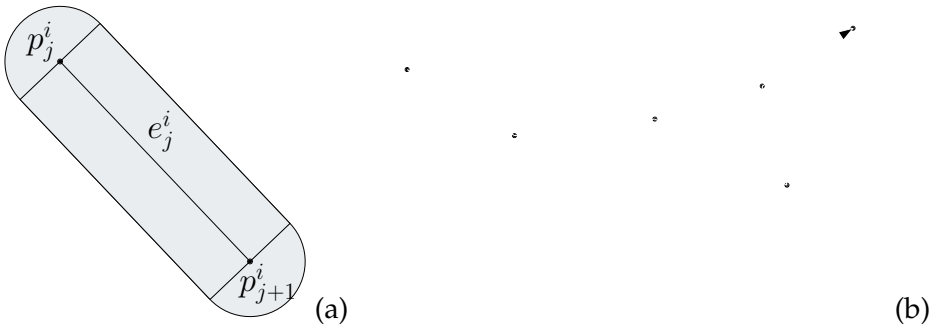



Figure 10: (a) The offset region $O_r(e_j^i)$. (b) The region $O_r(t_i)$.

Observe that, when a point $p \in O_r(t_i)$ then $D(p, r) \cap t_i \neq \emptyset$. Thus, the popularity map $\mathcal{M}_r(T)$ can be obtained from the arrangement of regions $O_r(t_i)$, $0 \leqslant i \leqslant n - 1$, since the number of regions in the arrangement containing a point coincides with the popularity of the point.

To compute $\mathcal{M}_r(T)$ we will discretize the plane into $H \times W$ points, such that each point corresponds to a pixel in the graphics pipeline. The idea is to paint each region $O_r(t_i)$ in the GPU as a set of rectangles and disks. Inside the pipeline, all primitives will be rasterized into fragments

according to the screen resolution $H \times W$. Then, we will store the number of fragments corresponding to each pixel. In this way, we obtain the popularity of each pixel and consequently, a discretization of the popularity map $\mathcal{M}_r(T)$ (Fig. 11).

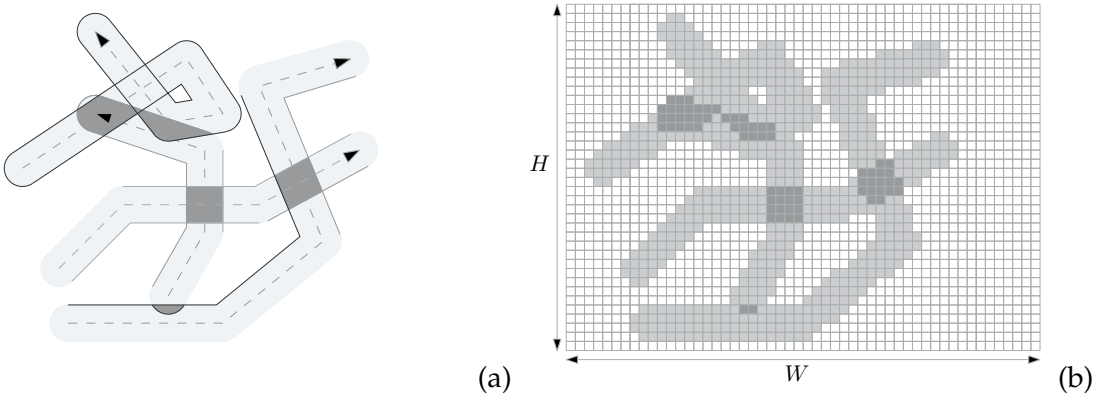



Figure 11: (a) Weak popularity map. (b) Discretization of the weak popularity map.

OpenGL has functions to render points, lines, triangles and rectangles; however, since it does not have a function to render disks, we have to render a disk with Cg using a fragment shader. The disk $D(p_j^i, r)$ is painted as a square centered at $p_j^i$ of side $2r$. Then, a fragment shader is activated so that fragments whose Euclidean distance to $p_j^i$ is bigger than $r$ are discarded.

We use the stencil and depth tests to count the number of fragments corresponding to a pixel and the stencil buffer to store them. Thus, we have to deal with some OpenGL pipeline issues, enable the stencil test and configure it. Note that, two different fragments of the same trajectory path can correspond to the same 2D space that is, to the same pixel, and with the weak criterion they should be counted once only. Using the default configurations of the stencil and depth test, this pixel will be counted twice. This can be solved by avoiding painting the auto-intersection points of the primitives, but this is not just difficult to compute, but also very susceptible to errors. What we do instead, is to paint all the rectangles and disks of $O_r(t_i)$ in a plane parallel to the $XY$-plane at a fixed depth value, and configure the depth test to only pass those fragments whose depth value is strictly smaller than the already stored one. In this way, fragments which coincide with any other fragment of the same trajectory path will have the same depth value, so that only the first fragment will pass the depth test and the following ones will be discarded. This results in pixels covered by region $O_r(t_i)$ being counted once, even if several fragments belong to the same pixel. Note that, the stencil test is configured to increment its stencil pixel value only if the fragment passes the depth test.

Thus, we paint each region $O_r(t_i)$ at a different depth value $z = z_i$ from further ($z = 1$) to closer ($z = 0$) (see Fig. 12). This means that fragments of different trajectory paths will always pass the depth test thus incrementing its stencil pixel value. As a result, the stencil buffer stores, for each pixel, the number of different trajectory paths that are at a distance smaller than $r$ from a given pixel, that is the discretized $\mathcal{M}_r(T)$.

There are some restrictions to the graphics hardware we have to deal with. Stencil and depth buffers are 8 and 32 bits precision buffers, respectively. This means that the buffer can overflow whenever we overtake the range of $[0 \dots 2^8 - 1]$ for the stencil or $[0 \dots 2^{32} - 1]$ for the depth buffer. If we do not take into account these constraints whenever we have a popularity point with popularity bigger than 255 or if we render more than $2^{32}$ trajectories, the buffers will overflow and the algorithm will report incorrect results.

If we work with data sets with thousands of trajectory paths, it won't be unusual to look for regions visited by more than 255 entities. Consequently, we paint from further to closer groups of 255 trajectory paths to ensure that the stencil will never be overflowed. For each painted group, we read the stencil buffer values back to a CPU and the values are accumulated in a CPU array of size $H \times W$. Then, we clean the stencil and depth buffer (all values are set to 0) and we restart

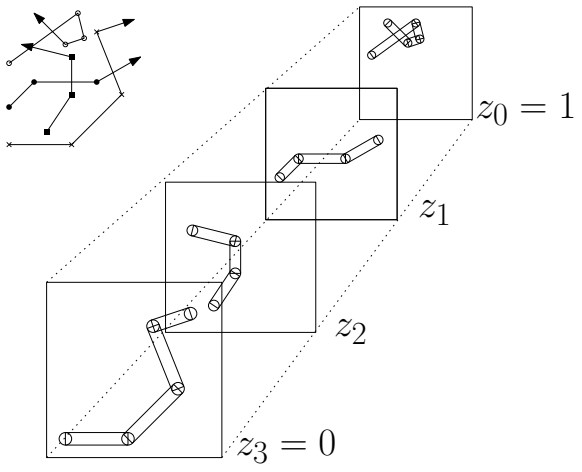



Figure 12: Trajectory paths are rendered with different depth from z=1 to z=0.

the process, until all entities have been painted. Despite the GPU-CPU communication being an expensive operation, we simply have to read from GPU to CPU $n/255$ times, where $n$ is the number of trajectory paths.

Note that, we also clean the depth buffer for every painted group. That is, we clean the depth buffer every 255 trajectories, far from its $2^{32}$ precision constraint. This approach avoids the depth precision buffer issue and removes any constraint on the number of entities in the input data.

*Weak popularity map computation overview*

The summary of the main steps for computing the discretized weak popularity map is:

1. Initialize the depth and stencil buffers to 0 and set the stencil to be incremented if a fragment passes the depth test.
2. Paint each $O_r(t_i)$ region at different depth.
3. For every 255 painted regions, read the stencil values, accumulate them in a CPU array and set the depth and stencil buffers back to 0 again.

#### 3.2.1.1 *Complexity analysis*

Between all the processes that take place in an OpenGL algorithm, we have to take into account the following considerations. 1) the time needed to process a fragment is small compared to the one needed to access a texture or copy information to a texture, and 2) the time needed to read a pixel from GPU to CPU takes much more time than the others. The notation used for the complex analysis is the following:

we denote by $P_f$ the time needed to render and color $f$ fragments, $A_f$ the time spent to make $f$ accesses to a texture, $C_f$ that needed to copy $f$ pixels from texture to texture, and finally, the time needed to transfer $f$ pixels from GPU to CPU is denoted by $\eta$.

We paint up to $2n\tau$ squares of side $2r$ covering the disks centered on the trajectory path vertices. The painted area is $8n\tau r^2$ corresponding to $8n\tau r^2HW$ pixels. In order to paint the rectangles covering the trajectory paths line segments, we paint rectangles of side $2r$. Denoting $L$ the sum of all the trajectory paths length, this represents $2LrHW$ painted pixels. Thus, the total number of painted pixels is $O(P_{(8n\tau r^2+2Lr)HW})$. On the other hand, every 255 trajectory paths we read the stencil buffer back to CPU and so, during the whole process we read back to CPU $HWn/255$ pixels, representing an extra $O(\eta HWn/255)$ time. Consequently the algorithm time complexity is of $O(P_{(8n\tau r^2+2rL)HW} + HWn/255\,\eta)$ and $O(2HW)$ additional space needed to maintain the stencil buffer.

### 3.2.2 *Computing popularity maps under the strong criterion*

When we use the strong criterion to count intersections, we must deal with the edges of a trajectory path, instead of the whole trajectory path. Given a value of $r$, we define $P_r(e^i_j)$ as $P_r(e^i_j) = O_r(e^i_j)$ if $j = 0$ and $P_r(e^i_j) = O_r(e^i_j) \setminus D(p^i_j, r)$, otherwise. These regions can be represented using a rectangle and two semidisks (Fig. 13). Note that, for $1 < j < \tau - 1$, one of the semidisks is subtracted.

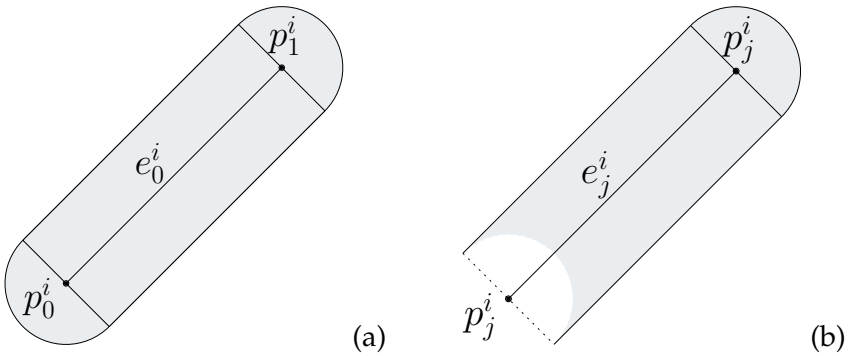



Figure 13: (a) The region $P_r(e^i_j)$ for j=0. (b) The region $P_r(e^i_j)$ for $1 < j < \tau - 1$.

Observe that for a given trajectory path $t_i$, the number of regions $P_r(e^i_j)$, $0 \leqslant j \leqslant \tau - 1$, containing a point $p$ equals the number of maximal sub-trajectory paths of $t_i$ contained in $D(p, r)$ or, in other words, the number of intersections between the disk $D(p, r)$ and the trajectory path $t_i$ under the strong criterion. In consequence, the popularity map $\mathcal{M}_r(T)$ can be obtained from the arrangement of regions $P_r(e^i_j)$, $0 \leqslant j \leqslant \tau - 1$, $0 \leqslant i \leqslant n - 1$, since, again, the number of regions of the arrangement containing a point coincides with the popularity of the point. Notice that, the intersection between the regions $P_r(e^i_j)$ and $P_r(e^i_{j+1})$ determined by two consecutive edges $e^i_j$, $e^i_{j+1}$ of trajectory path $t_i$, defines a region such that a disk centered in any of their points and radius $r$ intersects $t_i$ in two maximal sub-trajectory paths, one contained in $e^i_j$ and the other in $e^i_{j+1}$, and therefore, the number of intersections between $t_i$ and the disk is two (Fig. 14(a)).

To compute $\mathcal{M}_r(T)$, we use the same idea as the one used in the case of the weak criterion. We discretize the plane into $H \times W$ points (one per pixel) and then, we paint the regions $P_r(e^i_j)$ of the arrangement as a set of rectangles and disks (Fig. 14(b)).

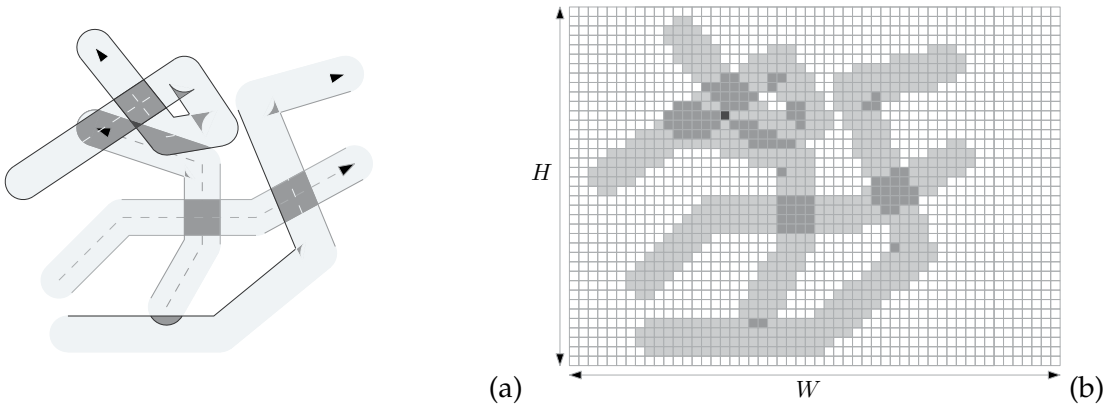



Figure 14: (a) Strong popularity map. (b) Discretized strong popularity map.

We should paint each $P_r(e^i_j)$ with a different depth value so that the stencil value is incremented for each different region. This way, a stencil value will also be incremented when fragments of different time steps coincide, even if they correspond to different time steps of the same trajectory path. The main drawback of this approach is that there is a huge number of data transfers from GPU to CPU. In the weak criterion, to avoid overflows in the stencil buffer, we read back the stencil

values to the CPU every 255 trajectory paths, while in the strong criterion we will have a read back every 255 time steps. Since the number of time steps may be really big, this makes the algorithm useless.

In order to avoid these slow data read backs from GPU to CPU, we propose a new method based on the implementation of a stencil buffer in a fragment shader, that is, modifying the graphics pipeline behavior by using Cg shaders. The idea is to paint each time step with a different depth value and inside the fragment shader increment, for each fragment arriving in the pipeline, the color value of its pixel. When all the time steps have been painted, we can determine how many fragments correspond to each pixel, depending on its final color.

Since we have 3 channels of color (RGB) with 8 bits per channel $[0 \dots 255]$, for each fragment arriving in the fragment shader pipeline, the shader will increment by 1, the color value of the red channel. When the red channel is overflowed (255), the green channel is incremented by 1, and the red channel is restarted at 0. That is, the green channel indicates how many times the red channel has overflowed. The same process is applied to the blue channel when the green channel is overflowed. By recovering RGB final values, we know how many fragments correspond to a given pixel.

The algorithm proceeds as follows. We paint each region $P_r(e_j^i)$ at a different depth value $z = z_j^i$ from further ($z = 1$) to closer ($z = 0$) (Fig. 15). The Cg fragment shader is used throughout the process, not just for adding the color values for all fragments, but also to paint a disk when needed, as explained in Section 3.2. Since we have a depth precision of 32 bits, we can paint up to $2^{32}$ time steps at different depth levels.

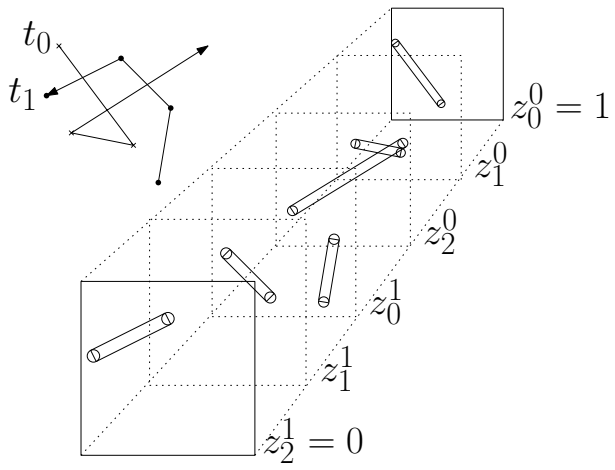



Figure 15: The $P_r(e_j^i)$ regions rendered from z=1 to z=0.

Notice that, each disk corresponding to $p_j^i$ for $1 < i < \tau - 2$ is painted twice, once for each trajectory path segment it defines, and then with different depth values. This causes the color value to be incremented twice where it should be counted only once. In addition, the overlapped regions between rectangles and disks must also be avoided. To solve this problem, we add a parameter to the fragment shader to know whether the disk fragments have to increment the color or only modify the depth value. The fragment shader updates the depth value, but does not increment the color when painting the first disk. When the rectangle and the second disk are painted, both color and depth values, are updated. Proceeding in this way, the overlapped fragments between the two disks along with the rectangle, will be discarded by the depth test and the disks painted twice are counted just once.

When the number of time steps is smaller than $2^{32}$ and the popularity of a point is smaller than $2^{24}$, no read backs to CPU have to be done. This turns the number of read backs from GPU, to zero, in most applications.

The main differences between the two proposed algorithms concerning weak and strong criteria is that the first one needs more readbacks to the CPU, and the second one, in most cases, needs no readbacks, and instead of using the stencil buffer we simulate it with a Cg fragment shader. Despite the stencil buffer being faster than a shader at counting fragments, the relation 'stencil speed'-'read backs' is slower than 'shader speed'-'no readback'. Thus, the strategy used to compute $\mathcal{M}_r(T)$, depends generally on the counting criterion. That is, for the weak criterion it is better to use the stencil counting technique, whereas the strong one is faster to compute using the shader approach.

*Strong popularity map computation overview*

The summary of the main steps for computing the discretized strong popularity map is:

1. Initialize the depth buffer and the three color channels (red, blue, green) of the color buffer to 0.
2. Paint each $P_r(e_j^i)$ region at different depths.
3. Use a fragment shader to accumulate the color in the buffer, depending on whether the fragment passes or not, the depth test.
4. Every $2^{24}$ painted regions, read the color values, accumulate them in a CPU array and reset the depth and color buffers to 0.

#### 3.2.2.1 *Complexity analysis*

The number of painted disks for each trajectory path is $2\tau$, from which, only $\tau$ disks actively access texture values to update the color. They represent $8\tau r^2 nHW$ painted pixels and $\tau 4\pi r^2 HW$ texture accesses. In the case of the rectangles, both, the number of accesses to a texture and pixels painted is $2LrHW$, where $L$ is the sum of all the trajectory paths length. Finally, the information from texture $B$ is copied to texture $A$ a total of $\tau$ times per trajectory path, providing $HW\tau n$ copied values. Thus, by using the notation presented in Section 3.2.1.1, the time complexity is $O(P_{(8\tau r^2 n+2Lr)HW} + A_{(n4\pi r^2\tau+2Lr)HW} + C_{n\tau HW})$ and no extra space is needed.

### 3.2.3 *Error analysis*

The discretzation of the $O_r(t_i)$ regions into pixels, leads to a precision error due to the rasterization process of the GPU pipeline. A pixel partially covered by $O_r(t_i)$ can be painted, or not, depending on the rasterization criterion. According to Segal et al (1994), a pixel is considered part of a region if the geometry of the region covers the center of the pixel. In our case, a pixel will be considered part of a region, and consequently painted, if any $O_r(t_i)$ covers the center of the pixel.

The error occurs when a popular point relies within a not painted pixel, or when a not popular point relies within a painted pixel. In both cases, the maximum error produced is $d$, where $d$ is half of the pixel diagonal. It is not difficult to see that, $d$ increases when the covered area of the input data increases and decreases when the screen resolution increases.

Given a desired error, we can determine the discretization required, according to the covered area dimensions. In the case that the required discretization is not supported by the GPU, the covered area is subdivided into several smaller suitable regions and the algorithms are run independently, in each of them. However, in general, resolutions much smaller than the GPU bounds, which nowadays are $16,384 \times 16,384$, are adequate enough for real applications to guarantee accurate results.

## 3.3 POPULAR REGIONS COMPUTATION

An $(r, k)$-popular region of $\mathcal{P}_{r,k}(T)$, is a maximal connected region composed of regions on the popularity map $\mathcal{M}_r(T)$ whose points have popularity at least $k$.

A discretization of $\mathcal{P}_{r,k}(T)$ can be easily obtained from the discretized $\mathcal{M}_r(T)$. We store the discretization in a 2D array $AP$ of the same dimension, $H \times W$, as the array $AM$ that stores the discretized popularity map. We create $AP$ according to the following criterion: $AP(i,j) = 1$ if $AM(i,j) \geqslant k$ and $AP(i,j) = 0$, otherwise. The resulting array represents a binary image corresponding to the discretization of the collection of $(r,k)$-popular regions.

### 3.3.1 *Popular regions visualization*

The visualization of popular regions provides important reasoning mechanisms to improve the understanding of entities movements, giving to end users images from which they are able to get a quick idea of the situation [17].

We could visualize $\mathcal{P}_{r,k}(T)$ as a binary image where those pixels whose $\mathcal{P}_{r,k}(T)$ value is smaller than $k$ are painted black, while the others are painted white. Alternatively, in order to obtain better visual information, we visualize $\mathcal{P}_{r,k}(T)$ from the discretized $\mathcal{M}_r(T)$, where pixels of $\mathcal{M}_r(T)$ with values smaller than $k$ are painted white, and the remaining pixels are painted according to their popularity. Then, instead of having just two colors, we uniformly distribute the popularity range values among the whole RGB range (red, green and blue). In particular, when $k = 1$ we obtain the visualization of the popularity map $\mathcal{M}_r(T)$.

Figures 5 and 16 show examples of where a gradient of three colors is used. Regions of small popularity values are painted in yellow, middle values in blue, and the most popular ones, in red. Note that, as expected, differences between using the weak or strong criteria are obtained. Consequently, before starting to solve a specific problem in a realistic situation, we have to take care to choose the appropriate criterion according to the different information they report and to the problem we are dealing with.

In the example of Fig. 5, a group of ants look for food on a region. When an ant finds food, it takes a piece of food and transports it to the anthill, leaving a trail which is followed by the rest of the ants. Ant movement between the food and the anthill can be appreciated in both Figures 5(b) and 5(c). When ants do not find food, they move randomly over the region looking for any place with food. No ant can return to the anthill without food. Note that, when using the weak criterion, the most popular place is the anthill, which is located in the center. This is reasonable, because at the beginning every ant has visited the anthill at least once, on their way out to look for food. So, any other region could be, at most, as popular as the anthill. By contrast, using the strong criterion, we appreciate that the most popular region is not the anthill, despite having been visited at least once and sometimes twice or more by those ants that have found food. This leads to the conclusion that ants have spent more time looking for food, rather than, finding food.

Another example is shown in Fig. 16. It represents a group of people moving in a city with perpendicular streets. Imagine that we want to know the most suitable location for a pollster. Obviously we want him to poll as many people as possible, so we need to know the busiest area. The information provided from the $\mathcal{P}_{r,k}(T)$ visualization using the weak criterion (Fig. 16(a)), shows that the most popular places are the two street intersections in the upper-left corner. On the other hand, the information reported when we use the strong criterion (Fig. 16(b)), shows that the most popular region is the second street starting from the right. The street intersections marked as more popular when we use the weak criterion, are the places visited by many different people. However, while the most popular street with the strong criterion is a very busy street, it may be being visited by the same people many times. Since a poll is taken only once per person, we are interested in the popular regions reported when the weak criterion is used.

## 3.4 POPULAR REGION SCHEMATIZATION

Sometimes, it may be interesting to obtain a schematization of the popular regions by simplifying their geometry, in order to reduce complexity and to focus on relevant information.

We will use the skeleton of the discretized $\mathcal{P}_{r,k}(T)$ to schematize popular regions.

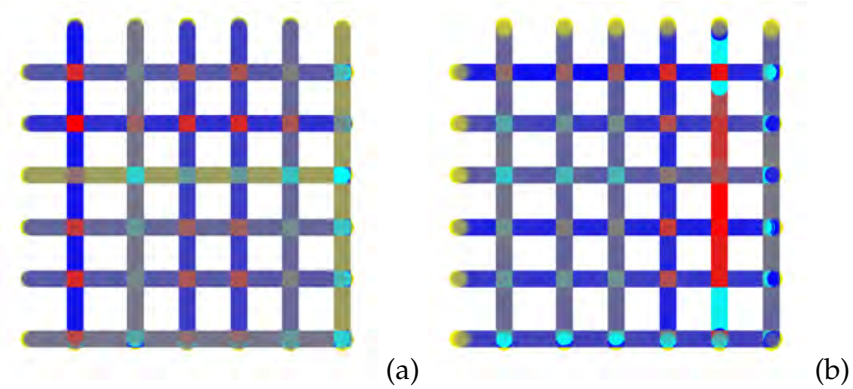



Figure 16: Visualization of $\mathcal{P}_{r,k}(T)$ under (a) weak and (b) strong criterion, respectively.

### 3.4.1 *The skeleton of a multi-object binary image*

Let $\Omega$ be a multi-object binary image in a 2D discrete space $\mathcal{D}$ and $\delta\Omega$ its boundary. The feature transform assigns to each pixel $p \in \mathcal{D}$ the set $F(p) = \arg\min_{q \in \delta\Omega}\{d(p,q)\}$ of its nearest boundary pixels under the Euclidean distance. A simple feature transform computes only one nearest feature pixel $S(p)$ per pixel $p$, which is sufficient for many applications.

The skeleton or medial axis [20], is defined as the locus of pixels at equal distance from at least two boundary pixels:

$$S(\delta\Omega) = \{p \in \mathcal{D} \mid \exists\, q, r \in \delta\Omega, q \neq r : d(p,q) = d(p,r)\}.$$

Given a pixel $p = (i,j) \in \mathcal{D}$, we will denote $S_{i,j} = S(p)$. Once, $\{S_{i,j} \mid \forall (i,j) \in \mathcal{D}\}$ is computed, we can quickly generate an approximated skeleton as:

$$\widetilde{S}(\delta\Omega) = \{(i,j) \in \mathcal{D} \mid max\{d^2(S_{i+1,j}, S_{i,j}), d^2(S_{i,j+1}, S_{i,j})\} > \lambda\}$$

where $\lambda$ is a given fixed parameter that determines when a transition of the nearest feature pixel between adjacent pixels is big enough to be considered an skeleton pixel.

### 3.4.2 *Computing the skeleton*

In order to obtain an approximated skeleton $\widetilde{S}(\delta\Omega)$ of the discretized $\mathcal{P}_{r,k}(T)$, the collection of $(r,k)$-popular regions, we: 1) extract the contour of $\mathcal{P}_{r,k}(T)$; 2) compute the simple feature transform $S(\delta\Omega)$ of the extracted contour; and 3) generate the approximated skeleton from $S(\delta\Omega)$. We implement these three steps inside the GPU by using OpenCL over CUDA architecture, without any readback to the CPU except for the final result $\widetilde{S}(\delta\Omega)$.

*Extracting the contour*

Many methods have been proposed in order to extract the contour of a binary image [22]. A well-known approach for detecting contours is image filtering. There are many possible filters and each one is determined to be appropriated, or not, depending on the features of the entry image. Since our input image, $\mathcal{P}_{r,k}(T)$ is a base case of contour detection (totally binary and noise free), we will apply a derivative filter. For each pixel, the $(1,-1)$ filter is applied on $x$ and $y$ so that the transition between regions of 0's to regions of 1's is detected. Note that, we are shifting each contour pixel a half pixel right and a half pixel down since the real transition from 0 to 1 comes between the two pixels.

We compute the contour with an OpenCL kernel by assigning a thread to each pixel. Then, using the derivative filter, we determine whether each pixel is a boundary point. In this way, we get a binary image that we then store in a 2D array $AC$ of dimension $H \times W$, so that $AC(i,j) = 1$ if

$(i, j)$ is a contour pixel and $AC(i, j) = 0$, otherwise.

*Computing the simple feature transform*

The computation of the simple feature transform $S(\delta\Omega)$ is an expensive operation, because finding the nearest contour pixel of each one of the $H \times W$ pixels of the discretized plane is required. However, since the proximity of one pixel to the contour pixels is independent of the others, we compute the single feature transform by using OpenCL kernels which simultaneously compute all nearest contour pixels for each pixel.

To efficiently compute $S(\delta\Omega)$, we make use of the ideas described in [23], to solve nearest neighbor queries. First, a uniform grid over the set of contour pixels (that can be obtained from array $AC$) is built, and next, the grid is used to simultaneously search for the nearest contour pixel of the $H \times W$ pixels on the discretized plane. Both parts, the grid construction and the nearest neighbor obtained, are done in parallel using OpenCL. We store the simple feature transform obtained, in a 2D array $AS$ of dimension $H \times W$.

*Generating the approximated skeleton*

The approximated skeleton $\widetilde{S}(\delta\Omega)$ is obtained as a distance calculation between the simple feature transform of adjacent pixels thresholded by parameter $\lambda$. We store the skeleton in a 2D array $AK$ of dimension $H \times W$, so that $AK(i, j) = 1$ if $\max\{d^2(AS(i + 1, j), AS(i, j)), d^2(AS(i, j + 1), AS(i, j))\} > \lambda$ and $AK(i, j) = 0$, otherwise. Again, this operation can be done using an OpenCL kernel where each thread computes a pixel value. Then, the result is read back from GPU to $AK$.

Note that, the three steps to compute the skeleton are executed entirely inside the GPU. We write $\mathcal{P}_{r,k}(T)$ in GPU memory in the first step, and we read back $AK$ to the CPU in the final step. This is a high priority practice for any GPU algorithm.

#### 3.4.2.1 *Complexity analysis*

The complexity analysis of algorithms that use CUDA not only give the total work of an algorithm, which represents the total number of operations done in order to solve the problem, but also give the step complexity, which is the time that the algorithm would take if we had an infinite number of parallel processors (threads). We also reflect the number of accesses to memory and the amount of information transferred. The time needed to read $f$ and write $g$ values from global memory is represented by $G_{f+\rho g}$, when shared memory is used it is denoted by $S_{f+\rho g}$, and finally, $T_{f+g\rho g}$ represents $f$ transferred values from CPU to GPU and $g$ transferred values from GPU to CPU.

The time complexity of this part of the algorithm is the sum of computing the derivative filter, finding the simple feature transform and finally, obtaining the thresholded distance.

Computing the derivative filter for the $HW$ pixels has a complexity of $O(HW + T_{HW} + G_{4HW+HW\rho})$. For the simple feature transform by using a grid to find the nearest neighbor, we denote $S_f$, $S_{ft}$, $S_{fw}$ and $S_{fr}$ the total work, the number of transferred values to the GPU and the written and read values from global memory, respectively. Thus, the second step complexity is $O(S_f + T_{S_{ft}} + G_{S_{fr}+S_{fw}\rho})$. Finally, obtaining $\widetilde{S}(\delta\Omega)$ yields a $O(HW + G_{4HW+HW\rho})$ complexity. Summing them all up, the algorithm that computes the skeleton has complexity $O(HW + S_f + T_{HW+S_{ft}} + G_{HW+S_{fr}+(HW+S_{fw})\rho})$.

### 3.4.3 *Popular region schematization visualization*

The skeleton, used to schematize the popular regions, can be visualized using the reported array $AK$, which is a binary image. We initialize the stencil buffer with $AK$ values and the color buffer with white. We enable the stencil test and set it to pass only those fragments whose stencil value is equal to 1. Next, we paint a colored rectangle covering the whole screen, only the pixels representing the skeleton of $\mathcal{P}_{r,k}(T)$ pass the stencil test and are colored (see Fig. 18(c) and 17(c)).

The skeleton is specially useful in aiding the understanding of the information contained in popular regions. In this case, we can differentiate between two different scenarios depending on whether the movement of entities is subject to some constraints.

#### 3.4.3.1 *Not constrained movement*

The skeleton can be used when entities have no movement constraints, thus providing a schematization of the most popular region. In Fig. 17, again we can see the ant trajectory paths, their weak popularity map and the corresponding skeleton. Notice that, in this case the skeleton provides schematic and interesting information compared to the set of trajectory paths.

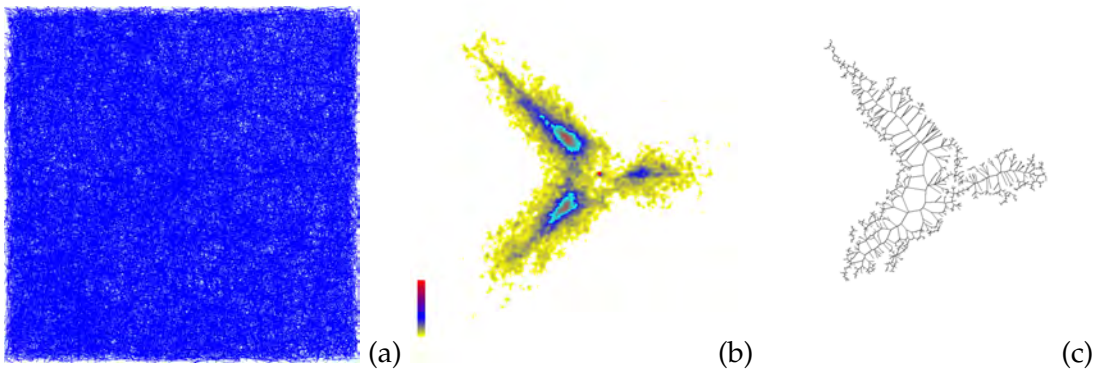



Figure 17: (a) A set of ants trajectory paths, (b) their weak popularity map and (c) the corresponding skeleton.

#### 3.4.3.2 *Constrained movement*

When the movement of entities is subject to some constraints, as in the road network case, the skeleton provides a good schematization of the popular regions. Fig. 18 shows the trajectory paths of a real set of cars moving on a road network, their popularity map and their skeleton. For the specific case of trajectory paths on road networks, whenever the sampling rate of the input data is dense enough to allow the trajectory paths follow the roads, the skeleton tends to match with the most popular roads.

Notice that, in the special case of trajectories on road networks, the problem we solve is completely different to that of the previously presented work, which includes: a) hot routes, where general paths followed by multiple moving objects are detected [24, 25]; b) trajectory patterns, providing a sequence of spatial regions frequently visited in the order specified by the sequence [26]; and c) interesting places in trajectories allowing the *stops* and *moves* semantic analysis [27]. The problem we solve identifies popular places, which are places that have been visited many times, however, we do not differentiate unreported movements in opposing directions, nor do we maintain the order in which the places are visited. Popular places can be used, for instance, to determine which parts of the city are noisier or more polluted.

## 3.5 ADDITIONAL CONSTRAINTS

To improve the understanding of trajectory path data, our algorithms handle constraints on both, the input trajectory paths to be analyzed and the extracted popular places.

As an input constraint, we consider the time aspect of the initial trajectories by assigning specific time slots to the analysis of the popular places. The user can select a time interval (Fig. 19), and the corresponding popular places are computed by considering only the sub trajectories contained within the given time interval, with their associated sub trajectory paths. By considering different

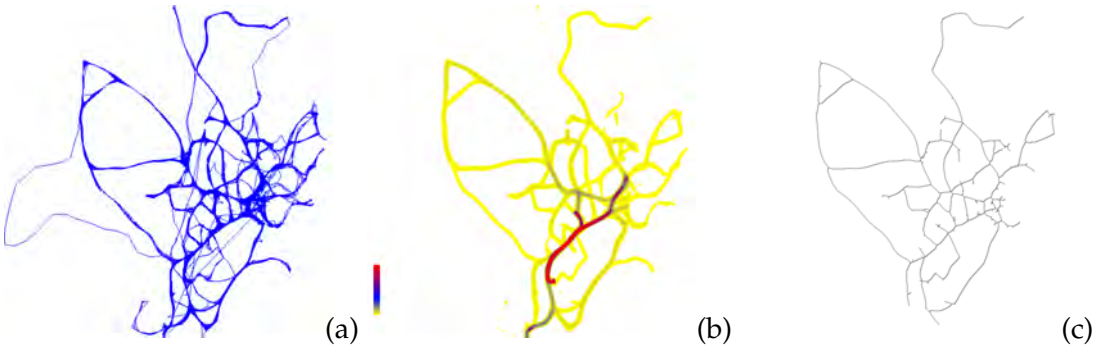


Figure 18: (a) A set of real lorry trajectory paths, (b) their weak popularity map and (c) the corresponding skeleton.

time slots, we are able to determine how popular places change with time. Notice that, this additional constraint is the only moment when the trajectories time aspect is collaterally taken into account, because the problem tackled here does not consider the trajectories time aspect.

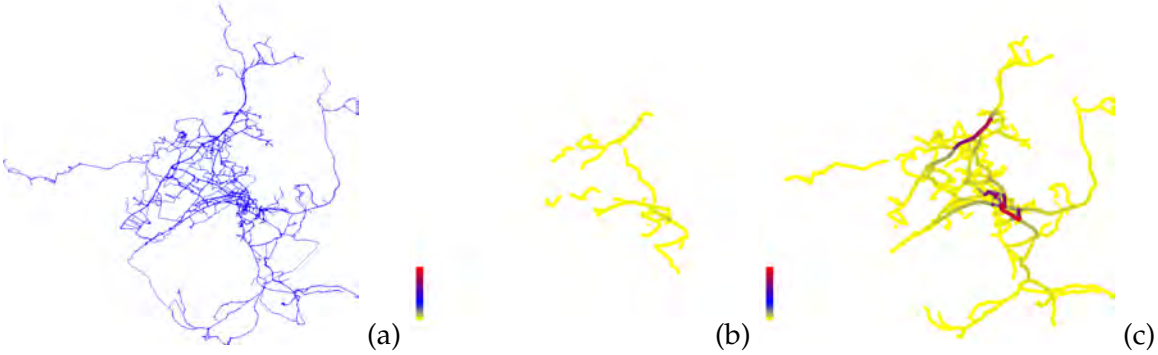


Figure 19: (a) A real data set of trajectory paths tracking Athens school buses (b) the weak criterion popularity map between 0:00h and 6:00h and (c) between 8:00h and 15:00h.

As an output constraint, given that popular places can be excessively small, the user can require a minimum area/length for the popular regions/paths to be visualized (Fig. 20). This is specially useful when we use the strong criterion where, as shown in Fig. 9b, some reported areas/paths are very small/short because of the small angle between trajectory path edges corresponding to two consecutive time steps.

## 3.6 EXPERIMENTAL RESULTS

The aim of this section is to test our algorithms from a running time efficiency point of view. We have implemented an application to change the parameters of our problems easily, and to check the output. Tests have been computed in an Intel Core 2 CPU 2.13GHz, 2GB RAM and a GPU NVidia GeForce GTX 480.

The methods presented for computing and visualizing popular places have been tested under the following datasets.

- **Buses and Trucks:** A set of school buses and trucks moving in the Athens metropolitan area extracted from [28]. The data have been obtained with the WGS84 reference system for latitud and longitud.
- **Cars:** Real trajectory paths tracked with a GPS placed in a set of cars moving around the area of Girona, Spain.

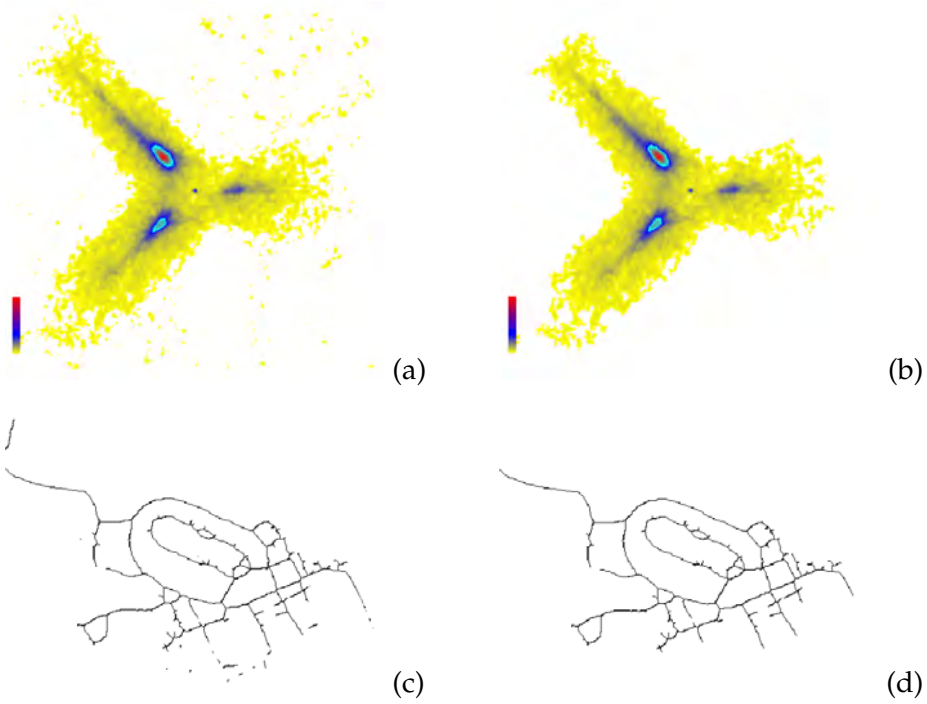



Figure 20: (a) Popular regions for the ant data set. (b) The resulting output after applying the area constraint. (c) Visualization of $\mathcal{P}_{r,k}(T)$ skeleton for the 'State fair' data set and (d) the resulting output after applying the length constraint.

- **State Fair:** A daily GPS track collected from human movement in the NC State Fair held in North Carolina [29]. Positions are stored as the distance to a reference system located in the surrounding area.
- **Traffic and animal simulations:** Two synthetic datasets generated with Netlogo [18]. The Traffic simulation is a basic simulation of entities moving along streets where each entity moves one block per time step and at each crossroad a new random direction is chosen. Animal simulation simulates animals moving on a terrain with no movement restrictions.
- **Soccer simulation:** A synthetic data set containing the simulation of soccer players during a match. The positions are stored as the distance to an owned reference system located in the middle of the soccer field.

In Tab. 1, detailed information of the real data sets are presented. The number of entities, the total number of time steps, the sampling rate, the area covered and the error produced for different

Table 1: Real data sets details information

| Data set | $n$ | $\tau$ | Sam. Rate (s) | Area($m^2$) | Error($m$) | |
|---|---|---|---|---|---|---|
| | | | | | $512 \times 512$ | $1024 \times 1024$ |
| Busses | 145 | 66,096 | 30 | $25 \times 10^6$ | 6.90 | 3.45 |
| Trucks | 276 | 112,203 | 10 | $400 \times 10^6$ | 27.62 | 13.81 |
| Cars | 340 | 87,031 | 30 | $18.2 \times 10^6$ | 12.56 | 6.28 |
| State Fair | 2 | 5,861 | 30 | $2.29 \times 10^6$ | 1.66 | 0.83 |

Table 2: Computation times reported, when applying the methods presented to compute $\mathcal{M}_r(T)$ under the different counting criteria.

| | | $\mathcal{M}_r(T)$ *Computation time (s)* | | | |
|---|---|---|---|---|---|
| | | *Weak* | | *Strong* | |
| T | r | Stencil | Shader | Shader | Stencil |
| State fair | 2 | 0.006 | 0.011 | 0.441 | 0.731 |
| | 5 | 0.006 | 0.011 | 0.439 | 0.721 |
| | 10 | 0.006 | 0.010 | 0.439 | 0.721 |
| Animal Sim. | 2 | 0.201 | 0.562 | 12.850 | 22.888 |
| | 5 | 0.201 | 0.561 | 12.852 | 22.889 |
| | 10 | 0.202 | 0.563 | 12.850 | 22.902 |

screen resolutions are shown for each real data set. Note that 'Traffic and animal simulation' and 'Soccer simulation' data sets are not present in the table because they are synthetic data sets and it makes no sense to report the sampling rate or the covered area. The purpose of these simulations is to test the algorithm with data sets containing a large number of entities and time steps, due the difficulty in finding data sets which are real and large. Traffic and animal simulations have $1,000$ and $10,000$ entities respectively, with a total of $30,000$ time steps for the traffic simulation and $200,000$ for the animal simulation. In the 'Soccer simulation' data set we simulate soccer players during a match obtaining a total of $4.14 \times 10^6$ time steps.

The times presented are total running times, including the transferring times from and to the GPU, the initialization times, etc. Each running time reported, has been computed as the average of 10 executions of the same parameter values. We can not compare our execution times against others because, to the best of our knowledge, no other implementations exist. Benkert et al. presented in [19], an approach to report popular places, but their proposal is from a theoretical point of view. No running times are reported and, as far as we know, no other papers about this particular pattern have been published.

In Section 3.2, we have presented two approaches for computing $\mathcal{M}_r(T)$, depending on the counting criterion used. Table 2 shows the total running times needed to compute $\mathcal{M}_r(T)$ with the algorithms using each approach for both criteria. 'Stencil', refers to the algorithm presented for the weak criterion, which uses the stencil buffer to count the fragments, and 'Shader', refers to that presented for the strong criterion, which uses a shader simulating the stencil. We can see that, as stated in Section 3.2, 'Stencil' is the best option for the weak criterion and 'Shader' for the strong criterion.

Figures 21 and 22 show computational and visualization running times, for different $k$ and $r$ values. Fig. 21 provides the running times when using a screen resolution of $512 \times 512$ and Fig. 22, for a $1024 \times 1024$ screen resolution. Even though the biggest discretization size supported by the GPU is $16384 \times 16384$, we have chosen these resolutions because, as we see in Table 1, the error produced is acceptable.

In both, Fig. 21 and Fig. 22, (a) and (b) present the time in seconds needed to compute $\mathcal{M}_r(T)$ and a specific $\mathcal{P}_{r,k}(T)$ for the weak and strong criterion, respectively. Note that, the running times are fairly affected by the different $r$ and $k$ input values, showing a good scalability in these parameters. We have provided the total time to compute $\mathcal{M}_r(T)$ and extract one $\mathcal{P}_{r,k}(T)$, however, for a fixed

$r$ and a given data set, obtaining $\mathcal{P}_{r,k}(T)$ from $\mathcal{M}_r(T)$ is really fast. For instance, for the 'Animals Sim.' dataset, once the $\mathcal{M}_5(T)$ is computed, it takes 8.81, 9.32 and 9.11 miliseconds to compute the $\mathcal{P}_{r,k}(T)$ for $k = 2$, $k = 5$ and $k = 10$, respectively. Comparing both criteria, we can see how, as expected, the computational times are bigger under the strong criterion, especially when the number of time steps increases. For instance, in the animal simulation dataset, the computational times are up to six times faster under the weak rather than the strong criterion. That is because the algorithm directly depends on the number of painted fragments so, the more time steps, the slower the strong criterion computation. We also tested the strong criterion algorithm with the soccer data set. With a $1024 \times 1024$ screen resolution, it took 302.78 seconds to compute the $\mathcal{M}_r(T)$. These experiments show the scalability of our algorithm, which has no limits in the input data. As for the number of time steps, the sets provided range from $5,861$ to $4.15 \times 10^6$ time steps.

The times in seconds needed to compute the skeleton from an already computed $\mathcal{P}_{r,k}(T)$, are shown in Fig. 21 and Fig. 22 (c) and (d). The skeleton computation is not affected by the input parameters, nor by the number of entities. In fact, times differ without showing any correlation between $r$, $k$, the number of entities or time steps. This is because it depends on the distribution of the boundary points of $\mathcal{P}_{r,k}(T)$, which is mainly related to trajectory path distribution. When comparing images (c) and (d) in Fig. 21 and 22, we can see that the only affected times, when increasing the screen resolution, are those of the animal and traffic simulation, especially with $r = 50$ and $k = 5$.

Finally, in Fig. 21 and Fig. 22 (e) and (f), $\mathcal{P}_{r,k}(T)$ and skeleton visualization times in miliseconds are provided, respectively. The visualization of $\mathcal{P}_{r,k}(T)$ depends on the range of popularity values obtained and on screen resolution. However, the visualization times are really small.

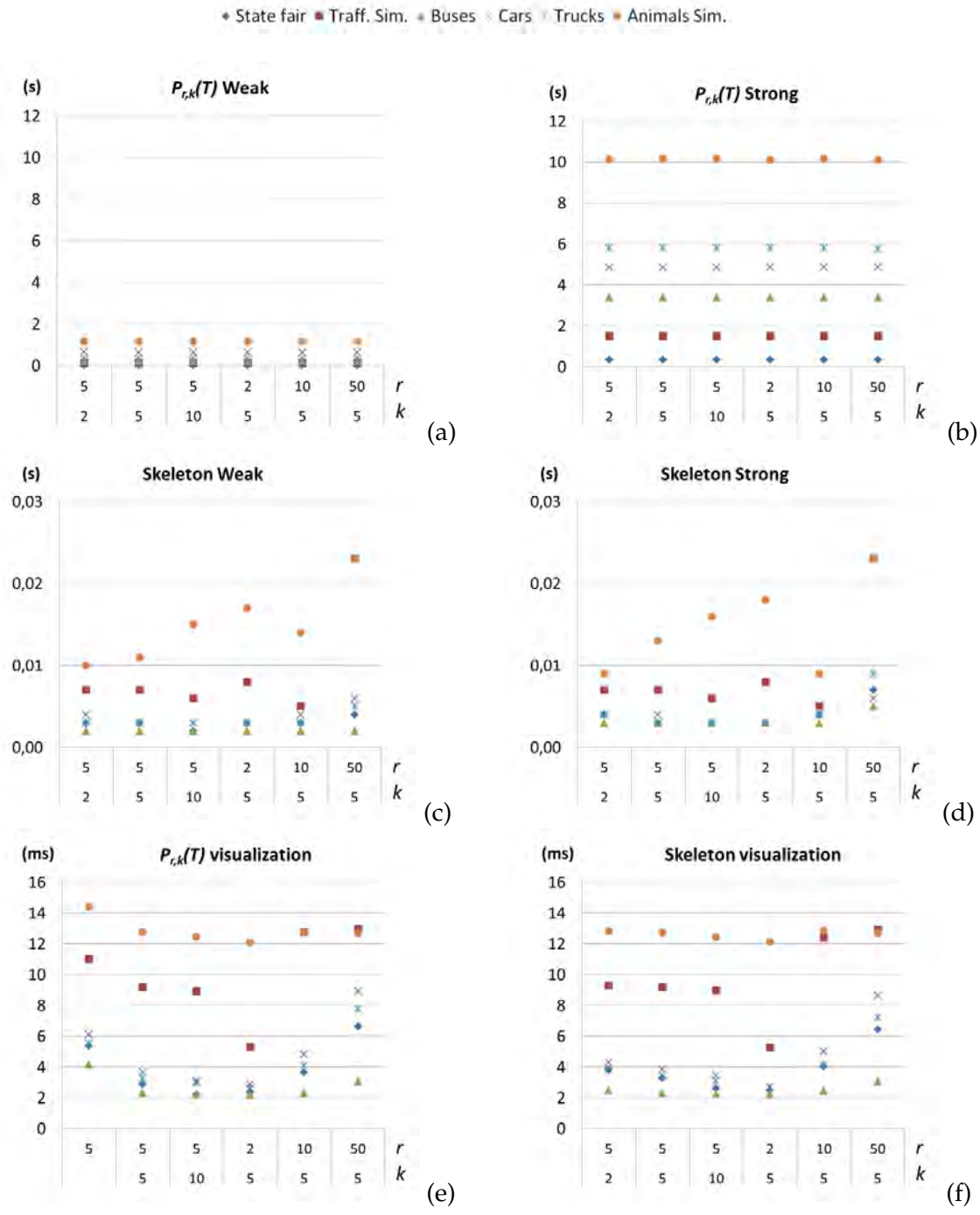



Figure 21: Running times for a $512 \times 512$ screen resolution

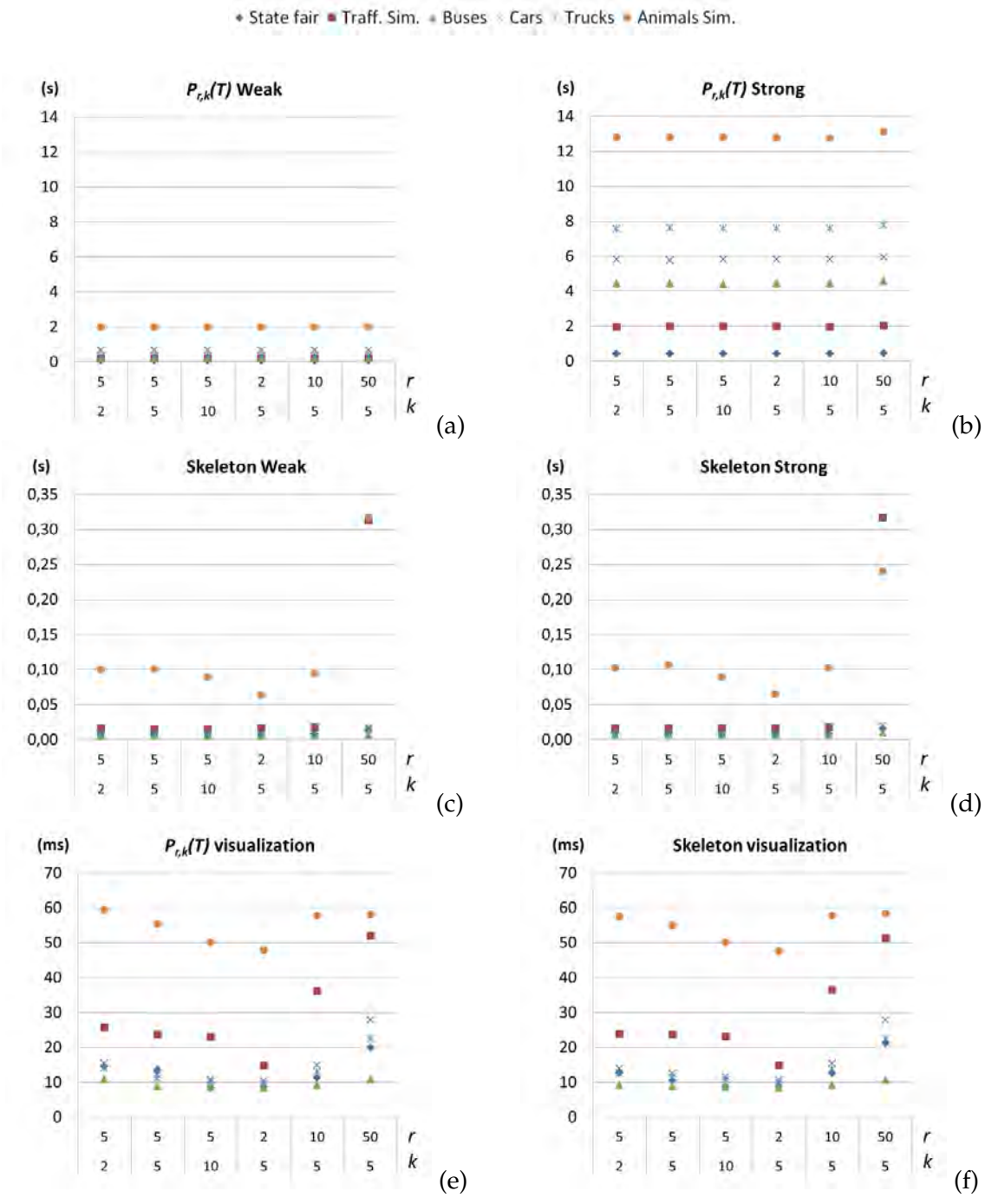



Figure 22: Running times for a $1024 \times 1024$ screen resolution

# 4

# SUBTRAJECTORY CLUSTERING

*It's better to wait for a productive programmer to become available than it is to wait for the first available programmer to become productive.*
***– Steve McConnell***

In this Chapter we focus on the problem of detecting subtrajectory clusters in a trajectory. That is, we search for similar subtrajectories along a single trajectory, as is illustrated in Fig. 23a. Depending on the application a subtrajectory cluster (SC) can have a different meaning. For example, for a person a SC might indicate a daily commuting behavior between home and work. When tracking birds a SC might indicate annual migration behavior or regular movement between the nest and foraging areas. A final example, which we will focus our attention on in this Chapter, is football where a subtrajectory cluster indicates frequent movements on the field performed by a player, see Fig. 23b for an example.

More formally, following the definition from [30], the input is a moving point object in d-dimensional Euclidean space, called *entity*, whose location is known at $n$ consecutive time-steps. We assume that an entity moves between two consecutive time steps on a straight line. Given the trajectory $T$ of a moving entity, an integer $m > 0$ and two positive real values $\ell$ and $\varepsilon$, we define a subtrajectory cluster $C$ as a set of at least $m$ subtrajectories of $T$, where the subtrajectories are within distance $\varepsilon$ from each other, at least one subtrajectory has length $\ell$ and the time intervals of two subtrajectories in $C$ overlap in at most one point.

To measure spatial similarity we choose the Fréchet distance between (sub)trajectories. In this Chapter we only consider the trajectory as a directed curve in 2D, thus invariant under differences in speed: for instance, in a transportation context, this allows us to detect a commuting pattern even in the presence of different traffic conditions and varying means of transport. However, speed can be taken into account by considering time to be the third dimension. The algorithm presented in this Chapter works for any dimension, so does the Fréchet metric, but for the application we consider the variance in speed is of minor importance.

The problem was considered in [30] in which the authors also developed approximation algorithms using both the discrete Fréchet distance and the continuous Fréchet distance. Their algorithms has a running time of $O(\ell n^2)$ and $O(n^3 m \cdot 2^{\alpha(n/m)}(\log n \log(n/m) + m))$ when using the discrete and continuous Fréchet distance (to be defined), respectively. In the case when at least one of the subtrajectories in a cluster has to start and end at vertices of the trajectory the algorithm requires $O(n^2\ell)$ time and $O(n\ell^2)$ space, which is also the version we will focus on in this Chapter The algorithm in [30] can be modified to return many different outputs depending on the settings, however, in general it reports a set of subtrajectory clusters where each cluster $C$ in the set contains at least $m$ subtrajectories of $T$, at least one of the subtrajectories has length $\ell$ and the Fréchet distance between any two subtrajectories in $C$ is at most $2\varepsilon$. Due to this the algorithm is said to be a 2-distance approximation algorithm.

For the discrete Fréchet distance only distances between vertices are computed. The discrete Fréchet distance works well in the case when the input data is very dense. In these cases there is only a small difference between the discrete and continuous Fréchet distance. However, in most cases the data is not dense, and even in the case when the input data is dense one often wants to compress the data by using a simplification algorithm [31, 32]. In most realistic settings the input data can be compressed to $5-10\%$ of its original size, thus developing a practical approach for trajectory clustering using the continuous Fréchet distance is a crucial open problem. Note that due

to the high complexity only the algorithm using the discrete Fréchet distance was implemented and tested in [30]. See also [33].

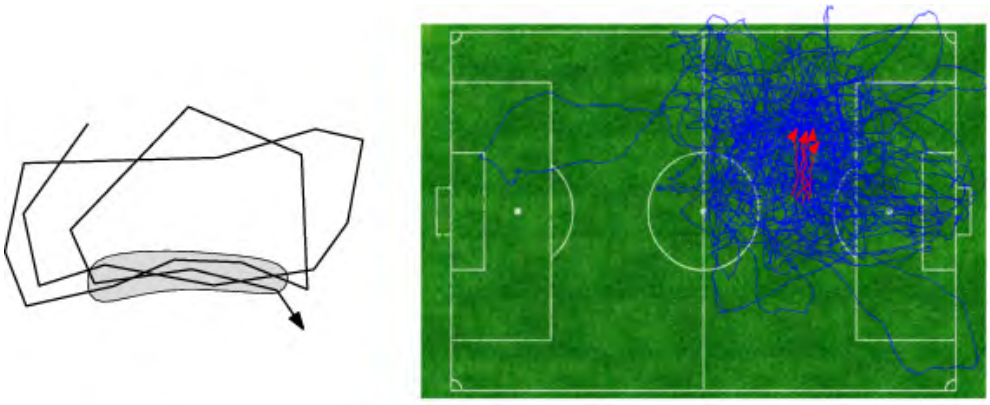



Figure 23: (a) Illustrating a subtrajectory cluster. (b) An example of a subtrajectory cluster reported from a trajectory of a football player.

## 4.1 PRELIMINARIES: THE FRÉCHET DISTANCE

The Fréchet distance can be illustrated as follows: suppose a man is walking his dog and that he is constrained to walk on a curve $f$ and his dog on another curve $g$. Both the man and the dog are allowed to control their speed independently, but are not allowed to go backwards. Then, the Fréchet distance between $f$ and $g$ is the shortest leash that enables both the person and the dog to travel along $f$ and $g$ with a leash connecting them.

For completeness, we include the formal definition:

**Definition 1.** *Let* $f : I = [l_I, r_I] \to \mathbb{R}^d$ *and* $g : J = [l_J, r_J] \to \mathbb{R}^d$ *be two curves, and let* $|\cdot|$ *denote the Euclidean norm*[1] *in* $\mathbb{R}^d$. *Then the Fréchet distance* $\delta_F(f, g)$ *is defined as*

$$\delta_F(f, g) = \inf_{\substack{\alpha:[0,1]\to I \\ \beta:[0,1]\to J}} \max_{t\in[0,1]} |f(\alpha(t)) - g(\beta(t))|,$$

*where* $\alpha$ *and* $\beta$ *range over continuous and increasing functions with* $\alpha(0) = l_I$, $\alpha(1) = r_I$, $\beta(0) = l_J$ *and* $\beta(1) = r_J$.

Playing around with this metric, one realizes that because of the continuity requirement on the mapping $f$, this measure is in most cases much better suited than the classical Hausdorff distance between the curves [35], or many other proposed metrics (the Longest Common Subsequence model [36], a combination of parallel distance, perpendicular distance and angle distance [37], the average Euclidean distances between paths [38] and so on [39]). Unfortunately, in the continuous setting, computing the Fréchet distance between two curves requires quadratic time in the complexity of the curves [34]. It is currently an open question if it is possible to do better than quadratic even when one tries to approximate this quantity within a constant multiplicative factor. Alt [40] conjectured that the decision problem may be 3SUM-hard (i.e. a subquadratic time algorithm is unlikely to exist). The only subquadratic algorithms known are for restricted classes of curves such as for closed convex curves and when the input has low density (k-packed, k-straight, k-bounded) [41]. Very recently Agarwal et al. [42] showed that the discrete Fréchet distance can be computed in subquadratic time, namely in $O(n^2 \frac{\log\log n}{\log n})$ time.

1 Other norms are possible as well, see Alt and Godau [34].

### 4.1.1 *Related results*

Vlachos et al. [36] state that in the area of spatiotemporal analysis it is important to detect commuting patterns of a single entity, or to find objects that move in a similar way. An efficient clustering algorithm for trajectories, or subtrajectories, is essential for such analysis tasks. Gaffney et al. [43, 44] proposed a model-based clustering algorithm for trajectories. In this algorithm, a set of trajectories is represented using a regression mixture model. Specifically, their algorithm is used to determine cluster memberships and it only clusters whole trajectories. Lee et al. [37] argued that clustering trajectories as a whole would not detect similar portions of the trajectories. Instead they suggested an approach that partitions a trajectory into a set of line segments and then group similar segments. The obvious drawback is that only segments are clustered, while a commuting pattern might be a complicated path whose shape may not be captured by a single segment. Mamoulis et al. [45] used a different approach to detect periodic patterns. They assumed that the trajectories are given as a sequence of spatial regions, for example ABC would denote that the entity started at region A and then moved to region C via region B. Using this model data mining tools such as association rule mining can be used. Vlachos et al. [36] also looked at discovering similar trajectories of moving objects. They mainly focused on formalizing a similarity function based on the longest common subsequence which they claim is very robust to noise. However, their approach matches the vertices along the trajectories which requires that the vertices along the trajectories are synchronized, or almost synchronized, which is rarely the case. Also, if the trajectories are simplified (compressed) in a preprocessing step then the data will not be synchronized.

## 4.2 PROBLEM SETTING

As mentioned in earlier work [5, 46], specifying exactly which of the patterns should be reported is often a subject for discussion. In this Chapter we focus on reporting all SCs whose length is above a given threshold $\ell$ containing at least $m$ subtrajectories. A cluster $C$ of (sub)trajectories has length at least $\ell$ if there exists a (sub)trajectory in $C$ whose length along $T$ between its start- and endpoint is at least $\ell$.

**Definition 2.** *Given a trajectory* $T$ *with* $n$ *vertices, a* subtrajectory cluster $C_T(m, \ell, \varepsilon)$ *for* $T$ *of length* $\ell$ *consists of at least* $m$ *subtrajectories* $\tau_1, \ldots, \tau_m$ *of* $T$ *such that the time intervals for any two subtrajectories in the cluster overlap in at most one point, the distance between the subtrajectories is at most* $\varepsilon$*, and at least one subtrajectory has length* $\ell$*.*

Many of the fundamental problems in this area of research have one issue in common, namely calculating the similarity between two trajectories (or subtrajectories). Simplified, the problem is as follows. Given two (polygonal) curves $f$ and $g$, one would like to match them up in an optimal way (i.e., find a continuous mapping from the points of $f$ to the points of $g$, so that the mapping maps the endpoints of one curve to the endpoints of the other curve). Such a mapping is especially useful if it is associated with an appropriate metric. The Fréchet metric is one of the most natural measures of this type.

## 4.3 THE DATA STRUCTURE

In the following sections we will show how to construct the data structure in the GPU needed to compute the subtrajectory clusters, then in Section 6.3 we show how it can be used to report the clusters. The algorithm builds upon the corresponding CPU algorithm by Buchin et al. [30], which we now will describe in more detail.

Their algorithm uses the *free space diagram* of two polygonal curves $h$ and $g$, which is a geometric data structure introduced by Alt and Godau [34] for computing the Fréchet distance. Let $h$ be a polygonal curve with $n$ vertices $p_1, \ldots, p_n$ and let $g$ be a polygonal curve with $m$ vertices $q_1, \ldots, q_m$. We use $\phi_h$ to denote the following natural parametrization of $h$. The map

$\phi_h\colon [1,n] \to \mathbb{R}^d$ maps $i \in \{1,\dots,n\}$ to $p_i$ and interpolates linearly in between vertices. The free space diagram of two polygonal curves $h$ and $g$, with $n$ and $m$ vertices, respectively, is the set

$$F_\varepsilon(h,g) = \{(s,t) \in [1,n] \times [1,m] : |\phi_h(s), \phi_g(t)| \leqslant \varepsilon\}$$

which describes all tuples $(s,t)$ such that the points $\phi_h(s)$ and $\phi_g(t)$ have Euclidean distance at most $\varepsilon$, the so-called free space of the diagram. See Fig. 24 for an example of a free space diagram with $n-1$ columns and $m-1$ rows, the white region in the diagram is the free space. Alt and Godau [34] showed that the Fréchet distance between $h$ and $g$ is less than $\varepsilon$ if and only if there exists an $xy$-monotone path in the free space of $F_\varepsilon(h,g)$ from $(0,0)$ to $(n,m)$.

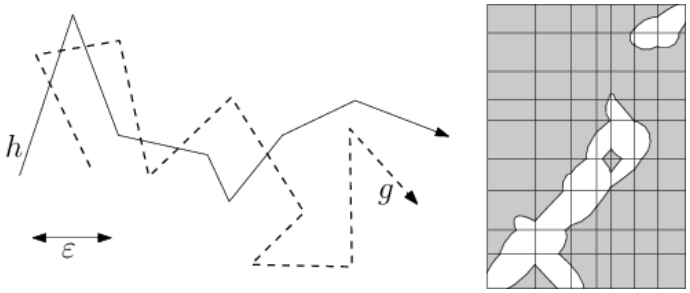



Figure 24: Two trajectories $h$ and $g$ and the corresponding free space diagram with respect to $\varepsilon$.

Consider a free space diagram $F_\varepsilon(T,T)$ of a polygonal curve $T$ and itself. In [30] it was noted that if a subtrajectory $\tau$ of length $\ell$ within $T$ belongs to a subtrajectory cluster $SC(m,\ell,2\varepsilon)$ then there exist $m$ $xy$-monotone paths from the vertical line intersecting the $x$-axis at the start point of $\tau$ and the vertical line intersecting the $x$-axis at the endpoint of $\tau$. Furthermore, the projections of the $xy$-monotone paths onto the $y$-axis are not allowed to overlap (otherwise the subtrajectories in the cluster would overlap along $T$). We call $\tau$ the reference subtrajectory. See Fig. 25 for an example. Thus the aim is to find all maximal length intervals along the $x$-axis where there are $m$ $xy$-monotone paths from the left vertical boundary to the right vertical boundary of the interval.

We assume that the reference subtrajectory will always start and end at a vertex of $T$, while no restriction is made on the start and end points of the other members of the SC. In $F_\varepsilon(T,T)$ each *cell* corresponds to two line segments of $T$ and the free space in one cell is the intersection of the cell with an ellipse, possibly degenerated to the space between two parallel lines [34]. There are at most eight intersection points of the boundary of the cell with the free space. We call these intersection points *critical points*, see Fig. 26. For each critical point, place a vertex on it in the free space diagram. Furthermore, all critical points in the free space of the corresponding row or column are propagated. That is, propagate critical points on vertical cell boundaries horizontally to the right, and critical points on horizontal cell boundaries vertically upwards. Consider a critical point $p$ on a vertical boundary. Move $p$ horizontally to the right in the diagram until it hits a non-free space boundary. Every time $p$ moves over a cell boundary an intersection point is added. When $p$ hits the free space boundary the propagation stops. Each critical point can generate at most $n-1$ propagated critical points. Figure 27 shows an example of horizontal propagation.

Following the idea presented in [30] we use a labeled graph with the critical points and the propagated points as vertices so we can later navigate through the graph in order to find the clusters. Each critical and propagated point has a label that stores the smallest $x$-coordinate reachable by an $xy$-monotone path via this point.

These are the data structures needed to compute the clusters. In this section we will show how to construct these in the GPU. After the structure has been constructed one can easily find the subtrajectory clusters. In Section 6.3 we will show how this is done in the GPU. However, for the reader to have a full understanding of the approach we next describe the CPU approach.

Using the labeled graph, one can determine whether there are $m$ cluster curves between two vertical lines $l_s$ and $l_t$. This is done by greedily searching for monotone paths from $l_s$ (start boundary) to $l_t$ (end boundary). Let $i$ be the $x$-coordinate of $l_s$. Start at the topmost vertex on

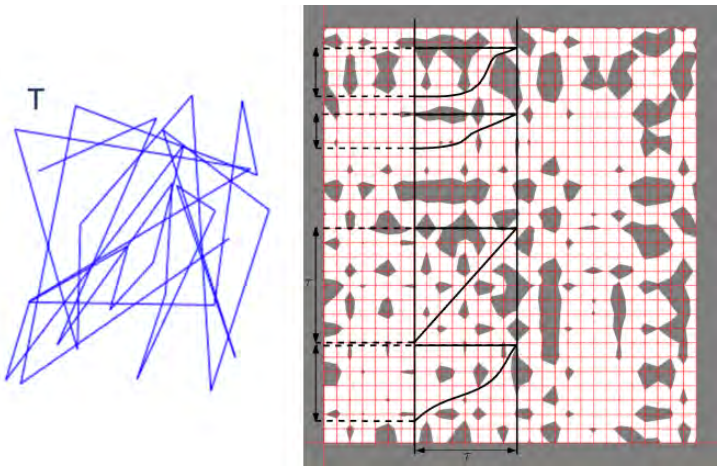



Figure 25: A free space diagram of a trajectory T with itself. The four xy-monotone curves represents a cluster with four subtrajectories of T within distance $\varepsilon$ from $\tau$, including $\tau$ itself.

$l_t$ which has an outgoing edge whose label is at most $i$. In each vertex follow the topmost edge whose label is again at most $i$. This ends on a vertex $(i, j)$ on $l_s$. Continue with the topmost vertex on $l_t$ at height at most $j$ which as before has an outgoing edge whose label is at most $i$. Stop when we have found $m$ curves, or no more vertices on $l_t$ with an outgoing edge whose label is at most $i$ exist. Note that the edge labels prevent us from going into dead ends in the graph.

To obtain the labeled graph we need to show how to (1) compute the free space diagram (Section 4.3.1), (2) propagate the points (Section 4.3.2) and (3) label all the points of the free space diagram (Section 4.3.3) in the GPU model. The labeled graph is then used to report the clusters (Section 4.4.1).

In the following sections we will focus on a subproblem to the subtrajectory clustering problem. Given two trajectories $h$ and $g$, and a positive constant $\varepsilon$ we next show how to construct the free space diagram for $h$ and $g$ with respect to $\varepsilon$ in the GPU.

### 4.3.1 *Computing the free space diagram in the GPU*

Inside the GPU it is not possible to use dynamic memory so all the memory required by a kernel must be allocated before it is executed. Furthermore the data structure must be optimized so that a maximum number of threads can access any position in memory in constant time. Here we will discuss the structure used to compute the free space diagram.

#### 4.3.1.1 *The free space diagram inside the GPU*

As input we are given two trajectories $h = \langle h_1, \ldots, h_n \rangle$, $g = \langle g_1, \ldots g_m \rangle$ and a positive constant $\varepsilon$. Next we will show how to construct the free space diagram in the GPU. Each cell in the free space diagram is a convex polygon, but instead of storing the whole polygon structure we just store the critical points (the intersections between the boundary of the cells with the free space), see Fig. 26. There are at most eight critical points per cell in the free space diagram.

The critical points of each cell is divided into two sets; the set of points on the top boundary (T-set), not including the ones in the corners of the cell, and the set of points on the right boundary (R-set). Thus each cell in the free space diagram will be stored as two sets. We store every critical point as floats in the interval $[0, 1]$ indicating the distance from the left or bottom boundary of the cell.

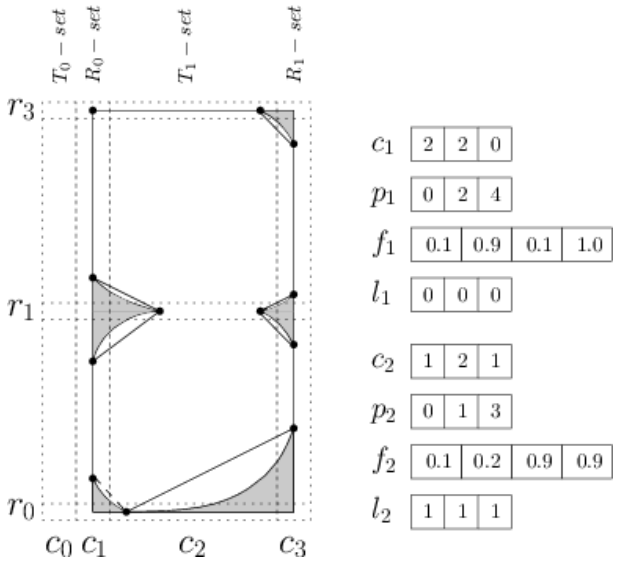



Figure 26: An example of $\mathcal{F}(h, g)$ where $n = 1$ and $m = 2$.

We create a matrix M with $2(n+1)$ columns and $m+1$ rows, $M[0..2(n+1)-1, 0..m]$. For each column in the free space diagram we have two columns in M; one for the T-set and one for the R-set. The first two columns of M correspond to the left boundary of the leftmost column in the free space diagram. Note that the T-set in this column ($T_0 - set$) is an empty set and it is only included to make the structure uniform. Finally, there is an extra row for the points on the bottom boundary of the bottommost row, Fig. 26.

Inside the GPU, each column $i$ of the matrix M is stored in three arrays. The counting array $c_i[0..m]$ is an integer array such that $c_i[j]$ stores the number of critical points in cell $(i, j)$ of M. The positioning array $p_i[0..m]$ is also an integer array and stores the prefix sum of $c_i$, that is

$$p_i[0] = 0 \quad \text{and} \quad p_i[j > 0] = \sum_{k=0}^{j-1} c_i[k] = p_i[j-1] + c_i[j-1].$$

Finally, the critical points array $f_i[1..k]$ is a float array, where $k$ is the total number of critical points in column $i$ of M. With this structure we can easily access any cell or critical point using the fact that the critical points of the cell $(i, j)$ in the free space diagram starts at position $(p_{2i}[j] + 1)$ of $f_{2i}$ and is stored in $c_{2i}[j]$ consecutive positions of $f_{2i}$. We will refer to this entire structure, $(M, c, p, f)$, as $\mathcal{F}(h, g)$, and $\mathcal{F}_i(h, g)$ as the $i$th column of $\mathcal{F}(h, g)$. An example is shown in Fig. 26.

Finally an additional array $l$ is initialized to store the label of each point (to be defined in Section 4.3.3).

In the rest of this section we assume that $\mathcal{F}_i(h, g)$ fits in the GPU memory. However this is not always possible and in Section 4.4.2.1 we will explain how to partition the free space diagram into smaller pieces where the data structure for each piece can fit in the GPU memory.

#### 4.3.1.2 *Computing the critical points using the GPU*

An optimal scenario in parallel computation is when a single element can be computed and reported without the interference from other elements and without requiring the results from the computation of other elements. Luckily the computation of the critical points is an optimal scenario since each critical point is independent from the others, which allows us to compute them very efficiently in parallel. This process is performed in three steps:

*Step 1:*

Launch a parallel kernel with $(n+1) \times (m+1)$ threads, one thread for each cell $(i, j)$ in the free space diagram. Here we imagine a 0th row and column. Each thread computes the number of critical points in the T- and R-points for that cell. The number of points in the T-set of $(i, j)$ is stored in $c_{2i}[j]$ and the number of points in the R-set of $(i, j)$ is stored in $c_{2i+1}[j]$.

*Step 2*

Perform a parallel scan [47] over each $c_i$ to obtain the values needed to populate the position array $p_i$.

*Step 3:*

For each $i$, $0 \leqslant i \leqslant 2n+1$, allocate $p_i[m] + c_i[m]$ memory to $f_i$. Launch a kernel with $(n+1) \times (m+1)$ threads, one per cell in the free space diagram. A thread $(i, j)$ stores the $T_i$ and $R_i$ critical points starting at position, $p_{2i}[j] + 1$ for the T-set and $p_{2i+1}[j] + 1$ for the R-set.

### 4.3.2 *Propagate the critical points*

We need to propagate the critical points of $\mathcal{F}(h, g)$ vertically and horizontally. This process is easy to parallelize since the propagation of a critical point does not affect the rest of the propagations. The process is performed in four steps, see Fig. 27.

*Step 1:*

For each $i$, $0 \leqslant i \leqslant 2n+1$, create an auxiliary array $\texttt{accum}_i$ of size $m+1$ which will be used to keep track of the number of propagated points in each cell. To populate `accum` we launch a kernel with as many threads as critical points in $\mathcal{F}(h, g)$, assigning to each thread a critical point and propagating that critical point horizontally or vertically, depending on if the critical point lies on a horizontal or vertical boundary.

Consider a critical point $q$ in a cell $(i, j)$ of $\mathcal{F}(h, g)$. If column $i$ represents a T-set ($i$ is even) then $q$ is propagated vertically upward otherwise, if column $i$ represents an R-set, it is propagated horizontally to the right. Assume without loss of generality that column $i$ is even. Move $q$ vertically upwards until it either hits a non-free space boundary or until it intersects a horizontal cell boundary. Every time $q$ hits a cell boundary an intersection point is added. When $q$ hits the free space boundary the thread immediately terminates.

Thus, if $q$ belongs to a T-set then it might generate up to $m - i$ new propagated points and if it belongs to an R-set then it might generate up to $n - j$ new propagated points. At each iteration in the propagation the corresponding `accum` cell value is incremented by 1 indicating that there is a new propagated point in this cell. At the end of the process `accum` stores the number of propagated points in each cell.

A complication is that two different critical points can be propagated to the same cell boundary, and consequently to the same `accum` position at the same time in the process. This can lead to memory interference between threads. To avoid this situation we use atomic operations when writing to `accum`.

*Step 2:*

For each $i$, $0 \leqslant i \leqslant 2n+1$, create an integer array $c'_i[0..m]$ and set $c'_i[j] = c_i[j] + \texttt{accum}_i[j]$, for $0 \leqslant j \leqslant m$. Also create an integer array $p'_i[1..m]$ and set $p'_i[j]$ to the sum of $c'_i[1..j]$.

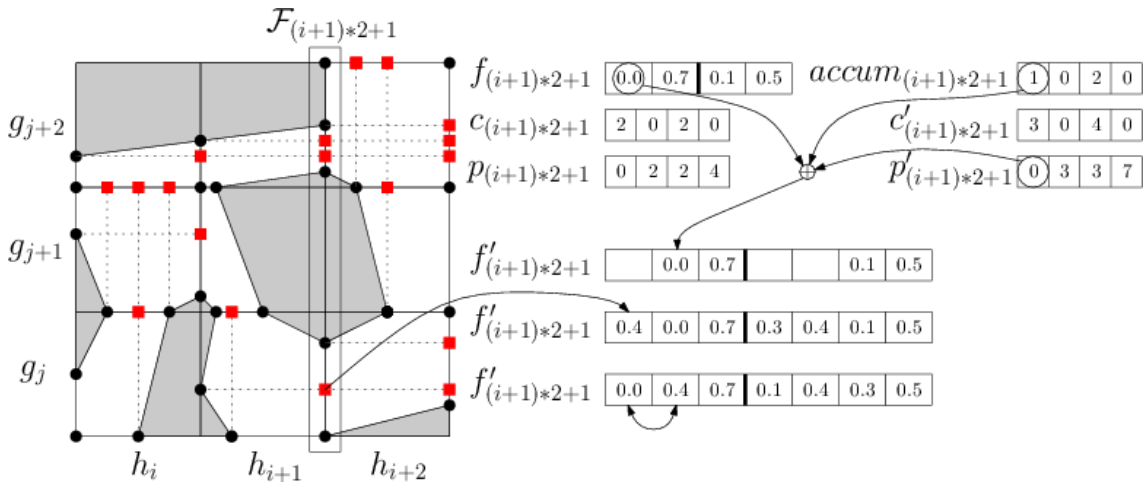



Figure 27: Critical points marked as black disks and the propagate points as red squares.

*Step 3:*

For each $i$, $0 \leqslant i \leqslant 2n+1$, create $f'_i$ of size $p'_i[j] + c'_i[j]$ and then run a kernel with one thread per critical point in $\mathcal{F}(h, g)$. Note that we now repeat the propagating process but this time storing the propagated points in $f'$. Once this is done $\mathcal{F}'(h, g)$ stores the free space diagram including all the propagated points.

Note that the corner points never have to be propagated (either they already exist or they do not lie in the free space).

*Step 4:*

For each cell $(i, j)$ in $M$, if it is a T-set then place the leftmost point at position $(p'_i[j] + 1)$ of $f'_i$ and the rightmost point at position $p'_i[j+1]$ of $f'_i$. Similarly, if it is an R-set then place the bottommost point at position $(p'_i[j] + 1)$ of $f'_i$ and the topmost point at position $p'_i[j+1]$ of $f'_i$. This can be done in parallel with one thread per cell, Fig. 27. This step is needed to speed up the labeling step that will be describe next.

### 4.3.3 *Parallel labeling*

The labeling process is the most expensive part of the algorithm since there is no natural way to parallelize it. The reason for this is that the label of the points in cell $(i, j)$ in the free space diagram depends on the labels of all the points to the left and below cell $(i, j)$. We say that a point $u$ is *reachable* from a point $v$ if there is an $xy$-monotone path from $u$ to $v$. Imagine the situation showed in Fig. 28 were the regions reachable from $a$ are marked in stripes, the regions reachable from $b$ in lightgray and the regions reachable from both $a$ and $b$ is shown as checkered.

In Fig. 28 the point $v$ (f[43]) is reachable from both $a$ and $b$. If we label all the points in parallel then $f'[43]$ could be labeled with any of the two labels $a$ or $b$ depending on the thread race condition. This is a problem since the algorithm requires information at each point in the free space diagram of the leftmost point it can reach with an $xy$-monotone path. This is crucial in deciding if there exists a subtrajectory cluster or not.

To get around this problem we label the points as follows. We use an integer array $l[1..k]$, where $k$ is the total number of critical and propagated points in the free space diagram. For each critical and propagated point $p$ we store a label in $l(p)$, where $p$ is associated to an integer in $[1, k]$. Initially a critical point is labeled with the row index of the cell it belongs to and a propagated point is initially labeled with the same label as the label of the critical point that induced it.

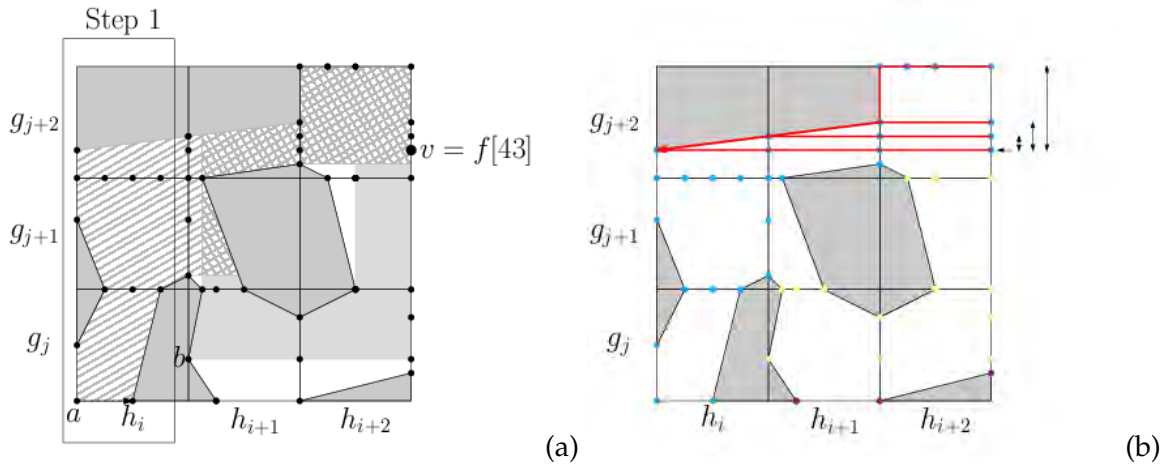



Figure 28: (a) The visibility of points. (b) Four potential clusters found.

In order to avoid excessive relabeling we compute the labeling in several steps. Recall that the labels in $f'_0$ and $f'_1$ of $M$ do not have to be re-labeled. So first label all points in $f'_2$ (T-sets) with labels from the points in $f'_1$, then $f'_3$ with the labels from the points in $f'_2$ and $f'_1$, $f'_4$ with the labels from the points in $f'_3$, $f'_5$ with the points in $f'_4$ and $f'_3$, and so on. To do this efficiently in parallel we run a kernel with one thread per point (critical and propagated) in $f'_i$ and each thread labels all its visible points. Navigating through $\mathcal{F}$ is easily done using the first and last point of each $f'_i[j]$.

Even though this avoids a large part of the unnecessary re-labeling we still have points that are labeled several times. Some of them can be avoided by defining a new stop condition based on the idea of which points are reachable by an $xy$-monotone path. Observe that all points reachable from a point $u$ are reachable from a point $v$ if $u$ is reachable from $v$. Using this observation, we can force a thread to stop when it tries to re-label a point $v$ with the same label, or a smaller label, since this implies that another thread already labeled it and will label all other points that are reachable from $v$.

This concludes the construction of the data structure.

## 4.4 REPORTING SUBTRAJECTORY CLUSTERS

Which clusters that should be reported is often a topic for discussion, since it often depends on the application. For instance, is it necessary to report a subtrajectory which has already been reported in another cluster? Do we allow two subtrajectory clusters to overlap? Do we want the clusters of maximum length or the ones with a maximum number of subtrajectories?

Another major problem is the huge amount of clusters reported if no restrictions are made on the output. Especially if subtrajectories are allowed to overlap, or if one allow clusters that are very similar but have different representative subtrajectories. Therefore we made the following restrictions on the subtrajectory clusters.

1. Two subtrajectories in a single cluster may only overlap along $T$ in a single point. (Section 4.4.1)
2. A representative subtrajectory must start and end at a vertex of $T$. (Section 4.4.2)
3. A subtrajectory in a cluster must have length greater than zero. (Section 4.4.3)
4. Subtrajectories belonging to different clusters may only overlap along $T$ in a single point. (Section 4.4.4)
5. The reported cluster should have maximal length in the following sense. No two clusters can be merged into a single cluster (Section 4.4.2). A representative subtrajectory cannot

be extended in either direction without destroying the cluster or overlapping with another cluster (Section 4.4.4).

Consider a free space diagram $F_\varepsilon(T,T)$ of a polygonal curve $T$ and itself. Recall that if a subtrajectory $\tau$ of length $\ell$ within $T$ belongs to a subtrajectory cluster $SC(m,\ell,2\varepsilon)$ then there exist $m$ $xy$-monotone paths from the vertical line in the free space diagram intersecting the $x$-axis at the start point of $\tau$, denoted $l_s$, and the vertical line intersecting the $x$-axis at the endpoint of $\tau$, denoted $l_t$. Furthermore, according to restriction (2) the projections of any two $xy$-monotone paths onto the $y$-axis are not allowed to overlap in more than a single point (otherwise the subtrajectories in the cluster would overlap along $T$). We call $\tau$ the reference subtrajectory. See Fig. 25 for an example. Thus the aim is to find all maximal length intervals (restriction (5)) along the $x$-axis where there are $m$ $xy$-monotone paths from the left vertical boundary to the right vertical boundary of the interval such that the projections onto the $y$-axis of any two $xy$-monotone paths overlap in at most one point.

### 4.4.1 *Finding $m$ $xy$-monotone paths in the free space diagram*

Consider the case when we have a fixed representative subtrajectory, that is the left and right boundaries ($l_s$ and $l_t$) are fixed. In [30] it was shown that a greedy approach can be used in the sequential setting to decide if there is $m$ $xy$-monotone paths between the left and right boundary. That is, start at the topmost vertex on the right boundary which has an edge connected to a vertex whose label is $l_s$. In each vertex follow the topmost edge (decreasing along the $y$-coordinate) connected to a vertex whose label is again $l_s$. This ends on a vertex $(l_s, j)$ on the left boundary. Continue with the topmost vertex on the right boundary at height at most $j$ which, as before, has an edge connected to a vertex with label $l_s$. Stop when $m$ curves have been found, or no more vertices on the right boundary with an edge connected to a vertex with label $l_s$ exist. Note that this guarantees that restriction (1) and the first part of restriction (5) are fulfilled.

We propose a semi parallel approach of this process. We use a thread per point in $\mathcal{F}_{2t+2}$. Each thread, which point has label $l_s$, moves along adjacent cells of $\mathcal{F}$ until they reach a point in $\mathcal{F}_s$ with label $l_s$. This is done without any thread interference and at the same time. Because each thread has follow its independent path, the resulting paths may overlap. The path overlap removing process is difficult to parallelize because the removal of a path depends on the above paths which can be removed or not. This leads into several kernel executions to guarantee correct results. We tested this approach but it is slower than executing the sequential algorithm in the CPU, so we decided to merge our approach with the one presented in [30]. We found the $xy$-monotone paths in parallel using the GPU and then we remove the overlapped ones using sequential CPU algorithm.

### 4.4.2 *Sweep the free space diagram*

Above we showed how to find all $xy$-monotone paths in the free space diagram of two given trajectories $h$ and $g$. We can use the same approach to report subtrajectory clusters within a trajectory $T$. This can be done by considering $T$ as the trajectory $h$ and also as the reference trajectory $g$, having $T$ in the both dimensions of the free space diagram. We want to report all subtrajectory clusters while we consider all the possible reference subtrajectories within $T$. To this end we proceed as follows.

We define $s$ and $t$ as the start and end vertices of a possible representative subtrajectory $T(s,t)$ of $T$, respectively. Let $l_s$ and $l_t$ be the vertical lines in the free space diagram with $x$-coordinate $s$ and $t$, respectively. The idea is to sweep $l_s$ and $l_t$ from left to right while maintaining the number of $xy$-monotone paths starting on $l_s$ and ending on $l_t$ while fulfilling the restrictions. The number of $xy$-monotone paths between $l_s$ and $l_t$ is denoted $\lambda(s,t)$ and is computed using the approach described in the previous section.

Initially set $s = 0$ and $t = 1$. If $\lambda(s,t) < m$ then increase both $s$ and $t$ by 1, until $\lambda(s,t) \geqslant m$. This implies that there is a subtrajectory cluster of size at least $m$ and length $|T(s,t)|$ having $T(s,t)$ as

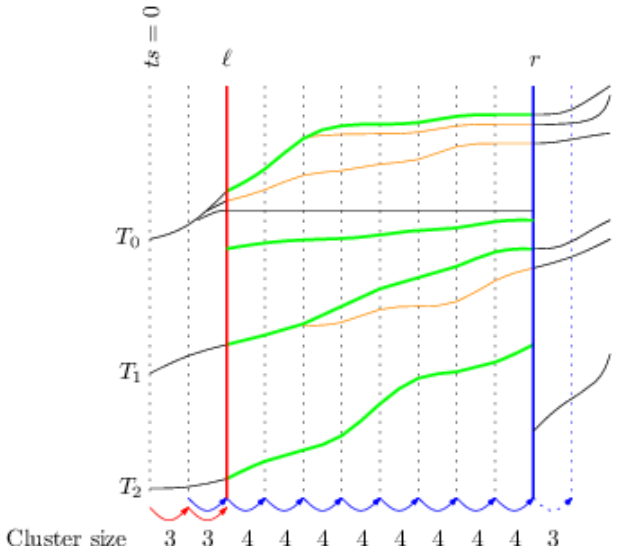



Figure 29: Left and right movement for a $m = 4$. The reported subtrajectories are drawn fat.

its representative subtrajectory. Since we need to find the maximal length cluster (restriction (5)) we increase $t$ until we reach the first value where $\lambda(s,t) < m$. Then report the subtrajectory cluster with $T(s,t-1)$ as a representative subtrajectory.

The process starts all over again by setting $s = t-1$. This is repeated until $t = n$. Since the event points of the sweepline is the integer coordinates this guarantees that restriction (2) is fulfilled. An example of the process is shown in Fig. 29.

It is important to note that when $t$ moves one step to the right we only compute the $xy$-monotone paths from $l_{t-1}$ to $l_t$, since all the necessary information is already stored in the vertices along $l_{t-1}$. That is, paths found are connected with the already stored paths on $l_{t-1}$ obtaining $xy$-monotone paths from $l_s$ to $l_t$.

#### 4.4.2.1 *Super columns*

As mentioned in Section 4.3 the approach presented in previous sections assumes that the whole $\mathcal{F}(T,T)$ fits in the GPU memory. This is obviously unrealistic since the structure uses quadratic space and the trajectory can be very large, even after it has been simplified. To overcome this problem we compute $\mathcal{F}(T,T)$ in parts using what we call super columns, that is, a set of $M$ consecutive columns that fit inside the GPU.

Initially compute and store $\mathcal{F}_0$ to $\mathcal{F}_{M-1}$ in the GPU. Super columns are stored as a CPU vector array $S[0..n]$, where each $S[i]$ contains a pointer to the position in the GPU memory storing $\mathcal{F}_i$. Next, search for subtrajectory clusters with $s$ and $t$ in this interval. When $s$ is incremented the previous column is removed from the GPU memory.

The right boundary, $t$ is incremented as described above until it reaches the end of the super column. The next super column is then computed and the corresponding pointers in the GPU memory is added to the CPU array $S$. Since all the necessary information is stored in the last column $\mathcal{F}_{M-1}$ (Section 4.4.2) we can free up the GPU memory storing all the earlier column of the super column (see also Section 4.4.4). This implies that even if a subtrajectory cluster is very long the algorithm will find it.

Note that this approach requires that the size of the GPU memory is $\Omega(n)$ since at least one column, $\mathcal{F}_i$ is required to fit in memory. We believe it is a reasonable assumption.

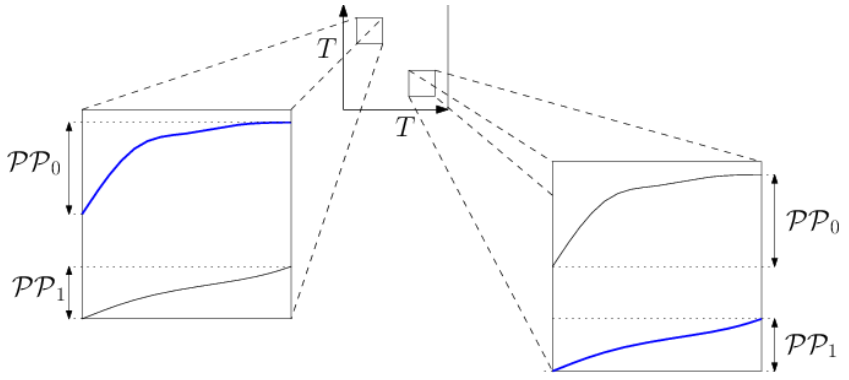



Figure 30: Two copies of the same two subtrajectories occurring at two places in the free space diagram.

### 4.4.3 *Paths of zero length*

Another issue we have to deal with is when a monotone increasing path has length 0 (constraint (3)). Even though it fulfills the cluster condition and it is not overlapping any other subtrajectory, it makes no sense from a practical point of view. This usually happens when $\varepsilon$ is very large with respect to $\ell$, or when a representative trajectory has stayed within a small spatial region for a long time, which is not an uncommon situation in real data sets.

To handle this constraint we add an extra constraint to the $xy$-monotone paths between $l_s$ and $l_t$, namely that no path is allowed to be a single horizontal segment. Note that any restriction on the length of the segments can easily be added.

### 4.4.4 *Avoiding overlapping clusters*

To avoid reporting very similar clusters the fourth constraint guarantees that a subtrajectory that is already included in a found cluster cannot be included (even in parts) in a different cluster. The problem occurs for two reasons; (1) we check all possible start and endpoints for the representative clusters during the sweep and (2) the free space diagram is symmetric we will find the same cluster with the reference subtrajectory swapped, see Fig. 30.

To handle this we create an array of size $n$ initialized to 0. Each time a cluster is reported we set all the time steps of the subtrajectories involved in the cluster to 1. Then when we search for the $xy$-monotone paths we check if any of its time steps have been already been reported. If they have the corresponding subtrajectory is discarded.

This pruning step also has the additional effect that when a maximal length reference subtrajectory $T(s,t)$ for a cluster has been found we do not have to consider any part between $s$ and $t$ again. The result of this is that all the information needed during the sweep can be stored in column $t-1$, we do not have to keep any information in the columns between $s$ and $t-1$, which gives a considerable speed-up.

## 4.5 COMPLEXITY ANALYSIS

When we analyze the complexity of a GPU algorithm, we consider the total work and the thread work. The total work is the total number of instructions performed by the algorithm. The thread work is the number of operations executed by a single thread. The thread work (and the number of threads) gives an idea of the algorithms degree of parallelism. We analyze the time complexity of the algorithm following the steps in Section 4.3. A summary of the analysis can be found in Table 4.

The first step of the algorithm is the initialization of the $\mathcal{F}(h, g)$ structure, that is, setting entries in the counting arrays to zeros, costs $O(nm)$ time which can be done by $nm$ threads in $O(1)$ time.

Computing the critical points is performed in three separate steps where each step costs $O(nm)$ which can be performed by $nm$ threads in constant time. The first and last steps runs one thread per cell where each thread performs a constant amount of work. The second step is harder, but using the Nvidia parallel scan [47] one can do it with constant amount of work per thread using $O(nm)$ threads.

The propagation of the points (Section 2.2) is also performed in several steps. The initialization of `accum` costs $O(m)$ in total, performed by $m$ threads in parallel. Then, in the first step, every critical point may generate a linear number of propagated points. Thus using one thread per critical point, $O(nm)$ in total, means that $O(n + m)$ work is performed per thread. The same bound holds for the third step of the propagation phase.

The second and fourth steps of the propagation requires $O(nm)$ work in total, thus using $O(nm)$ threads each thread only performs $O(1)$ operations.

The most time consuming step of the algorithm is the parallel labeling step discussed in Section 4.3.3. In the worst case every critical point can be propagated to every column to the right, or every row above. Thus the algorithm performs the labeling in $O(n)$ rounds and in each round $O(nm)$ threads are launched using $O(n + m)$ work/thread.

However, in practice the number of propagated points is much less which is shown in Section 4.6.2. One can also argue that the number of propagated points is much smaller of one assumes some uniform distribution of critical points along the boundary. Assume that the position of the critical points along the boundary of each cell is distributed uniformly at random. Note that here we assume that every cell side has a critical point, we will discuss this assumption in more details below. Consider a critical point $p$. Assume without loss of generality that $p$ belongs to an $R_j$-set and lies in cell $(j, k)$. What is the probability that $p$ will generate exactly $i$ propagated points? For this to happen we know that the horizontal projections of $p$ onto the right boundaries of cells $(j + 1, k), \ldots, (j + i, k)$ must lie in free space. While the projection onto the right boundary of cell $(j + i + 1, k)$ lies in the non-free space. So the probability for exactly $i$ propagated points is bounded by $1/2^{i+1}$. Thus summing up over all values of $i$ we get:

$$\sum_{i=1}^{\infty} i/2^{i+1} < 1.$$

Hence the expected total number of propagated points is at most the number of critical points. As mentioned above this is only for the case when every boundary cell has critical points. However, as observed in our experiments (Section 4.6.2), a large part of the cells in a free space diagram lie completely inside the free space and, hence, has no critical points on their boundary. Let $FC_k$ be the set of cell in row $k$ that lies entirely in the free space. Using a similar argument as above one can argue that for a cell $(j, k)$ with a critical point on the right boundary the expected number of propagated points is at most 1. So if there are cells in $FC_k$ following cell $(j, k)$ on row $k$ then the expected number of propagated points is at most 1 per cell in $FC_k$. Summing up this gives us that the expected number of propagated points, in the special case when the critical points along the boundary of each cell are distributed uniformly at random, is upper bounded by the number of critical points plus the number of corner points in the free space diagram.

Thus the algorithm performs the labeling in $O(n)$ rounds and in each round $O(nm)$ threads are launched using $O(1)$ expected work/thread.

In the above analysis we assumed that the critical points are uniformly distributed along the boundaries of the columns and rows. This is, of course, not a valid assumption for any input data. However, it might explain why the bounds in the experiments are so much better than the worst-case bounds.

Finally the algorithm needs to find the $xy$-monotone paths. The sweep is performed in $n$ rounds. Each round generates one thread per point on right boundary. Number of points is $O(nm)$. Each thread finds the topmost path reaching $l_s$, which may require $O(1)$ time per thread, $O(nm)$ total work. When paths have been generated one has to remove overlapping paths. Note that if a path

is selected then all other overlapping paths can be skipped, the top-most non-overlapping path can be found in $O(1)$ using one thread per point on the right boundary. So given the $xy$-monotone paths one can find the non-overlapping paths in linear time with respect to the number of non-overlapping paths. Since a simple upper bound is $n$, the total work per thread is $O(n)$.

| | Total work | Thread work |
|---|---|---|
| $\mathcal{F}(h, g)$ initialization | $O(nm)$ | $O(1)$ |
| Critical points | $O(nm)$ | $O(1)$ |
| Count propagation | $O(nm)$ | $O(n + m)$ |
| Report points | $O(nm)$ | $O(1)$ |
| Parallel labeling | $O(n^2m)$ | $O(1)$ (done in $n$ rounds) |
| $xy$-monotone path | $O(nm)$ | $O(1)$ |
| Remove overlapping $xy$-monotone path | $O(nm)$ | $O(n)$ |

Table 3: Worst-case complexity analysis

## 4.6 EXPERIMENTAL RESULTS

For the GPU algorithms we used an i5-200 CPU with a Nvidia GTX 580. For the CPU algorithms we ran the experiments on an i7-2620M@2.70GHz CPU with 8 GB RAM.

We studied the running times and the influence of the input parameters using three different data sets. The trajectories of two soccer players during one match (see the diagrams in Fig. 33 and Fig. 34) and a trajectory of a bird tracked for one month (Fig. 35). The soccer data sets have $28,597$ and $27,894$ time steps, respectively, and the bird trajectory has $4,161$ time steps.

The soccer data was generate from high-definition cameras which are mounted around the pitch and during the game all objects on the pitch are tracked 10 times per second with high accuracy (5cm). Several companies [48, 49, 50] provide this service.

All the reported running times are computation times together with the CPU-GPU memory transfers needed.

### 4.6.1 *Input parameters*

The first part we tried was to modify the input parameters. Not very surprisingly this shows a relation between the number of reported clusters and the length of the clusters. In the experiments we only measured the length of the longest cluster, however, in general the average cluster length increases when the length of the longest cluster increases. In Fig. 33 left column, we increase $\varepsilon$ while $m$ remains fixed. Naturally the length of the longest cluster increases. Note that the number of reported clusters also decreases, the main reason for this is that several clusters may merge into a single cluster while only a small number of new clusters is found.

In the right column of Fig. 33, we increase $m$ while $\varepsilon$ remains fixed. As $m$ increases the constraint becomes more restrictive and the algorithm finds less and shorter clusters. This is what we expected, however, a different behavior is seen in Fig. 34 when performing the same experiment. As mentioned above, both trajectories are generated by soccer players, however, the second trajectory was generated by a goalkeeper and, hence, the distribution of the points is very different. For the goal keeper the trajectory is enclosed in a very small region and when $\varepsilon$ becomes large almost all the points of the trajectory lies within distance $\varepsilon$ from all other points of the trajectory. As a result the algorithm finds many clusters of length zero (Section 4.4.3) which are discarded and consequently, fewer clusters are reported. Note that, since $T$ is normalized in the $[0, 1]^2$ space, a 0.15 $\varepsilon$-value represents 15% of the length of the soccer field, which is approximately the size of

the region in which a goal keeper moves within. This does not happen in Fig. 33 since a "normal" player moves within a much larger area.

For the goal keeper trajectory we also tested the algorithm by fixing $\varepsilon$ and varying $m$. Again, shorter clusters were reported while the number of clusters found increased. This is counter-intuitive and we believe it is a result of the large $\varepsilon$ value (0.15).

### 4.6.2 *Critical/propagated points and labeling analysis*

The total work required by the algorithm mainly depends on the number of critical points, the number of propagated points, and the time needed to perform the labeling. In Figure 31(left column) we study the number of corner, critical and propagated points using the soccer player data set. The top left graph shows field player while the bottom left graph shown the goal keeper. In both cases we study the number of points when epsilon varies.

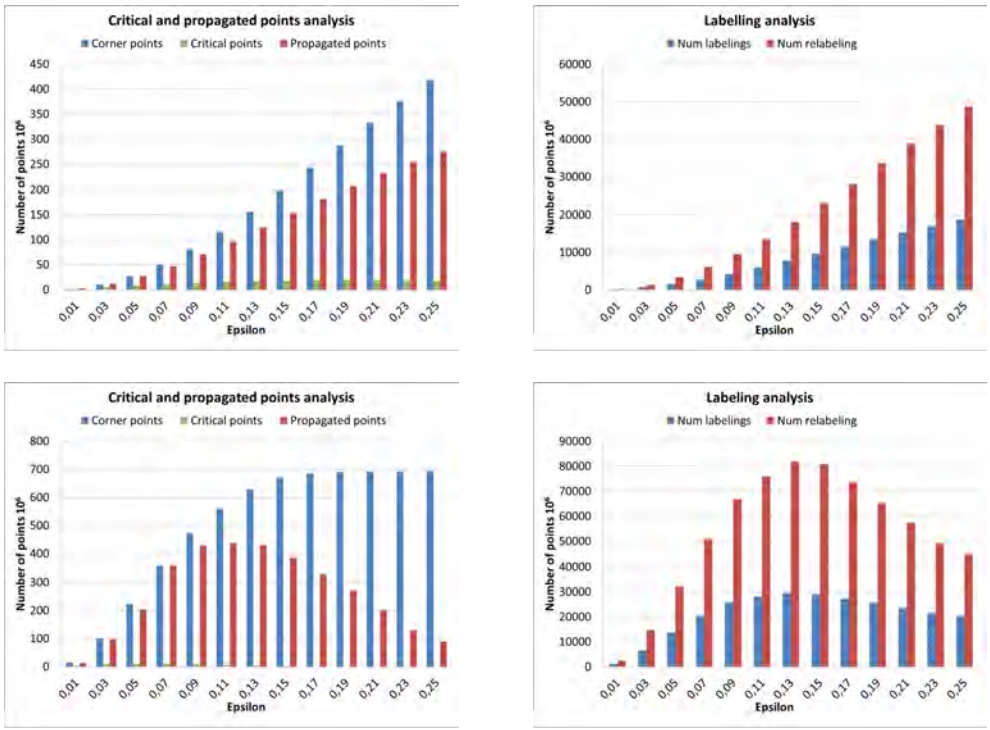



Figure 31: Analysis of the critical points and propagated points and labeling processes.

When comparing the number of corner points with the number of critical points it turns out that between 80% and 99% of the total number of points are corner points, as is shown in the two graphs on the left side of Fig. 31. The theoretical worst-case analysis of the number of propagated points much higher than what is shown by the experiments. In all the experiments even the number of corner points is greater than the total number of propagated points. Recall that in the analysis we are able to get a much better bound if we assume that the critical points are well distributed along the boundaries. In this case we can bound the number of propagated points by the sum of the number of critical points and the number of corner points. Studying the graphs in Fig. 31 this seems to be a reasonable upper bound.

The number of propagated points for the lower left graph in Fig. 31 is decreasing when $\varepsilon$ is increasing. Recall that this is the goal keeper and the reason the number of propagated points is decreasing is simply because the free space diagram hardly contains any critical points.

In the right column of Fig. 31 we study the total number of points which has been labeled or relabeled, see Section 4.3.3 for a discussion. It shows that the number of relabeled points increases with the number of critical points. For some of the cases the number of relabeled points is more

than twice the total number of labeled points. In general we can observe that the thread stop condition defined in the labeling process has a very positive effect when there is a small number of propagated points, see for example the bottom row of Fig 31. This is because the visibility stop condition has a greater effect whenever a corner point is directly visible from some other corner point. Propagated points initially have the same label as the critical point that induced it. Since the stop thread condition says that the thread terminates when it attempts to relabel a point with the same, or smaller, label. In this case propagated points may complicate the stop condition.

### 4.6.3 *Running times*

The running times shown in Figs. 33-35 are divided, just as the algorithm, into three main parts: building the free space, label the graph and finding the clusters. The *total time* includes the time of computing the subtrajectory clusters and reporting them. As expected the most expensive process is the labeling (Section 4.3.3) which is the part of the algorithm that is not using the parallelism fully. Note that the running times depend on $\varepsilon$ while $\ell$'s value has a minor effect.

Since the subtrajectory clustering algorithm using the continuous Fréchet never has been implemented earlier we cannot make a valid comparison between our GPU based approach and a basic CPU based algorithm. However, to get some kind of comparison we compared the clustering method implemented in [30] that uses the discrete Fréchet distance for the CPU model with our method that uses the continuous Fréchet distance in the GPU model, see Fig. 32. We do expect that the algorithm using the discrete Fréchet is much faster than the corresponding algorithm using the continuous Fréchet distance. The theoretical bound on the running time of the CPU algorithm is $O(n^2\ell)$, which is also supported by the experimental results. The algorithm was able to handle data sets containing up to 39k points, anything larger caused it to crash.

Comparing the run times of the two algorithms clearly shows that the running time of the CPU algorithm grows much faster than the GPU algorithm presented in this Chapter. The GPU algorithm was tested with trajectories with up to 113k points without any problems.

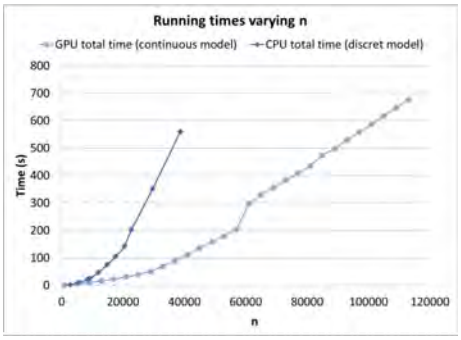



Figure 32: Clustering running times varying $n$. The results shown are, for the GPU, using the continuous Fréchet distance and, for the CPU, using the discrete Fréchet distance

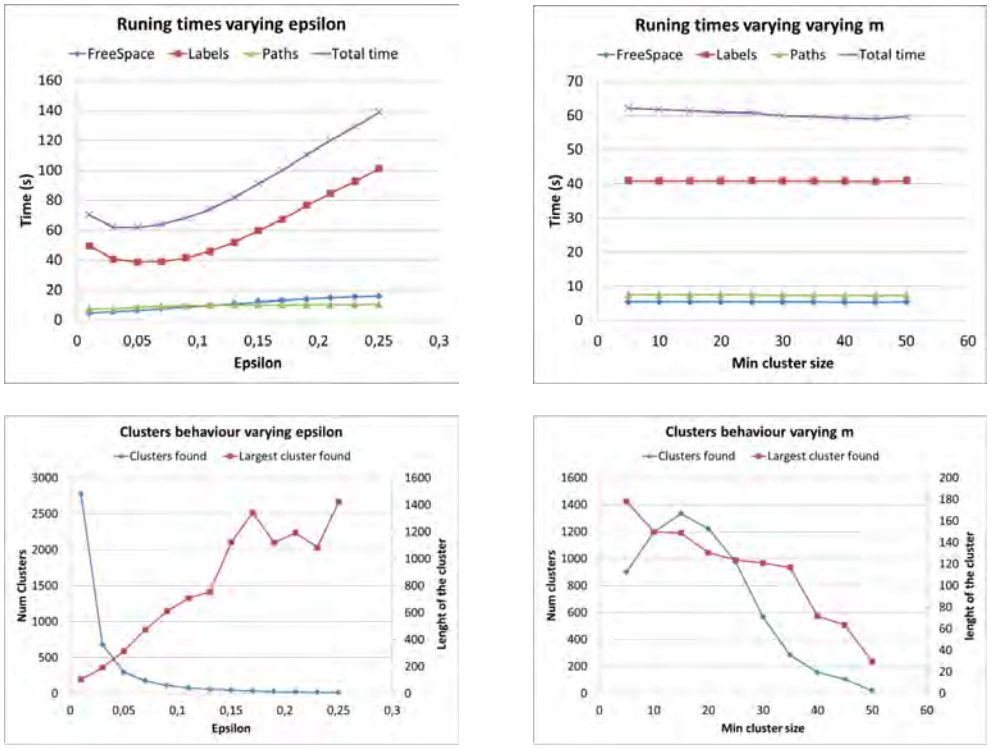



Figure 33: Soccer data set player. 28597 time steps.

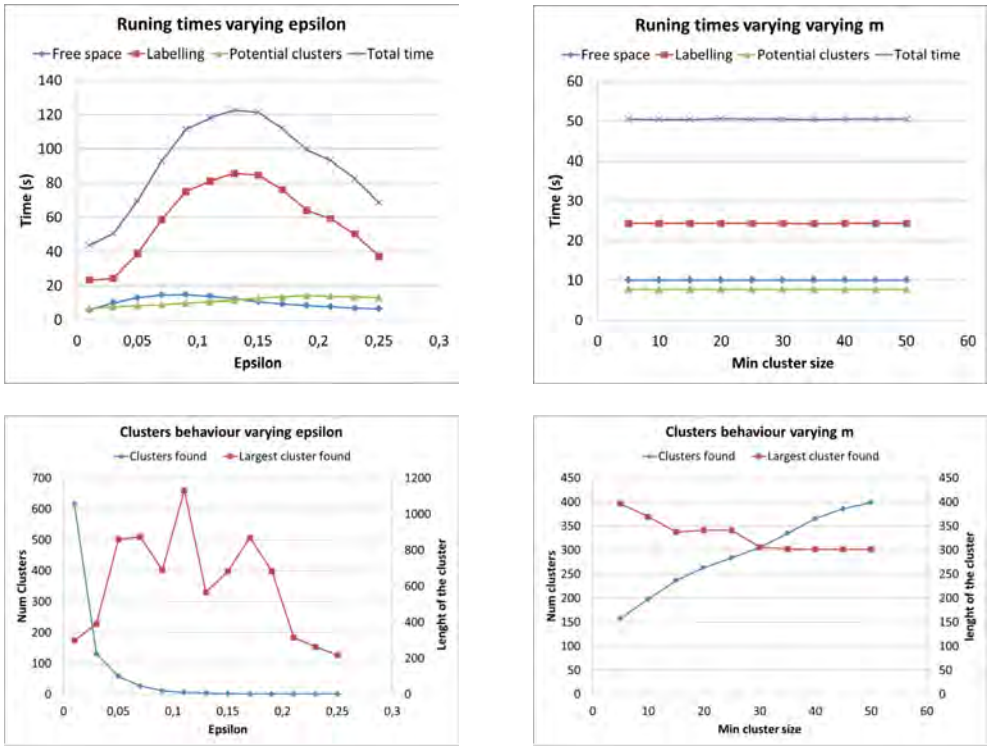



Figure 34: Soccer data set player. 27894 time steps.

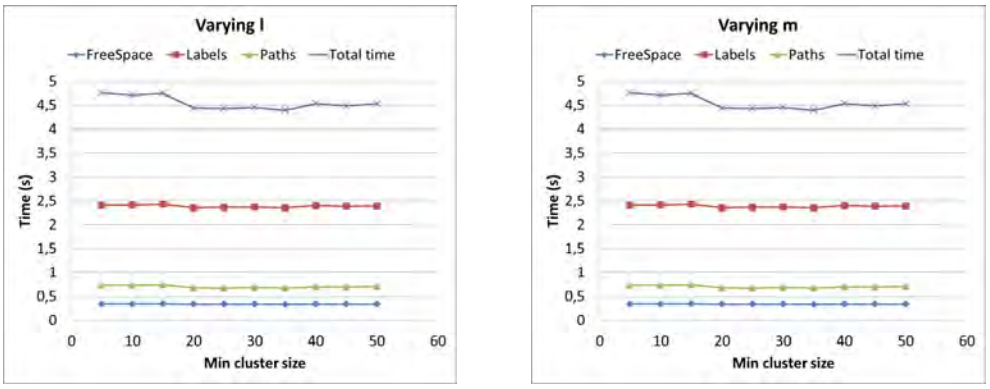



Figure 35: 1 seagull data set. 4161 time steps.

# 5

# MAXIMALS SETS

*// Dear maintainer:*
*//*
*// Once you are done trying to 'optimize' this routine,*
*// and have realized what a terrible mistake that was,*
*// please increment the following counter as a warning*
*// to the next guy:*
*//*
*// total_hours_wasted_here = 42*
*Source code comment found by – **Anonymous***

In this Chapter we face the problem of finding all extremal sets of a family $\mathcal{F}$. Let $\mathcal{F} = \{S_0, \cdots, S_{k-1}\}$ be a family of $k$ finite sets where each $S_i$ is a set over a domain D. Notice that a set may be repeated in the family, it is possible that $S_i = S_j$ for some $i \neq j$. The family $\mathcal{F}$ has a natural partial order induced by the subset relation. A set $S_i \in \mathcal{F}$ is a maximal (minimal) set of $\mathcal{F}$ if and only if there does not exist a set $S_j$ of $\mathcal{F}$ such that $S_i \subset S_j$ ($S_i \supset S_j$).

The problem of finding all extremal, maximal or minimal, sets of a family $\mathcal{F}$ of sets arises in many applications [51]. Initial algorithms for solving the problem were developed in the field of propositional logic [52]. Maximal set finding algorithms are used in the process of simplifying large satisfiability problem instances [53]. In the area of data-mining, maximal sets are used to produce condensed representations of all frequent itemsets [54] or discover maximal instances of some movement patterns among trajectory databases [55].

Let $m$ denote the total number of elements in the family $\mathcal{F}$, $m = \sum_{i=0}^{k-1} |S_i|$. Yellin and Jutla [56] and Pritchard [57] presented the first sub-quadratic algorithms for the extremal sets finding problem that run in $O(m^2/\log m)$ time. These works estimate the asymptotic time complexity of the algorithms but do not provide experimental results. Bayardo and Panda [51] present two practical algorithms, that borrow some concepts from [56, 57], that have good performance on very large families of sets. To obtain efficiency gains, they assume that the sets of the family are sorted sets whose elements are ordered increasingly according to its frequency within the family $\mathcal{F}$. Parallel algorithms on the CREW PRAM and CRCW PRAM models for finding extremal sets, that maps domain elements to bits and exploit bit-parallel operations, have been developed in [58].

In this Chapter we present efficient parallel GPU-based algorithms, designed under CUDA architecture, to find the extremal sets of a family $\mathcal{F}$ of sets. The algorithm presented, has no restrictions on the input family. We explain, first, the algorithm to find maximal sets in detail and then we explain the small changes made to the algorithm so we can find the minimal sets. Finally the complexity analysis of the presented algorithms together with experimental results obtained with their implementations are provided.

## 5.1 FINDING MAXIMAL SETS

Our approach to find the maximal sets of $\mathcal{F}$ is based on efficiently discarding the non maximal sets, that is, the sets $S_i$ of $\mathcal{F}$ such that there exist at least a set $S_j$ of $\mathcal{F}$ with $S_i \subset S_j$.

Fixed a set $S_i$ of $\mathcal{F}$, for each element $e$ of $S_i$ we find the family of the sets $S_j$ of $\mathcal{F}$ that contain the element $e$. If there exist any set $S_j$ common to all these families, it is $S_i \subseteq S_j$. If for at least one such set $S_j$ it is $|S_i| < |S_j|$, then the set $S_i$ is non maximal and can be discarded, otherwise for all this common sets $S_j$ it is $|S_i| = |S_j|$ and the set $S_i$ and all sets $S_j$ are maximal sets and, thus, we

do nothing. Although, in case we want to discard the repeated sets, the algorithm can be slightly modified so the repeated sets of the output are discarded.

### 5.1.1 *Storing the family of sets in the GPU*

Without loss of generality we can assume that the elements of the domain are indexed by non-negative integers and consider $D = \bigcup S_i = \{0, \ldots, n-1\}$.

To increase the efficiency of our algorithm to find maximal sets, the family sets need to be sorted by increasing cardinality order. Since our input does not have this restriction, while we store the family sets in the GPU, we will also reindex the sets according to the increasing order of their cardinalities.

All the 1D-arrays we use in the presented algorithm are stored in GPU global memory.

To represent the family $\mathcal{F}$ in the GPU, we use a data structure that consists of three 1D-arrays, the family, counting and positioning arrays denoted $f$, $c$ and $p$, respectively. Array $f$ stores the $m$ elements defining the family $\mathcal{F}$, starting with the elements of $S_0$ and ending with the elements of $S_{k-1}$. The counting and positioning arrays have size $k$ and store the sets cardinality and the position of $f$ where each set starts, respectively. With this structure we can easily access in a parallel way to any set or element using the fact that the set $S_i$ starts at position $p[i]$ of $f$ and is stored in $c[i]$ consecutive positions of $f$ (see Figure 42).

We start by storing the input family $\mathcal{F}$ in the CPU in a data structure, such as the one used to represent it in the GPU, using three 1D-arrays $c'$, $p'$ and $f'$. The structure is created while the input family is read. Then the cardinalities in $c'$ are sorted with a GPU sort algorithm [47], taking into account that while $c'$ is being sorted $p'$ is accordingly reorganized. When the sort algorithm ends $c'[i] \leqslant c'[i+1]$ for every $i$, and $p'[i]$ indicates the position where the initial set with cardinality $c'[i]$ stars in $f'$. These three arrays are used to build $c$, $p$ and $f$ which store the family structure with the sets in increasing cardinality order in the GPU. To start with, $c'$ is copied to $c$ and a prefix sum of $c$ is performed to obtain $p$ using the parallel GPU scan algorithm [47]. The family array $f$ is obtained by using $c, p, p'$ and $f'$ with a parallel kernel with $k$ threads. The thread with index $idx$ copies the elements of the set $S_{idx}$, in the reorganized family, to $f$. Thread $idx$ reads $c[idx]$ consecutive elements of $f'$ starting at $p'[idx]$ and writes them in consecutive positions of $f$ starting at $p[idx]$ (see Figure 42). When the process ends, $c', p'$ and $f'$ can be deleted from the GPU, and $c$, $p$ and $f$ contain the reorganized family which has $|S_i| \leqslant |S_j|$ whenever $i < j$, as we are interested in.

Once $f$ is reorganized according to the cardinalities, abusing notation we denote by $S_i$ the $i$th set stored in $f$, which in general is not the initial set $S_i$. We have to take into account that in the case that at the end of the process we want to report the indices of the maximal sets with the initial indexing, we should maintain in the CPU an auxiliary array with the set indices correspondence.

### 5.1.2 *Finding the maximal sets in the GPU*

The algorithm for finding maximal sets consists of two main steps. In the first step we determine the sets of $\mathcal{F}$ containing each element of the domain D, and in the second one we find the maximal sets of $\mathcal{F}$.

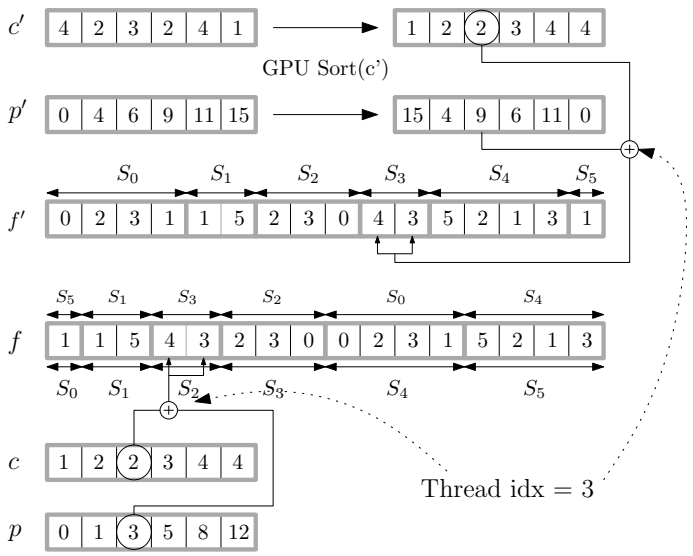



Figure 36: GPU data structure containing a family of six sets. The initial sets are shown in grey and in black the sets after sorting.

*Elements apparition determination*

In this first step we determine the sets in which each element $e \in D$ appears. This information is stored in the so called apparitions vector $A$, which is a vector of $n$ lists of set indices. The list $A[e]$ contains the indices of those sets containing the element $e \in D$, thus a set index $i$ with $0 \leqslant i \leqslant k-1$ is stored in $A[e]$, whenever $e \in S_i$. Notice that the total number of elements stored in $A$ is exactly $m$. An example of such vector is showed in Figure 43, where the element 1 appears in the sets $S_0$, $S_1$, $S_4$ and $S_5$, so the indices 0, 1, 4 and 5 are stored in $A[1]$.

As we will see later, in order to find the maximal sets efficiently, we need that each list of $A$ has the set indices stored in increasing order to then be able to locate a specific index set using a binary search algorithm. To obtain the apparitions vector efficiently using the GPU parallelism we first find $A$ without taking care of the lists order, and next a GPU sort algorithm [47] is used to sort all the lists in parallel.

The apparitions vector $A$ is stored in the GPU using the data structure previously used to store a family of sets. In this case the structure, referred as the apparition structure, consist of three 1D-arrays denoted $a$, $c_A$ and $p_A$. Array $a$, of size $m$, stores the elements of the lists defining $A$, the counting array $c_A$, of size $n$, stores the number of sets containing each element or equivalently the number of elements contained in each list $A[e]$. Finally, the positioning array $p_A$ stores in $p_A[e]$ the position of the array $a$ where the list $A[e]$ starts.

First we compute the array $c_A$, initializing it to 0 and launching a parallel kernel with $m$ threads, one per element in $f$. The thread with index $idx$ reads its corresponding element $e = f[idx]$ and increments $c_A[e]$ by one. In Figure 43, $c_A[2]$ is incremented three times by three different threads. Note that since many threads may modify the same memory position at the same time, the thread race condition can lead to incorrect results. To avoid this we use the atomic operations where memory accesses are done with no thread interferences. The positioning array of $A$, $p_A$, is the prefix sum of $c_A$ and is computed using the GPU scan algorithm.

To store the apparitions vector in the GPU array $a$, we run a CUDA kernel with one thread per element in $f$. The thread with index $idx$ reads its corresponding element in $f$, $e = f[idx]$, determines, by using $p$ and the index $idx$, the set $S_j$ where $e$ belongs to, and stores $j$ in $a$. The set index $j$ is determined by locating the thread index $idx$ in $p$ with a dichotomic search. To store $j$ in $a$ we use an auxiliary array $v$ of size $n$ initialized to zero containing the number of indices already stored in each list of $A$. Thus $j$ is stored as the $v[e]$ element of the list $A[e]$ which corresponds to the position $p_A[e] + v[e]$ of $a$, and consequently when storing $j$ the value of $v[e]$ has to be incremented

by one. Since several threads can be storing set indices in the same list $A[e]$ at the same time, the value of $v[e]$ is obtained and incremented in one by using an atomic operation to avoid collisions.

By handling all the elements in $f$ in parallel, the order in which the index sets are inserted in $A$ is not known in advance and the lists are not sorted. Thus, the obtained lists need to be sorted by increasing index order. To this purpose, we could read the array $a$ back to the CPU, perform $n$ sort algorithms and transfer again the sorted lists to GPU. However this is not an efficient solution, not only because the memory bandwidth is small but also because $n$ sort algorithms handling a total of $m$ elements is a big amount of work.

We will use an already existing and implemented GPU sorting algorithm [47] to sort all the lists. However, this GPU sort algorithm is very efficient sorting very large lists and in this case we have to sort many short lists. In order to take the maximum benefit of the GPU sort algorithm we compute $A_r$, by mapping the indices in each list to new integers guaranteing that the integers stored in $A_r[i]$ are smaller than all the ones in $A_r[j]$ whenever $i < j$. To use not too big integers in $A_r$ we use two arrays $A_{min}$ and $A_{max}$, with the minimum and maximum set index stored in each list of $A$, they are obtained while $A$ is computed using atomic operations. Next we compute $A_{dif}$, with $n$ threads, which stores the differences between $A_{max}$ and $A_{min}$. These differences are accumulated by using the exclusive scan algorithm, the result is again stored in $A_{dif}$. Finally $A_r$ is computed, by adding to each index of $A[i]$ the value $A_{dif}[i] - A_{min}[i]$. Now we consider $A_r$ as a unique list, by concatenating the different lists and $A_r$ is sorted with a GPU sort algorithm. Finally the set indexes are recovered by substracting the previously added value. Thus finally, we have the apparitions vector $A$ with the indices of the lists $A[e]$, $e \in D$, sorted by increasing order. An example of the process is shown in Figure 43.

Having the apparitions vector built with the lists of set indices sorted in increasing order, that corresponds to increasing set cardinality, we start the second step where the maximal sets are found.

*Maximal sets determination*

In this second step we actually discard the non maximal sets of $\mathcal{F}$ by working with the apparitions vector $A$ and the counting array $c$ that stores the sets cardinality.

We associate an apparitions vector $A_i$ to each set $S_i$ of $\mathcal{F}$. Given an element $e \in S_i$, $A_i[e]$ is the ordered sublist of $A[e]$ which stores the indices $j \in A[e]$ with $i \leqslant j$. They correspond to the sets $S_j$ of $\mathcal{F}$ that contain the element $e$, with $c[i] \leqslant c[j]$. Thus $A_i[e]$ is the ordered list of the indices of the sets that may contain $S_i$ (see Figure 38). Consequently, whenever an index $j$ is not contained in $A_i[e]$ for some $e \in S_i$, then $S_i \not\subseteq S_j$, thus $S_i$ can still be maximal and we cannot discard it. On the contrary, if the set index $j$ is present in all $A_i[e], e \in S_i$, then $S_i \subseteq S_j$. If $c[i] < c[j]$ for at least one such set $S_j$, then $S_i \subset S_j$ and consequently $S_i$ is not maximal and can be discarded, otherwise if $|S_j| = |S_i|$ set $S_j$ is equal to $S_i$, and thus they may still be maximal.

Denote $\tilde{e}$ an element of $S_i$ that determines a list $A_i[\tilde{e}]$ of minimal cardinality between the lists of $A_i$. In order to discard $S_i$ for not being maximal, we search for the indices of $A_i[\tilde{e}]$ that are present in all $A_i[e]$, $e \in S_i$ and $e \neq \tilde{e}$. If there exist at least one such index $j$ and also it is $c[i] < c[j]$, then $S_i$ is discarded. In Figure 38 we can see an example. Next we describe how this process can be computed in parallel.

Since allocating and constructing the apparitions vector $A_i$ associated to each $S_i$ is very expensive in time and in memory, we enlarge the structure of the apparitions vector $A$ so that we can obtain the information provided by all the $A_i$ without explicitly constructing them.

First, to determine for each set $S_i$ the list $A_i[\tilde{e}]$ we construct two arrays $\tilde{c}$ and $\tilde{s}$ of size $k$. We initialize all the values of $\tilde{c}$ to $k+1$. In $\tilde{c}$ we will store $|A_i[\tilde{e}]|$ and in $\tilde{s}$ the position where $A_i[\tilde{e}]$ starts within $A[e]$. This is done by launching a kernel with $m$ threads, one per element in $f$. Each thread $idx$ reads the $e$ element $f[idx]$ and computes its set index $i$ by performing a dichotomic search within $p$. Then the position $pos$ of $i$ within $A[e]$ is determined using a dichotomic search within $A[p_A[e]]$ and $A[p_A[e] + c_A[e]]$. Thus, set indices stored in $A[e]$ above $pos$ are the indices of the sets containing $e$ with a cardinality bigger or equal to $|S_i|$, so $|A_i[e]|$ is computed as $c_A[e] + p_A[e] - pos$. In the case that $|A_i[e]| < \tilde{c}[i]$ then we set $\tilde{c}[i] = |A_i[e]|$ and $\tilde{s}[i] = idx$ by using atomic operations in

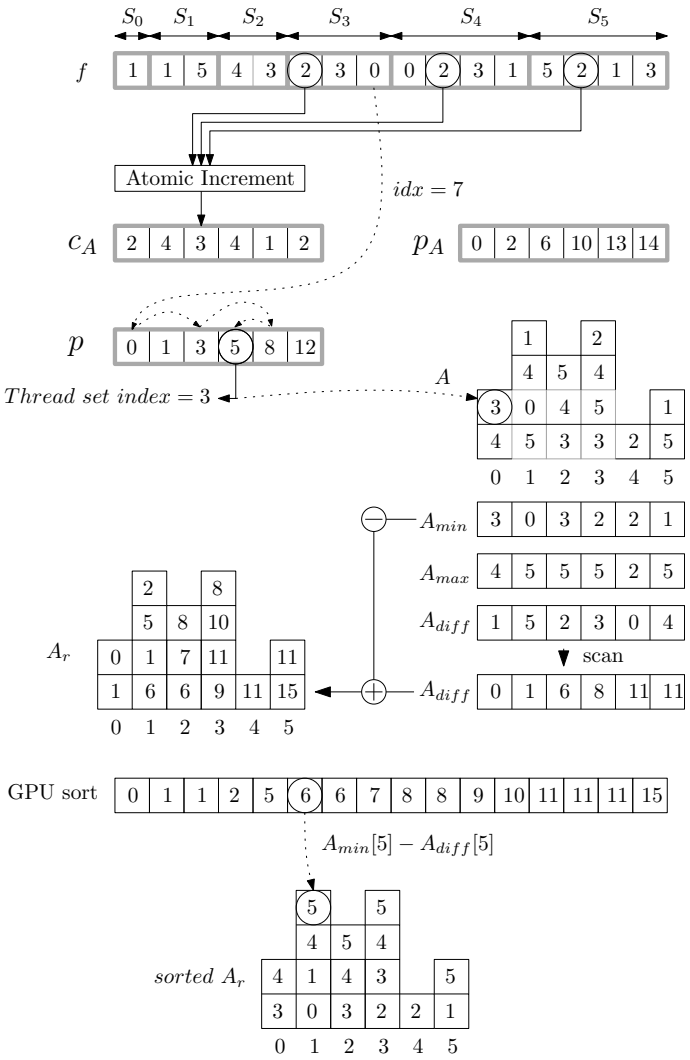



Figure 37: The construction of the apparitions vector $A$.

order to avoid thread conflicts. When we are done, $\tilde{c}[i]$ is the number of sets that are candidates to contain $S_i$.

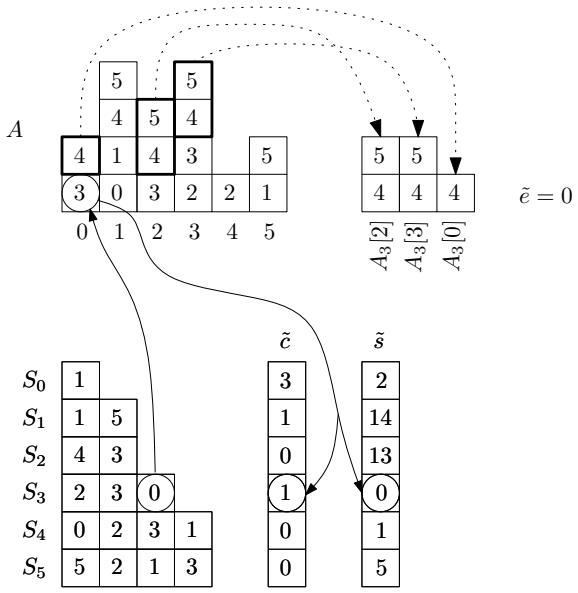



Figure 38: Lists of $A_3$ and arrays $\tilde{c}$ and $\tilde{s}$ of a family of sets.

Then we create $\tilde{p}$ as a prefix sum of $\tilde{c}$ using the parallel GPU scan algorithm [47], and we launch a kernel with $\tilde{p}[k-1] + \tilde{c}[k-1]$ threads. Note that we launch as many threads as possible candidates to contain another set. Each thread $idx$ finds the pair $\{S_i, S_j\}$ with $i < j$ it corresponds to, and next determines wether $S_i \subset S_j$ by identifying the set index $i$ performing a dichotomic search over $\tilde{p}$, and the index $j$ as $A[\tilde{s}[i] + idx - \tilde{p}[i]]$. Now, for all elements $e$ in $S_i$, the thread checks if the set index $j$ is present in $A[e]$. If it is so, and $c[i] < c[j]$, then $S_i \subset S_j$ and consequently $S_i$ is discarded by setting $c[i]$ to 0. When an element index $j$ is not present in some list, $S_j$ does not contain the element $e$, and the thread stops without discarding $S_i$ because it is not contained in $S_j$ and thus $S_i$ can still be maximal.

Observe that during the process several threads may be checking in parallel if the same set $S_i$ is contained in different other sets $S_j$ with $j > i$. It can occur that a thread discards the set $S_i$ for being contained in a thread $S_j$, while some other threads are still checking if another $S_{j'}$ contains $S_i$. Ones $S_i$ has already been discarded the other threads are wasting their work. Thus, each thread before analyzing a new element of the list $S_i$ checks whether $S_i$ has already been discarded. If it is so, the thread stops avoiding unnecessary computations.

Note that because the index elements of the list $A_i[e]$ are sorted in the increasing order of cardinalities of their corresponding sets, the location of an index in the list can be done by a dichotomic search, avoiding traversing the whole list.

At the end of this step, we have stored in the element $c[i]$ of the counting array $c$ the cardinality of $S_i$ if $S_i$ is a maximal set and a zero otherwise. With this information we can report the maximal sets as we explain in Section 5.4.

## 5.2 FINDING MINIMAL SETS

The algorithm for finding the minimal sets of $\mathcal{F}$ is similar to the one used for finding the maximal sets. The apparitions vector computation needs no changes. During the determination of the minimal sets, only the discarding condition is modified. When a thread checks that all elements

in $S_i$ are also present in $S_j$ and $c_i < c_j$, the set $S_j$ is discarded for containing strictly $S_i$ by setting $c[j] = 0$. At the end of the process, all sets $S_j$ with $c[j] \neq 0$ are the minimal sets of the family $\mathcal{F}$.

## 5.3 COMPLEXITY ANALYSIS

When we analyze the complexity of a GPU algorithm we should take into account the total work, the thread work, the accesses to memory and the transferred values between CPU and GPU. The total work is the total number of instructions realized during the whole algorithm. The thread work is the number of operation done by a single thread. The thread work gives an idea of the degree of parallelism obtained. Despite the parallelization does not decrease the total work complexity of the algorithm it has an important effect on the running times. Finally a GPU algorithms performance also depends on the number of memory accesses and on the transferred values from CPU to GPU and viceversa.

Next we analyze the complexity of the main steps of our approach separately.

*Family of sets storage.* This step is performed in the CPU. Reading the input family $\mathcal{F}$, and sorting the sets by its cardinality takes $O(m + k \log k)$. To store $\mathcal{F}$ in the GPU we use $m + 2k$ integer values, which are transferred from the CPU to the GPU.

*Apparitions vector computation.* To compute the apparitions vector $A$ we start initializing $c_A$ to zero representing a total work of $O(n)$ done with $n$ threads in $O(1)$ per thread work. The computation of $c_A$ requires $O(m)$ total work executed by $m$ threads doing $O(1)$ work each. To compute the unsorted $A$ we use $m$ threads with a thread work of $O(\log k)$, which is needed to determine the set each thread represents, thus the total work is $O(m \log k)$. To store $A$ we use $m + 2n$ integers, we also need $m + 2k$ integers to store $\mathcal{F}$ and $n$ extra integer values to compute the unsorted $A$. While computing $A$ we also use $2n$ extra integer values to store the minimum and maximum value of each list in $A$.

To obtain $A_r$ we compute $A_{dif}$ using $n$ threads doing $O(1)$ work each, next the differences are accumulated with a scan algorithm. When $n$ values are considered, the Nvidia scan algorithm [59] uses balanced trees with $d = \log^2 n$ levels and $2^d$ nodes per level, providing a total work of $O(n)$ done by $n$ threads providing $O(1)$ work per thread. Next, the integers of $A_r$ are computed by using $m$ threads doing $O(\log k)$ work each, representing $O(m \log k)$ total work. The resulting array is sorted using a GPU sort algorithm which sorts the $m$ elements in $O(m \log m)$ total work using $m$ threads giving a $O(\log m)$ work per thread. Finally the sets indices are reobtained with again $m$ threads doing $O(\log k)$ work each, representing $O(m \log k)$ total work. In all this process only $n$ extra integer values are needed to store $A_{dif}$.

*Extremal sets determination.* To obtain the extremal sets we start by initializing a vector of $k$ elements with a total work of $O(k)$ doing $O(1)$ work per thread. Next $\tilde{c}$ and $\tilde{s}$ are computed by using $m$ threads, whose hardest work is determining the set $S_i$ each tread corresponds to taking $O(\log k)$ work. Thus the total work is $O(m \log k)$, and $2k$ integer values are used to store $\tilde{c}$ and $\tilde{s}$. Next the $k$ elements of $\tilde{p}$ are computed by using again the standard scan algorithm, representing a total work of $O(k)$ done by $k$ threads providing $O(1)$ work per thread. Finally the non extremal sets are discarded by using $O(nk^2)$ threads, where each thread represents a pair of sets $\{S_i, S_j\}$. Finding the pair a thread corresponds to takes $O(\log k)$ work. Then to determine wether $S_i \subset S_j$ the required work is $O(n \log k)$. It represents a total work of $O(n^2 k^2 \log k)$.

## 5.4 REPORTING EXTREMAL SETS

Extremal sets can be reported in different ways, some applications may need to report just their indices and some others may need the family of extremal sets.

*Obtaining the indices of the extremal sets*

Since we know that the indices $i$ of the extremal sets are those such $c[i] \neq 0$, we just have to resize $c$ to the number of extremal sets we have found. This can be done in the GPU, but it requires several kernels and steps in order to count, resize and report the array. So, we read $c$ to the CPU memory and then we count the number of positions in $c$ whose value is different from $0$, create the final array $c_{ext}$ and report them in a sequential way. Remember from Section 5.1.1, that the indices reported are the indices of $\mathcal{F}$ once it has been sorted. If we want the original indices we just need to use the auxiliary array containing the relation between the original and the new indices that we created when we sorted $\mathcal{F}$.

*Obtaining the family of extremal sets*

Alternatively, we can report the family $\mathcal{F}_{ext}$ of extremal sets. To represent $\mathcal{F}_{ext}$ we use the already used data structure consisting of three 1D arrays, $c_{ext}$, $p_{ext}$ and $f_{ext}$. We compute this data structure in the GPU, because doing it in the CPU represents transferring and traversing all the family $\mathcal{F}$ which is expensive, thus we proceed as follows.

To determine the number $k_{ext}$ of maximal sets we launch a kernel with $k$ threads, one per set of $\mathcal{F}$. Each thread $idx$ increments a variable $k_{ext}$ by one whenever $c[idx] \neq 0$, it is done with an atomic function to guarantee correct results. When all threads are done, $k_{ext}$ contains the number of extremal sets.

Next, three arrays of size $k_{ext}$, $c_{ext}$, $p_{ext}$ and an auxiliary one $p'_{ext}$, are allocated in the GPU memory. Next we launch a kernel with $k$ threads so that each thread with index $idx$ having $c[idx] \neq 0$ copies $c[idx]$ to the first empty position of $c_{ext}$, and $p[idx]$ to the same position of $p'_{ext}$. To know, at each moment, the first empty position of $c_{ext}$, we use an auxiliary variable which is read and incremented, by using an atomic operation when a new value has to be stored in $c_{ext}$.

To obtain $f_{ext}$ we compute $p_{ext}$ as the prefix sum of $c_{ext}$. Then, once the total number of elements conforming the family of extremal sets is known, we allocate $f_{ext}$ in the GPU, and finally the corresponding sets of $\mathcal{F}$, which start at the positions stored in $p'_{ext}$, are copied to $f_{ext}$ by using $p_{ext}$ and $c_{ext}$ and a kernel with as many threads as elements are contained in $f_{ext}$. To end with, we read $\mathcal{F}_{ext}$ to CPU and report it. Notice that, in this case, the order in which the extremal sets are reported depends on the threads execution and does not coincide neither with the original one nor with that according to the sets cardinality.

## 5.5 RESULTS

In this section we present a wide analysis of the algorithm studying its behavior under several situations for the case of finding maximal sets. We provide the running times to obtain the maximal sets for synthetic and real data sets. Additionally, we compare our algorithm against the algorithm to find maximal sets provided by Bayardo et al. [51].

We have developed a tool to generate synthetic families. The generator allows us to change some parameters related to the characteristics of these synthetic families so we can check not only our algorithm goodness, but the algorithm response depending on the input and output families. For instance we want to know how does our algorithm run whenever we increase the number of maximal sets while the number of sets remains constant. We can select the number of maximal sets, the number $k$ of sets, the total number $m$ of elements of the family and the number $n$ of elements of the domain. We have generated the families using the pseudorandom generator method of Mersenne twister [60] which has a period of $[0, 2^{19937} - 1]$. The tests have been done in a i5 – 200 3.30 GHz CPU with 8 GB of memory and a Nvidia GTX 580. All the reported running times have been measured as the median of 10 executions per test.

Figure 39 shows the running times for the maximal sets algorithm. The main steps of the algorithm, and also the total time are shown, so we can see how they are affected by the variation of the input and output parameters. The considered steps are: 1) the load of the family from disk to memory and the initial sort of the family sets by cardinality, 2) the construction of the apparitions vector, 3) the maximal sets determination and 4) the reporting of the maximal sets. All the running

times provided in the experiments include the CPU to GPU and the GPU to CPU memory transfer times and have been measured after all GPU routines have finished.

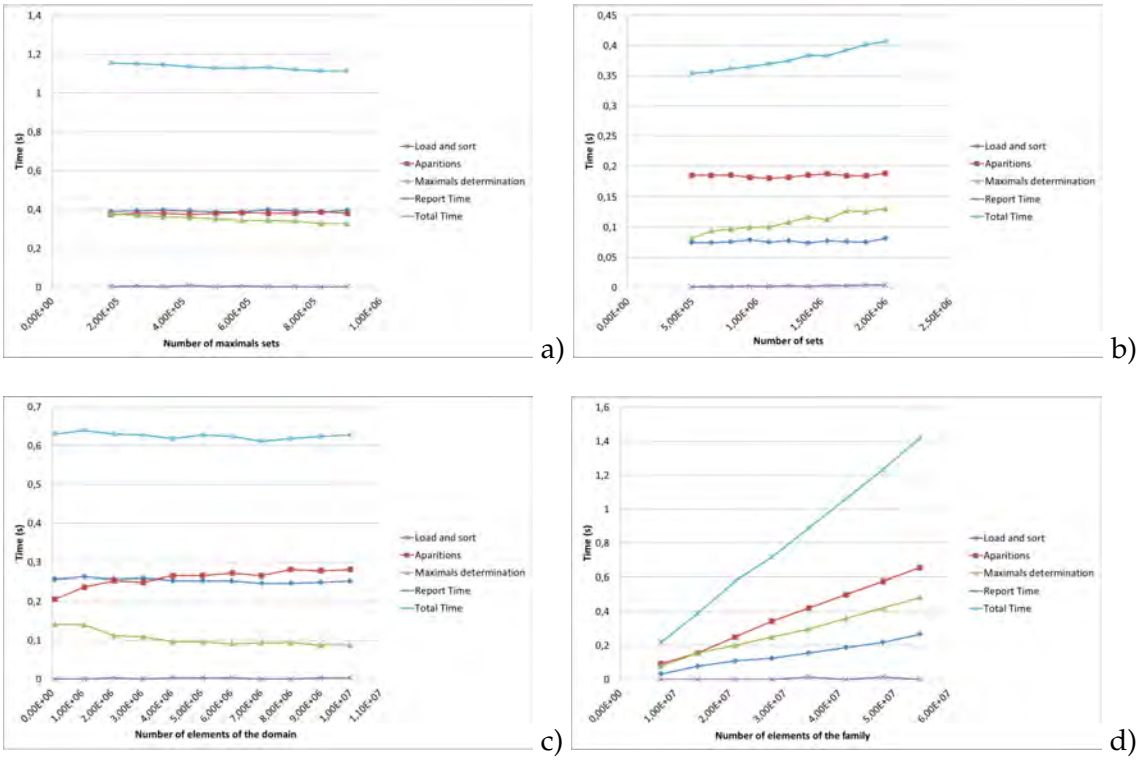



Figure 39: Total running times for finding maximal sets depending on the characteristics of the families.

Next we analyze the variation of the running times when the input/output parameters change:

*Varying the number of maximal sets*

In Figure 39 a) the number of maximal sets varies from $10^5$ to $9 \times 10^5$, for a fixed family of $k = 2 \times 10^6$ sets, with a median of 40 elements per set and considering a domain of $n = 10^6$ elements. The running time behavior of the algorithm is similar in all tests. The execution time is practically not affected by the number of maximal sets of the input family. The load of the family to memory and the initial sort of the family sets by cardinality as well as the apparitions vector computation times remain constant, which is normal since the processes mainly depend on the number of elements of the input family which is the same in all cases.

We observe a minimal variation of the running time needed for determining the maximal sets, which can be obtained faster when we increase the number of maximal sets. This is because the way we decide whether a set has to be discarded. A thread checking if the set $S_i$ is contained in the set $S_j$ stops whenever finds an element of $S_i$ not present in $S_j$. As long as we increase the number of maximal sets is more likely to have elements not present in the potential maximal sets and consequently more likely a thread stops without discarding the set. Thus, many threads finish their job earlier and consequently the algorithm runs faster.

*Varying the number of sets of the family*

In Figure 39 b) the number $k$ of sets goes from $5 \times 10^5$ to $2 \times 10^6$, for a domain of $n = 10^6$ elements, $4 \times 10^5$ maximal sets with a median of 40 elements per set. Again the load of the family to memory, the initial sort of the family sets by cardinality and the apparitions vector computation times remain constant, while the maximal sets determination time has a slight variation when we increase the number of sets. Contrary to what happens when we increase the number of maximal

sets, now the number of potential maximal sets increases while the number of maximal sets remains constant. Thus, threads have the same amount of work on each test but we have to lunch more threads as long as we increase the number of sets. Consequently the time needed to determine the maximal sets increases.

*Varying the number of elements of the domain*

In Figure 39 c) the number of elements of the domain goes from $10^5$ to $10^7$, for $k = 10^6$ sets, $7 \times 10^5$ maximal sets with a median of 40 elements per set. In this case, despite the running time is practically constant, we can observe that the time increases for the apparitions vector computation and decreases for the maximal sets determination. When we increase the number of elements of the domain it is more likely to have empty lists in the apparitions vector. Despite is does not matter when we report it, it does when we sort the lists of the apparitions vector. On the other hand, because the elements are more distributed, the number of potential maximal sets decreases and consequently the time needed for determining the maximal sets also decreases. Note that in both cases the time variation between different tests are really small.

*Varying the number of elements of the family*

Finally, in Figure 39 d) the number of elements varies from $7.5 \times 10^6$ to $55 \times 10^6$, having $1.5 \times 10^6$ sets within a domain of $n = 10^6$ elements. As we expected our algorithm efficiency strongly depends on the number of elements of the family. This is because all the kernels in our algorithm are element based. The apparitions vector is computed by considering all the elements at the same time. Thus, the time for computing it directly depends on the number of elements. This makes sense because is the most parallelizable, and consequently more efficient, way to compute it. The time needed to sort the lists of the apparitions vector also depends on the number of elements of the family. The same happens when we discard the non maximal sets. We consider a thread per element in each list against the potential maximal sets. Again, that makes that the running time of the algorithm depends on the number of elements. Note that the behavior observed in Figure 39 d) is linear in the number of elements of the family.

*Finding minimal sets*

We also have tested the algorithm for finding minimal sets using random families with $10^6$ sets. The results, running time and behavior, obtained with the experiments are very similar to the ones obtained with the experiments we have done with the algorithm for finding maximal sets.

*Algorithm comparison*

We compare our GPU algorithm against the CPU algorithms presented by Bayardo et al. in [51]. We use the source code provided by the authors. In the Chapter, they presented two approaches, AMS-card which exploits the cardinality property of the sets and AMS-lex which exploits the information provided when the input family is lexicographically sorted.

The algorithm presented in [51] assumes that the input families have some specific characteristics. First, the elements within each family set must be sorted in increasing order according the ordering of the domain. Second, the sets of the family must be sorted by increasing cardinality order for the AMS-card algorithm and by lexicographical order for the AMS-lex algorithm. Additionally, the authors remark that, due the nature of their approach, when the elements within each set of the input family are sorted by frequency of apparition within the whole family, the algorithms show a remarkable improvement in their performance. Thus, in order an input family fits to their algorithm needs, arbitrary families require to be preprocessed. The preprocess is done as follows. First we read the $m$ elements of the sets of the input family to determine the frequency of apparition of each element. Second, we reindex the domain elements according to its frequency, that is, element $0$ will now now denote the less frequent element in the family and element $n-1$ will denote the most frequent element in the family. Then, we reindex the elements of the sets of the family according to the new reindexed domain. Third we sort the reindexed elements of each

set in increasing order. Finally we sort the sets within the family in increasing cardinality for the AMS-car algorithm and in lexicographical order for the AMS-lex algorithm.

Moreover, the algorithms presented in [51] work on disk while our algorithm works in CPU and GPU memory. Thus, our algorithm cannot handle families as big as they can process. However, to work on disk implies that after performing the necessary preprocess applied to the input families, the result of the preprocess have to be written back to disk in order the algorithms AMS-card and AMS-lex work. In the results, the time needed to write to disk is also included in the preprocess step.

Because our algorithm does not have restrictions on the input families and does not work on disk, we compare the total running times of both algorithms including the preprocess. However, we provide the time of each algorithm for the specific process of finding the maximal sets.

In Figures 40 we compare the total running times of the algorithms using the same families of the experiments showed in Figures 39. In all cases we compare our GPU algorithm against their best algorithm, that is, when the elements of the family are sorted by frequency of occurrence. In Figures 40 a), c) and e) we compare our GPU algorithms with the AMS-Card algorithm and b), d), and f) with the AMS-lex algorithm. We can see that in all cases our GPU algorithm is faster than AMS-card and AMS-lex algorithms. We can observe how the input parameters have a clear influence in AMS-Card and AMS-lex, while our algorithm shows, mostly, a constant behavior. Figures 40 e) and f) show that the running time of the GPU algorithm increases as long as we increase the number of elements of the family, but AMS-Card and AMS-lex algorithms running times increase faster.

We also compared the algorithms when the number of elements of the domain varies and, although the GPU algorithm was again faster, the results did not give any relevant information.

Additionally we also tested the algorithms with a public data family which has been also used in [51] (Figure 41). The family consists in almost 1 million sets and has 14 elements per set on average. In this family the elements are not uniformly distributed. We can observe that the GPU algorithm runs up to 6 times faster than Bayardo et al. algorithms. Note that even excluding the needed preprocess from the running times of the Bayardo et al. algorithm from the total running time, the GPU approach is still faster.

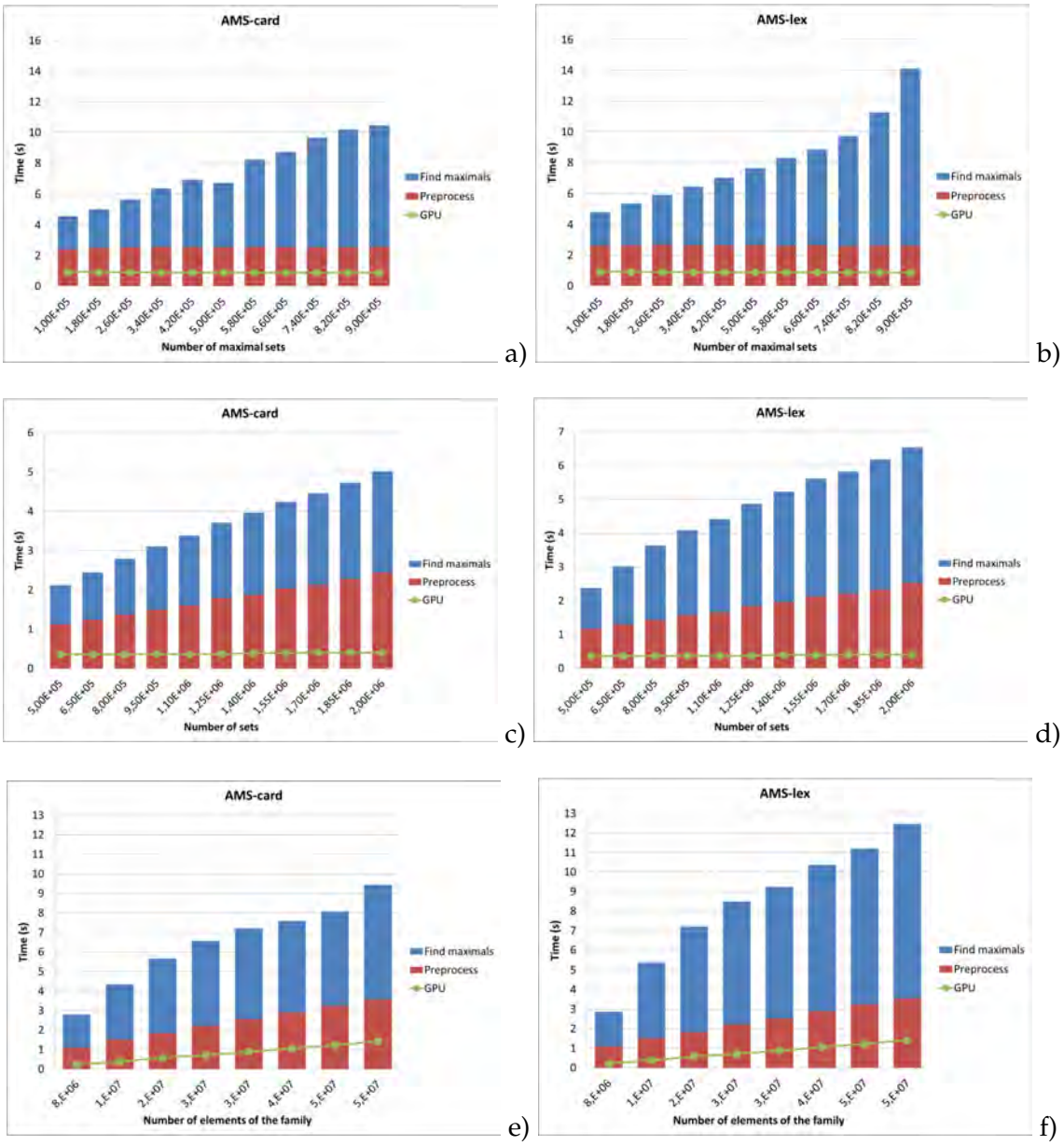



Figure 40: Comparison of our GPU algorithm against Bayardo et al. algorithms using synthetic data sets.

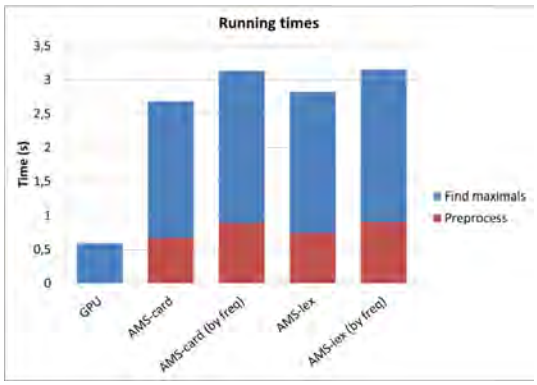



Figure 41: Comparison of our GPU algorithm against Bayardo et al. algorithm using a real data set.

# 6

# INTERSECTION SETS

*9 women cannot make a baby in one month*
***– Fred Brooks***

In this Chapter we face the problem of intersecting two families of sets. Let $\mathcal{F} = \{S_0, \cdots, S_{k-1}\}$ and $\mathcal{F}' = \{S'_0, \cdots, S'_{k'-1}\}$ be two families of non-empty finite sets over a domain D, where no set appears more than once in a family. The problem of intersecting the families $\mathcal{F}$ and $\mathcal{F}'$ is to find the family $\mathcal{I}$ of all non-empty sets, which are the intersection $S_i \cap S'_j$ of a set $S_i$ of $\mathcal{F}$ and a set $S'_j$ of $\mathcal{F}'$. Notice that, although the intersection of two different pairs of sets, each pair formed by a set of $\mathcal{F}$ and another of $\mathcal{F}'$, can provide the same intersection set, we only include it once in the family $\mathcal{I}$. Although our approach works with families with no restrictions in the input or output, due to practical purposes, we consider that the input and output families do not contain duplicated nor empty sets. However, by using the algorithms presented in Sections 6.1.5 and 6.1.6 our approach can be easily adapted so it handles input and output families with repeated or empty sets.

The resolution of several computational problems depends on an efficient algorithm for finding set intersections. Next, we present two application examples where two families of sets need to be intersected. The first example is related to the detection of movement patterns among trajectory databases. The second one falls in the area of web search engine systems and document data mining.

*Example 1*. A flock is defined as a large subset of entities moving together in a disc of fixed radius during several consecutive time-stamps. Vieira et al. propose in [55] an algorithm for reporting flocks that first computes subsets of potential flocks in two consecutive time-stamps and then joins (intersects) all the possible subsets of the first time-stamp with those of the second one.

*Example 2*. Assume that we are given a family of web pages crawled by a search engine or textual documents stored in a database. Each web page/document has assigned a unique positive integer identifier and is represented as a set of words. Every word has also assigned a unique positive integer identifier. A query is a set of important words corresponding to one class of a text categorization. We construct a family of queries with different classes of text categorization. A multi-intersection query finds all the sets which are the intersection of some set of the family of queries and some set of the family of web pages/documents. Since the number of intersection sets can be large, we should be interested in reporting only the sets with a minimum cardinality. Multi-intersection queries find applications, for example, in classification and recommendation systems.

As related work, we mention the maximal intersection query which finds a member of a given family of sets that has the maximal intersection with a single query set. Algorithms for solving maximal intersection queries are provided in [61]. The problem of computing the intersection of sorted lists has received significant research attention, theoretically [62, 63, 64, 65], and experimentally [66, 67, 68]. Finally, algorithms for computing the intersection of two or more sets on the Graphics Processing Unit (GPU) also exist [69, 70, 71, 72].

In this Chapter, we present an efficient parallel GPU-based approach, designed under CUDA architecture, for computing the intersection of two families of sets. To the best of our knowledge, this is the first time that the problem is addressed, either using the GPU or the CPU. The complexity analysis of the presented algorithm together with experimental results obtained with its implementation, showing the efficiency and scalability of the approach, are provided.

The Chapter is structured as follows. In Section 6.1, we explained the approach for reporting the families intersection sets. Finally, the complexity analysis and the experimental results, obtained with our algorithms implementation, are given and discussed in Sections 6.2 and 6.3, respectively.

## 6.1 REPORTING THE INTERSECTION FAMILY

The computation of the intersection family $\mathcal{I}$ of two input families $\mathcal{F}$ and $\mathcal{F}'$, whose sets elements do not need to be sorted in any specific order, is performed in two steps. First, we reduce $\mathcal{F}$ and $\mathcal{F}'$ by discarding the domain elements that do not appear in both families (Section 6.1.2). Then, we compute the intersections between the two reduced families (Section 6.1.4), by using two main structures: the GPU family structure (Section 6.1.1) and the apparitions vector structure (Section 6.1.3). Next, we can eliminate the empty sets (Section 6.1.5) and duplicates (Section 6.1.6) before reporting the intersection family $\mathcal{I}$ (Section 6.1.8). We also explain, how, in the case that we had not enough GPU memory, the output is obtained by parts (Section 6.1.7).

### 6.1.1 *Storing the input families on the GPU*

Without loss of generality we can assume that the elements of the domain are indexed by $n$ non-negative integers and consider $D = \bigcup S_i = \{0, \ldots, n-1\}$.

All the 1D-arrays we use in the presented algorithm are stored in GPU global memory.

To represent a family $\mathcal{F}$ in the GPU, we use a data structure that consists of three 1D-arrays, the family, counting and positioning arrays denoted $f$, $c$ and $p$, respectively. Array $f$ stores the $m$ elements contained in the $k$ sets of the family, starting with the elements of $S_0$ and ending with the elements of $S_{k-1}$. The counting and positioning arrays have size $k$ and store the sets cardinality and the position of $f$ where each set starts, respectively. With this structure we can easily access in a parallel way to any set or element using the fact that the set $S_i$ starts at position $p[i]$ of $f$ and is stored in $c[i]$ consecutive positions of $f$. An example is shown in Figure 42 where the elements of $S_2$ start at $f[6]$ and are stored in 3 consecutive positions.

We create one structure for the input family $\mathcal{F}$ and another for the input family $\mathcal{F}'$. We generate the arrays $c$, $p$ and $f$ representing $\mathcal{F}$ and $c', p'$ and $f'$ representing $\mathcal{F}'$ in the CPU while the input families are read, and then we transfer them to GPU memory.

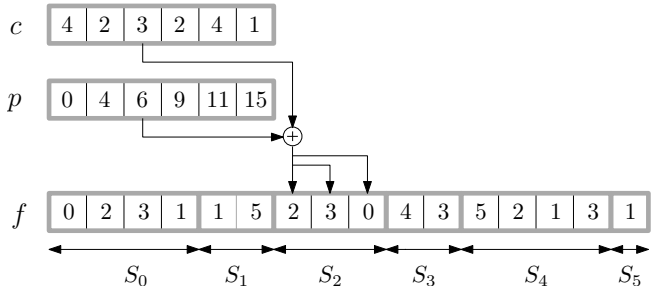



Figure 42: GPU data structure containing a family of six sets

### 6.1.2 *Reducing the input families*

To speed-up the computation of the intersection of $\mathcal{F}$ with $\mathcal{F}'$, we start with the following reduction process. First, we determine the domain $\overline{D} = (\bigcup S_i) \cap (\bigcup S'_j)$. Next, we find the new sets $\overline{S}_i = S_i \cap \overline{D}$ and $\overline{S}'_j = S'_j \cap \overline{D}$, for each set $S_i$ of $\mathcal{F}$ and $S'_j$ of $\mathcal{F}'$. The families determined by the sets $\overline{S}_i$ and $\overline{S}'_j$ are denoted $\overline{\mathcal{F}}$ and $\overline{\mathcal{F}}'$, respectively. Since $\overline{S}_i \cap \overline{S}'_j = S_i \cap S'_j$, instead of intersecting the family $\mathcal{F}$ with

$\mathcal{F}'$ we intersect $\overline{\mathcal{F}}$ with $\overline{\mathcal{F}}'$. However, before starting the intersection computation, the empty and duplicated sets within each family are removed.

To determine $\overline{D}$ we create a 1D array $\overline{d}$ of size $n$ initialized to zero. A parallel kernel is launched with one thread per element in $\mathcal{F}$. Each thread idx sets to one the position f[idx] of $\overline{d}$. Then, we do the same with $\mathcal{F}'$ but setting to two those positions which already have a one. When we are done, $\overline{d}[e]$ contains a two value, whenever the element $e$ is present at least once in each family. Finally, $\overline{d}$ is rewritten by setting those positions with a two to one, and the rest to zero. Additionally, an array $\overline{d}_{off}$ is created as the prefix sum of $\overline{d}$. It is used to maintain the correspondence between D and $\overline{D}$.

In the next step, we remove from $\mathcal{F}$ and $\mathcal{F}'$ all the elements $e$ with $\overline{d}[e] = 0$. To do this, we launch a kernel with one thread per element in $\mathcal{F}$. Each thread idx checks whether $\overline{d}$[f[idx]] is zero. In such a case, the thread determines the set the element idx belongs to by locating idx in p using a dichotomic search. Then, using atomic operations, c[i] is decremented in 1 indicating that the set $S_i$ has one less element and f[idx] is set to $-1$. Additionally, we have a counter to determine the number of completely eliminated sets which is incremented when the already decremented c[i] has been set to zero.

Then, we can allocate $\overline{c}$, $\overline{p}$ according to the new sizes. The array $\overline{c}$ is filled, in parallel, with the non-zero elements of c, by using k threads. Because it is done in parallel the order of the resulting values in $\overline{c}$ does not correspond to the order in c. Thus, we maintain an auxiliary array, $c_c$, storing the correspondence between the positions of $\overline{c}$ and c. Then, $\overline{p}$ is computed as the prefix sum of $\overline{c}$, the new size of the family $\overline{m}$ is determined and $\overline{f}$ allocated. Finally, $\overline{f}$ is filled, in parallel, with $m$ threads. Using $\overline{c}$, $\overline{p}$, c, p, $c_c$ and f we discard the $-1$ elements of f while we copy the others to $\overline{f}$. Additionally, while we store $\overline{f}$, each element $e$ is shifted by subtracting $\overline{d}_{off}$ to $e$, in order to keep on having as new domain $\overline{D} = \{0, \ldots, \overline{n} - 1\}$. Finally, $\overline{f}$ stores the elements of $\overline{\mathcal{F}}$.

This process is performed on the input families $\mathcal{F}$ and $\mathcal{F}'$. When we are done, we can guarantee that all the elements of $\overline{D}$ are present, at least once in both families. Finally, we remove the empty and duplicated sets in both $\overline{\mathcal{F}}$ and $\overline{\mathcal{F}}'$, by adapting the algorithm explained in Sections 6.1.5 and 6.1.6, respectively.

Abusing notation, from now on the new sets and the corresponding families are still denoted $S_i$, $S'_j$ and $\mathcal{F}$, $\mathcal{F}'$, respectively, the elements domain D; k and k′ denote the number of sets in $\mathcal{F}$ and $\mathcal{F}'$, and $n$ the number of elements in the reduced domain.

### 6.1.3 *Computing the apparitions vector*

Given a family $\mathcal{F}$ with k sets, the apparitions vector is a structure containing $n$ lists of set indices determining the sets where each element $e \in D$ appears. The list $A[e]$ contains the indices of those sets containing the element $e \in D$, thus a set index $i$ with $0 \leqslant i \leqslant k-1$ is stored in $A[e]$, whenever $e \in S_i$. Notice that the total number of elements stored in $A$ is exactly $m$. An example of such vector is showed in Figure 43, where the element 1 appears in the sets $S_0$, $S_1$, $S_4$ and $S_5$, so the indices 0, 1, 4 and 5 are stored in $A[1]$.

The apparitions vector $A$ is stored in the GPU using the data structure previously used to store a family of sets. In this case the structure, referred as the apparitions vector, consist of three 1D-arrays denoted $a$, $c_A$ and $p_A$. Array $a$, of size $m$, stores the elements of the lists conforming $A$, the counting array $c_A$, of size $n$, stores the number of sets containing each element or equivalently the number of elements contained in each list $A[e]$. Finally, the positioning array $p_A$ stores in $p_A[e]$ the position of the array $a$ where the list $A[e]$ starts.

First, we compute the array $c_A$, initializing its elements to zero and launching a parallel kernel with $n$ threads. Next, it is obtained by using $m$ threads, one per element in f. The thread with index idx reads its corresponding element $e$ = f[idx] and increments $c_A[e]$ by one. In Figure 43, $c_A[2]$ is incremented three times by three different threads. Note that, since many threads may modify the same memory position at the same time, the thread race condition can lead to incorrect results. To avoid this we use the atomic operations where memory accesses are done with no thread

interferences. The positioning array of $A$, $p_A$, is the prefix sum of $c_A$ and is computed using the GPU scan algorithm.

To store the apparitions vector in the GPU array $a$, we run a CUDA kernel with one thread per element in $f$. The thread with index $idx$ reads its corresponding element in $f$, $e = f[idx]$, determines, by using $p$ and the index $idx$, the set $S_j$ where $e$ belongs to, and stores $j$ in $a$. The set index $j$ is determined by locating the thread index $idx$ in $p$ with a dichotomic search. To store $j$ in $a$ we use an auxiliary array $v$ of size $n$ initialized to zero containing the number of indices already stored in each list of $A$. Thus, $j$ is stored as the $v[e]$ element of the list $A[e]$ which corresponds to the position $p_A[e] + v[e]$ of $a$, and consequently when storing $j$ the value of $v[e]$ has to be incremented by one. Since several threads can be storing set indices in the same list $A[e]$ at the same time, the value of $v[e]$ is obtained and incremented by one by using an atomic operation to avoid collisions.

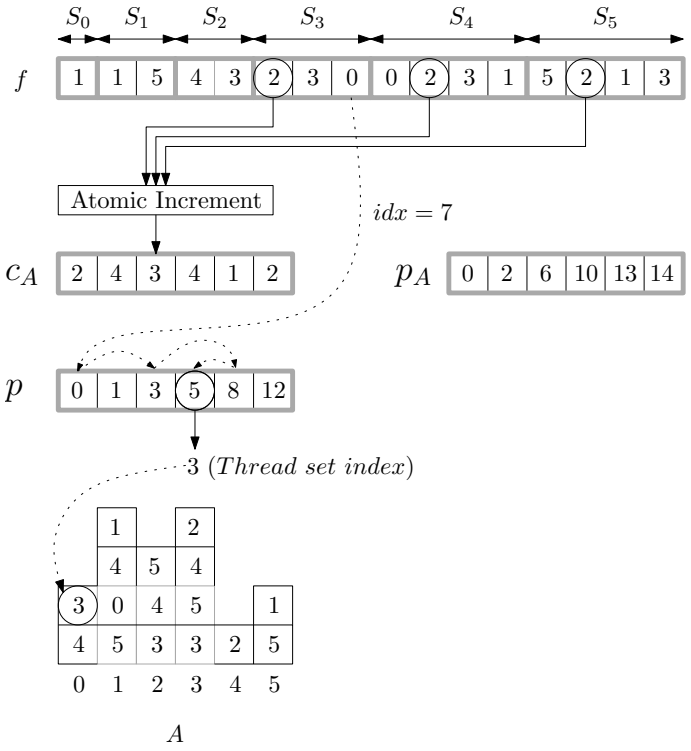



Figure 43: Apparitions vector $A$ construction

### 6.1.4 *Determining the intersection sets*

To compute the intersections between the families $\mathcal{F}$ and $\mathcal{F}'$ we use their apparitions vector structures $A$ and $A'$. The main idea is to count and report the elements in common between each pair of sets. This is done by using the fact that an element $e$ is contained in the sets stored in $A[e]$ and $A'[e]$, and thus, $e$ is present in, and only in, the intersection sets obtained by intersecting two sets whose indices are, one in $A[e]$, and the other in $A'[e]$. The process is parallelized not only by storing the element $e$ in each of the resulting intersection sets in parallel, but also considering the $n$ elements of the domain in parallel.

In the first step, we construct the apparitions vectors $A$ and $A'$ following the algorithm presented in Section 6.1.3. Then, we count the number of intersection elements, corresponding to the sum of the cardinalities of the intersection sets. With this aim we create the array $c_t$, of size $n$, with $c_t[e] = c[e].c'[e]$. That is, $c_t[e]$ is the total number of intersection sets where the element $e$ appears. Then, $p_t$ is created as the prefix sum of $c_t$. From them we know the number of intersection elements, which is denoted $m_{int}$.

Next, we compute the intersection sets. The idea is to run a CUDA kernel with as many threads as intersection elements so that each thread stores the element it represents in the intersection set it corresponds to. The process takes place in two steps. First, we count the cardinality of each intersection set, and then, the intersection sets are obtained.

To determine the intersection sets cardinality, we create the counting intersection matrix $c_{int}$ of size $k \cdot k'$ where $c_{int}[i][j]$ stores the number of elements in common between sets $S_i$ and $S'_j$. In the GPU, $c_{int}$ is linearized and stored in a 1D array. After initializing $c_{int}$ to zero, we launch a kernel with $m_{int}$ threads. We consider $c_A[e] \cdot c'_A[e]$ threads with consecutive indices for each element $e \in D$, thus, each thread is responsible for a specific intersection element $e \in S_i \cap S'_j$. The thread starts determining the element $e$ it corresponds to by locating the thread index $idx$ in $p_t[e]$ with a dichotomic search. Then, once $e$ is known, $p_t[e]$ is used to determine the positions of $A[e]$ and $A'[e]$ where the set indices $i$ and $j$ are stored. Finally, $c_{int}[i][j]$ is incremented by one using an atomic operation. An example of this process is shown in Figure 44.

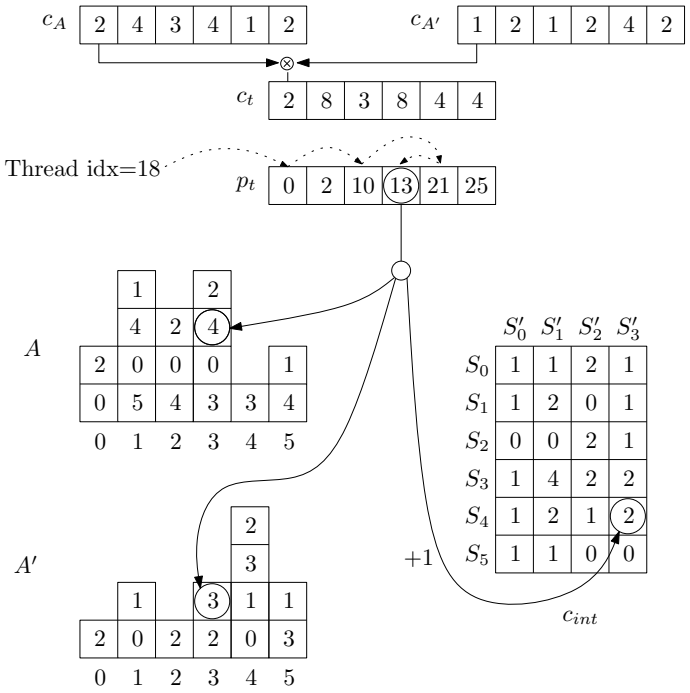



Figure 44: Counting intersections matrix obtention

In the second step, we create $p_{int}$ as the prefix sum of the 1D array $c_{int}$. Finally, allocate $f_{int}$, an array of size $m_{int}$ to store the intersection sets. This is filled with the process used to compute $c_{int}$, but now instead of incrementing in one $c_{int}[i][j]$ the element $e$ is stored in the corresponding position of $f_{int}$. This is done by using $p_{int}[i][j]$ and an auxiliary array $s$ of size $k \cdot k'$ initialized to zero, to know how many elements have already been stored in the set obtained as $S_i \cap S_j$. When the process ends, arrays $c_{int}, p_{int}, f_{int}$ represent the intersection family $\mathcal{I}$.

Notice that, by intersecting all the sets of one family $\mathcal{F}$ with those of $\mathcal{F}'$, the output family $\mathcal{I}$ may have many empty and duplicated sets. Thus, it is reasonable and in most cases necessary to remove them from $\mathcal{I}$. In next sections we explain how they can be removed.

### 6.1.5 *Removing empty sets*

The intersection family $\mathcal{I}$ obtained according to the process described before has exactly the $k \cdot k'$ sets obtained by intersecting each set of $\mathcal{F}$ with all the sets of $\mathcal{F}'$. But some of them, usually many

of them, correspond to empty sets. Whenever a set $\mathcal{F}_i$ has no element in common with a set $\mathcal{F}'_j$, $c_{int}[i][j]$ contains a zero. Because we do not know in advance the number of intersection sets which have at least one element and GPU's cannot make use of dynamic memory, we have to allocate enough memory space to handle the worst case, representing $k \cdot k'$ different intersection sets. Thus, if the number of empty sets is very big, the structure $\mathcal{I}$ is unnecessary very big. The aim is to reduce the structure $\mathcal{I}$ so it only contains the non empty sets, and free the unnecessary occupied memory. We proceed as follows.

The total number of nonempty intersection sets is computed during the intersection sets determination. We use a global variable which is incremented by 1 whenever a thread increments for the first time an element of $c_{int}$. To remove the empty sets, new counting and positioning arrays $c'_{int}$, $p'_{int}$ are allocated according to this counter. By using a parallel kernel with $k \cdot k'$ threads we obtain $c'_{int}$ storing the non zero elements of the 1D array $c_{int}$ in $c'_{int}$. Because we compute this in parallel, we need an auxiliary array $c_e$ to maintain the order of the sets. This array is also computed during the intersection sets determination.

First, we initialize the array $c_e$ of size $k \cdot k'$ to zeros. Then, while we compute the array $c_{int}$, every time a thread increments the value of a $c_{int}$ position whose actual value is 0, we set to 1 the value of the corresponding position of $c_e$. Thus, at the end $c_e[i] = c_e[j] = 1$ if $S_i \cap S_j \neq \emptyset$. Then, we compute $p_e$ as the prefix sum of $c_e$. Now, using $p_e$, we can compute $c'_{int}$ in parallel maintaining the order of the intersection sets.

Finally, $p'_{int}$ is computed as the prefix sum of $c'_{int}$. Notice that $f_{int}$ needs no modifications because empty sets are not present here.

From now, and abusing notation $\mathcal{I}$, $c_{int}$, $p_{int}$, $f_{int}$ and $k_{int}$ refer to the intersection family without empty sets.

### 6.1.6 *Removing duplicated sets*

Our parallel approach to remove the duplicated sets is based on the two following observations: 1) two equal sets have the same cardinality; 2) equal sorted sets are stored in consecutive positions in a lexicographical sorted family, according to any total order of the domain elements.

By using these observations we remove the duplicated sets in three steps. First, we split the intersection family $\mathcal{I}$ into groups of sets of equal cardinality. Then, we sort the sets elements and the sets of each group in lexicographical order. Finally, we remove the duplicated sets of each group checking whether two consecutive sets are equal. An example of the process is shown in Figure 46.

*Step 1: Splitting sets by cardinality*

We sort $c_{int}$ using a GPU parallel algorithm [47], while $p_{int}$ is accordingly reorganized. Then, observing that all sets of equal cardinality are now grouped together, we split $\mathcal{I}$ into groups of the same cardinality. We denote $\mathcal{I}_g$ the family containing only the sets of $\mathcal{I}$ of cardinality $g$.

Steps 2 and 3 are performed on each group $\mathcal{I}_g$. We use $c_g$, $p_g$ and $f_g$ to denote the corresponding counting, positioning and family arrays, meanwhile $k_g$ and $m_g$ represent the number of sets and elements of $\mathcal{I}_g$.

*Step 2: Lexicographically sorting the sets of equal cardinality*

To have equal sets in consecutive positions after sorting $\mathcal{I}_g$ in lexicographical order, we need to have the elements within each set also sorted. However, we do not need to sort the elements according to a particular ordering of the domain, any common ad-hoc order on the elements works. Thus, we use a simple and efficient technique, called weak sorting, which sorts the elements of all the sets uniformly according to an arbitrary total order computed on the fly [73, 74]. Weak sorting works as follows: a) associate to each element all the sets where it is contained; b) traverse the domain elements and place them to the sets they belong to. In this way, at the end, all sets contain

their elements stored in lexicographic order, according to the total order computed during the weak-sorting. We denote $\mathcal{I}_g^w$ the family of sets obtained from $\mathcal{I}_g$ after the weak sorting process.

The weak sorted family $\mathcal{I}_g^w$ is computed as follows. We use an already existing and implemented GPU sorting algorithm [47] to sort all the sets. However, this GPU sorting algorithm is very efficient sorting very large sets and in this case we have to sort many short sets. In order to take the maximum benefit of the GPU sorting algorithm we compute $\mathcal{I}_g^r$, by mapping the elements in each set to new integers guaranteing that the integers stored in $S_i^r$ are smaller than all those in $S_j^r$, whenever $i < j$. To use not too big integers in $\mathcal{I}_g^r$ we use two arrays $\mathcal{I}_g^{min}$ and $\mathcal{I}_g^{max}$, with the minimum and maximum element stored in each set of $\mathcal{I}_g$. They are obtained by launching a kernel with $m_g$ threads and updating their values whenever it is necessary by using atomic operations. Next, we store in $\mathcal{I}_g^{dif}$ the differences between $\mathcal{I}_g^{max}$ and $\mathcal{I}_g^{min}$ by using $k_g$ threads. These differences are accumulated by using the exclusive scan algorithm and the result is again stored in $\mathcal{I}_g^{dif}$. Finally, $\mathcal{I}_g^r$ is computed by adding to each element of each set $S_i$ the value $\mathcal{I}_g^{dif}[i] - \mathcal{I}_g^{min}[i]$. The family $\mathcal{I}_g^r$ is considered as a unique set by concatenating the different sets and $\mathcal{I}_g^r$ is sorted with a GPU sorting algorithm. Finally, the initial elements are recovered by substracting the previously added value. Thus, finally, we have the family $\mathcal{I}_g^w$ with the elements of the sets sorted by increasing order. An example of the process is shown in Figure 45.

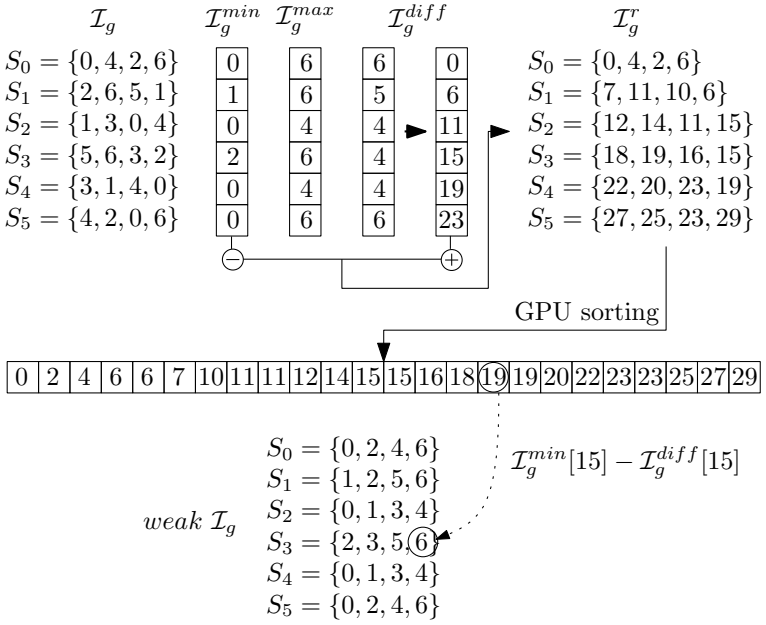



Figure 45: Sorting the sets elements

Next, we lexicographical sort the sets of the family $\mathcal{I}_g^w$ using the radix sort algorithm (Figure 46a). We name i-column, the set determined by the i-th element of a each set of $\mathcal{I}_g^w$. The radix sort algorithm sorts the sets of $\mathcal{I}_g^w$ column by column. First, we sort the sets by the 0-column, then, the resulting sorts of the first step are sorted by the 1-column, and keeps on until the last column is considered. It is important to remark that the sorting of each column must be stable in order to guaranty that the radix sort works.

We sort each column using a GPU stable sort algorithm [47]. Note that the sorting of the i-column depends on the the result of the sorting of the previous j-column for all $j < i$. Thus, in order to obtain the correct result we should reorganize the whole structure $\mathcal{I}_g^w$ after sorting each column. Because commonly $f_g \gg p_g$, instead of reorganize all the elements of $f_g$ at each iteration, we just reorganize $p_g$ and the GPU sorting algorithm sorts according to it. Once the process ends the sets are lexicographically sorted according to their elements which are also sorted.

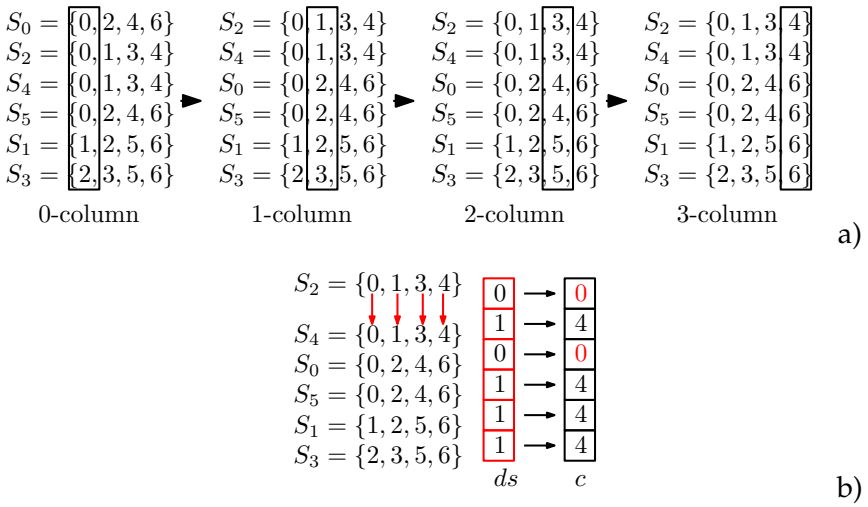



Figure 46: a) Lexicographically sorting of $\mathcal{I}_g$. b) Removing duplicated sets

*Step 3: Eliminating duplicates*

Once we have the elements within each set sorted and the sets of the family $\mathcal{I}_g^w$ lexicographical sorted, equal sets are stored in consecutive positions. Thus, we only need to compare adjacent sets to eliminate duplicates (Figure 46b). To determine if two consecutive sets, $S_i^w$ and $S_{i+1}^w$, are equal, we compare the element $e_i^j$ at position $j$ of the set $S_i^w$ with the element $e_{i+1}^j$ at position $j$ of the set $S_{i+1}^w$, for each $0 \leqslant j < n_g$. As long as we found two different elements the set $S_i^w$ is different from set $S_{i+1}^w$.

Since all the elements and all the sets can be compared independently, this is a very parallelizable process. Duplicated sets are eliminated by first creating an array $ds$ of size $k_g$, initialized zero. Then, we launch a kernel with $m_g - g$ threads assigning an element per thread, except for the elements of the last set which is not compared to any other set. Each thread compares its corresponding element, $e_i^j$ with $e_{i+1}^j$. If they are different $ds[i]$ is set to one indicating that the set $S_i^w$ and $S_{i+1}^w$ are different. When the process finishes those positions in $ds$ containing a zero correspond to the duplicated sets. To end with, we set to zero the corresponding positions of the original counting array $c$ indicating that they have to be removed, notice that if there exist equal sets only one of them is marked with a zero, the other one will maintain its original value.

When steps 2 and 3 have been computed for each group, the original $\mathcal{I}$ structure is updated according to the new size. The counting and positioning arrays $c$ and $p$ are resized and recomputed as it was done when removing empty sets. Array $f$ is also recomputed to remove duplicates by using both the original and the resized positioning and counting arrays and the original array $f$.

### 6.1.7 *Dealing with huge families*

Because the initial 1D-arrays $c_{int}$ and $p_{int}$ representing the intersections family $\mathcal{I}$ have size $k \cdot k'$, depending on the input families they the GPU memory may not be sufficient to store them. When this happens the intersection family can be obtained by parts in the following way.

The apparitions vector $A$ is created as before, meanwhile, while the apparitions vector $A'$ and the structure representing the intersections family $\mathcal{I}$ are created in several iterations. For each iteration, we determine two set indices $\alpha$ and $\beta$ with $0 \leqslant \alpha < \beta < k'$, so that we can store in the GPU all the needed 1D-arrays to represent both, the subfamily $\mathcal{F}'_{[\alpha,\beta]} = \{S'_j \text{ with } \alpha \leqslant j \leqslant \beta\}$ and the intersection family $\mathcal{I}_{[\alpha,\beta]}$ obtained as the intersection of $\mathcal{F}$ with $\mathcal{F}'_{[\alpha,\beta]}$. Then $\mathcal{I}_{[\alpha,\beta]}$ is computed in the GPU and once it does not contain empty sets nor duplicated sets, we transfer the output

vectors $c_{int}[\alpha, \beta]$ and $f_{int}[\alpha, \beta]$ to the CPU. All the GPU arrays, except for those needed to store the family $\mathcal{F}$, are deleted to restart the algorithm with the next subfamily of $\mathcal{F}'$.

In CPU memory, we store two vectors of arrays, $v_c$ and $v_f$. In $v_c$, we store the arrays $c_{int}[\alpha, \beta]$ copied from the GPU, so that $v_c[i]$ contains the array $c_{int}[\alpha, \beta]$ obtained in the $i$-th iteration. Similarly, $v_f[i]$ contains the array $f_{int}[\alpha, \beta]$ obtained in the $i$-th iteration. When all the sets of $\mathcal{F}'$ have been considered, $v_f$ may contain duplicated sets appearing in different arrays that should be eliminated. This is again done in the GPU, with this aim, we build the $\mathcal{I}$ structure allocating the 1D-arrays $c_{int}$, $p_{int}$ and $f_{int}$ in the GPU memory. The arrays of $v_c$ are stored one after the other in $c_{int}$ and those of $v_f$ in $f_{int}$, the array $p_{int}$ is computed as the prefix sum of $c_{int}$. Finally, duplicates are eliminated with the previously provided algorithm.

### 6.1.8 *Reporting intersection sets*

Remember from Section 6.1.2 that, the input families $\mathcal{F}$ and $\mathcal{F}'$ have been preprocessed so the elements of the postprocessed families only contain elements present in both families. Thus, elements had been reindexed according to the domain $\overline{D} = (\bigcup S_i) \cap (\bigcup S'_j)$. In order to restore the elements to their original values, we use the already created array $\overline{d}_{off}$ to reindex the elements from the family $\mathcal{I}$. We launch a parallel kernel with one thread per element in $\mathcal{I}$ and each thread reindex its corresponding element value according to $\overline{d}_{off}$. Then, we only have to copy from the GPU to the CPU memory the structure representing the family $\mathcal{I}$.

## 6.2 COMPLEXITY ANALYSIS

When we analyze the complexity of a GPU algorithm, we should take into account the total work, the thread work, the number of accesses to memory and the transferred values between CPU and GPU. The total work is the total number of instructions realized during the whole algorithm. The thread work is the number of operation done by a single thread. The thread work gives an idea of the degree of parallelism obtained. Despite the parallelization does not decrease the total work complexity of the algorithm it has an important effect on the running times. Finally, a GPU algorithms performance also depends on the number of memory accesses and on the transferred values from CPU to GPU and viceversa.

Table 4 contains the GPU complexity analysis of each step of our approach, here we analyze each part of the algorithm independently.

The total work of initial step is $O(m \log(k \cdot n) + m' \log(k' \cdot n) + n + k)$. The terms of the total work with a log factor are done by threads doing the log term work each, meanwhile the other factors are done by threads doing $O(1)$ work each.

Concerning the determination of the intersection family, the total work is $O(n + m \log k + m' \log k' + k \cdot k' + m_{int} \log n)$. Again, the terms of the total work with a log factor are done by threads doing the log term work each, meanwhile the other factors are done by threads doing $O(1)$ work each. Notice that in this case $n$, $m$, $k$ and $k'$ refer to the already reduced families, thus they are not the original values but those obtained after reducing the input families.

Removing the empty sets is a completely parallelized part of the process, it has a total work of $O(k \cdot k')$ work, but done with the same amount of threads doing $O(1)$ work each. Finally, removing duplicates requires splitting the sets by cardinalities which is done with a standard sorting algorithm with $O(m_{int} \log m_{int})$ total work. Next, the sets are handled per groups of equal cardinality. Sorting the sets elements, of the sets of cardinality $g$, requires a total work of $O(m_g \log m_g)$. Lexicographically sorting the sets is done with a sorting algorithm sorting $g$ times $k_g$ elements, representing a total work of $O(gk_g \log k_g)$. Once all the cardinalities have been considered, the total work done is $O(m_{int} \log m_{int})$ which is needed to sort the sets, ones they are already sorted, eliminating duplicates requires $O(m_{int} \log k_{int})$ total work with $O(\log k_{int})$ work per thread. Obtaining the output family without duplicates takes $O(m_{int} \log k_{int})$ total work with $O(m_{int})$ thread doing $O(\log k_{int})$ work each.

| | | Total work | Thread work | Mem. access |
|---|---|---|---|---|
| $\overline{D}$ determination | $d$ initialization to $\emptyset$ | $O(n)$ | $O(1)$ | $O(n)$ |
| | elements in $\mathcal{F}$ | $O(m \log k)$ | $O(\log k)$ | $O(m \log k)$ |
| | elements in $\mathcal{F}'$ | $O(m' \log k')$ | $O(\log k')$ | $O(m' \log k')$ |
| | $\overline{D}$ obtention | $O(n)$ | $O(1)$ | $O(n)$ |
| | $d_{off}$ computation | $n$ *values prefix sum* | | |
| $\overline{\mathcal{F}}$ computation | $\overline{m}$ determination | $O(m \log k)$ | $O(\log k)$ | $O(m \log k)$ |
| | new $c$ and $c_c$ | $O(k)$ | $O(1)$ | $O(k)$ |
| | new positioning | $\overline{k}$ *values prefix sum* | | |
| | new $\mathcal{F}$ | $O(m \log n)$ | $O(\log n)$ | $O(m \log n)$ |
| $\overline{\mathcal{F}'}$ computation | *equivalent to the previous one with* $k'$, $m'$ *and* $\overline{k}'$ | | | |
| $A$ | $c_A$ initialization | $O(n)$ | $O(1)$ | $O(m)$ |
| | $c_A$ obtention | $O(m)$ | $O(1)$ | $O(n)$ |
| | $p_A$ computation | $n$ *values prefix sum* | | |
| | $a$ | $O(m \log k)$ | $O(\log k)$ | $O(m \log k)$ |
| $A'$ | *equivalent to the previous case with* $k'$ *and* $m'$ | | | |
| $\mathcal{I}$ computation | $c_t$ initialization | $O(n)$ | $O(1)$ | $O(n)$ |
| | $p_t$ obtention | $n$ *values prefix sum* | | |
| | $c_{int}$ initialization | $O(k \cdot k')$ | $O(1)$ | $O(k \cdot k')$ |
| | $c_{int}$, $c_e$ obtention | $O(m_{int} \log n)$ | $O(\log n)$ | $O(m_{int} \log n)$ |
| | $p_{int}$ obtention | $k \cdot k'$ *values prefix sum* | | |
| | $f_{int}$ obtention | $O(m_{int} \log n)$ | $O(\log n)$ | $O(m_{int} \log n)$ |
| $\emptyset$ removal | $c_e$ initialization | $O(k \cdot k')$ | $O(1)$ | $O(k \cdot k')$ |
| | $p_e$ obtention | $k \cdot k'$ *values prefix sum* | | |
| | new $c_{int}$ and $p_{int}$ | $O(k \cdot k')$ | $O(1)$ | $O(k \cdot k')$ |
| Card. sorting | sorted $c_{int}$ - $p_{int}$ | $O(k_{int} \log k_{int})$ | *sorting* $k_{int}$ *values* | |
| *Sets sorting | $\mathcal{I}_g^{min}$ and $\mathcal{I}_g^{max}$ | $O(m_g \log k_g)$ | $O(\log k_g)$ | $O(m_g \log k_g)$ |
| | $\mathcal{I}_g^{dif}$ computation | $O(k_g)$ | $O(1)$ | $O(k_g)$ |
| | $\mathcal{I}_g^{dif}$ accumulation | $k_g$ *values prefix sum* | | |
| | $\mathcal{I}_g^{r}$ computation | $O(m_g \log k_g)$ | $O(\log k_g)$ | $O(m_g \log k_g)$ |
| | $\mathcal{I}_g^{w}$ obtention | $O(m_g \log m_g)$ | *sorting* $m_g$ *values* | |
| *Lex. order | *GPU stable sorting of* $g$ *columns with* $k_g$ *values each* | | | |
| *Duplicates elim | $d_s$ initialization | $O(k_g)$ | $O(1)$ | $O(k_g)$ |
| | $d_s$ computation | $O(m_g)$ | $O(1)$ | $O(m_g)$ |
| | $c_{int}$ update | $O(k_g)$ | $O(1)$ | $O(k_g)$ |
| Final $\mathcal{I}$ | final $c_{int}$ | $O(k_{int})$ | $O(1)$ | $O(k_{int})$ |
| | final $p_{int}$ | $k_{int}$ *values prefix sum* | | |
| | final $f_{int}$ | $O(m_{int} \log k_{int})$ | $O(\log k_{int})$ | $O(m_{int} \log k_{int})$ |
| Reporting $\mathcal{I}$ | reindexing $f_{int}$ | $O(m_{int})$ | $O(1)$ | $O(m_{int})$ |

Table 4: Complexity analysis (* refers to the group of sets of cardinality $g$)

In the CPU, we only create the counting an positioning arrays of the input families, thus it takes $O(m + m')$ time.

Summarizing, the provided approach has a CPU time complexity of $O(m)$, transfers $m + m' + 2(k + k')$ integer values to the GPU and $m_{int} + k_{int}$ to the GPU. Concerning the GPU total work is of $O(m \log(k \cdot n) + m' \log(k' \cdot n) + n + k \cdot k' + m_{int} \log m_{int})$ which is, as well, the number of global memory accesses. Notice that the algorithm is quasilineal with respect to the number of elements in the input families and the output one. The other factors of the total work complexity are not directly related to the input nor the output of the problem, but the parts of the algorithm where they appear are completely parallelized, doing $O(1)$ work per thread, which can not be improved at all.

## 6.3 RESULTS

In this section we provide an analysis of the results obtained. We do not compare our algorithms implementation with other existent algorithms, because, to the best of our knowledge, no other study of this problem exists, even only using the CPU.

The experimental results are obtained using an Intel Core2 CPU 6400 with a Nvidia GTX 480. To experiment with our strategy, we have developed a tool to generate synthetic families of sets, which allows to control some parameters of the families to check not only our algorithm goodness, but also its response depending on the input families. In our first experiment, we work with two families of sets generated according to the flock pattern problem presented in Section 5.2. In the second experiment, we consider two families of random uniformly distributed sets. Thus, while in the first experiment, the flock families do not differ much between them, in the second one, the random uniformly distributed families do differ. For both experiments, we intersect from 11.000 sets with 11.000 sets to 19.000 sets with 19.000 sets, where each set contains between 100 and 150 elements within a domain of $10^5$ elements.

*Flock families*. These families contain sets of potential flocks in two consecutive time-stamps. To simulate the flocks behavior two families that do not differ much between them are generated. The experimental results obtained with these families are presented in Figure 47. In Figure 47 a) we show the running times, in seconds, of the main steps of the algorithm: the input families reduction, the apparitions vectors computation, the obtention of the intersection sets with the empty sets already removed, and the duplicates removal. The accumulated value, corresponding to the columns height, gives the total running time of the algorithm. The needed GPU-CPU memory transfer times are also included.

We see that the total running times, which vary between 0.02 and 0.03 seconds, increase as long as the size of the input families increase. The most expensive steps are the reduction of the input domain and the duplicate sets removal, meanwhile, the running times to compute the apparitions vectors and to determine the intersection sets are very small with respect to the others. However, the expensive steps produce an important benefit to the algorithm, as can be seen in Figure 47 b) and c). Figure 47 b) shows how the domain decreases when the reduction step is realized. In this case, few elements are simultaneously present in both families, thus the reduction is huge and in the following steps the number of domain elements is much smaller. Figure 47 c) shows the reduction of the intersection family size when duplicates are eliminated. Finally, Figure 47 d) shows the importance of removing the empty sets by comparing the amount of memory needed when removing or not removing the empty sets. Eliminating empty sets not only saves memory on the GPU but also reduces the amount of transferred values from the GPU to the CPU.

*Uniform families*. In this experiment, we test two random uniformly distributed families. In this case, the different steps of the algorithm have a completely different behavior when compared with the flock families case. Figure 48 a) presents, as in the previous experiment, the running times of the main steps of the algorithm. With this kind of families, the total running times vary between 2 and 11 seconds. Now, the most expensive steps are the computation of the intersection sets and the

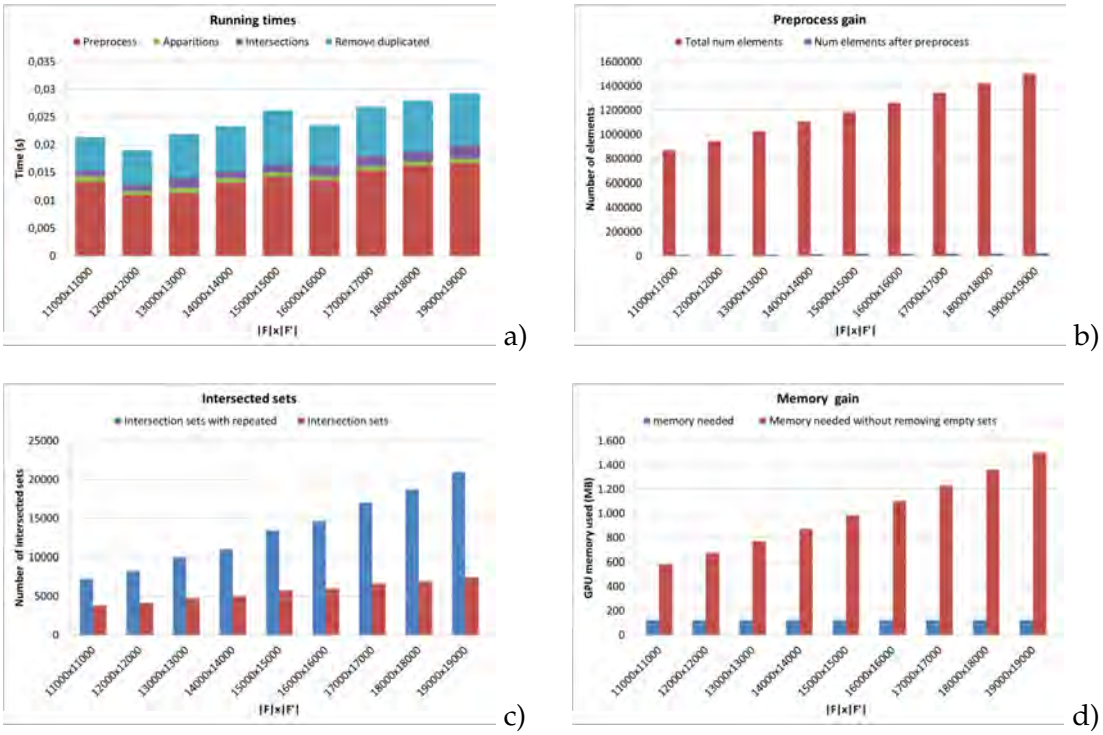



Figure 47: Flock families experimental results

elimination of duplicates. In Figure 48 b) we can see that, contrarily to what happens in the first case, the gain obtained with the reduction of the input families is inappreciable. However, now, the time this step needs is irrelevant. Thus, it makes sense trying to reduce the input families. On the other hand, the output has many duplicated an empty sets, as it can be seen in Figure 48 c) and d), and eliminating them provides a big positive impact in the output size. Note that, when we intersect 19.000 sets against 19.000 sets, the memory needed is 1 GB while without removing the empty sets it would be of 2.3 GB.

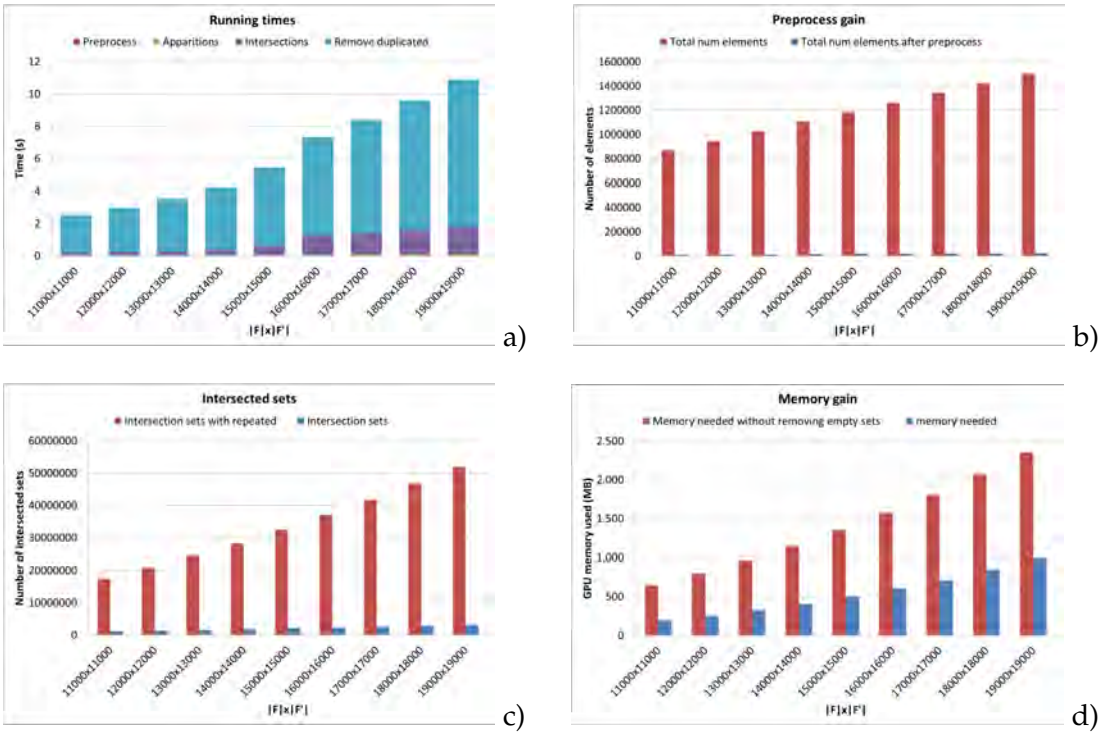


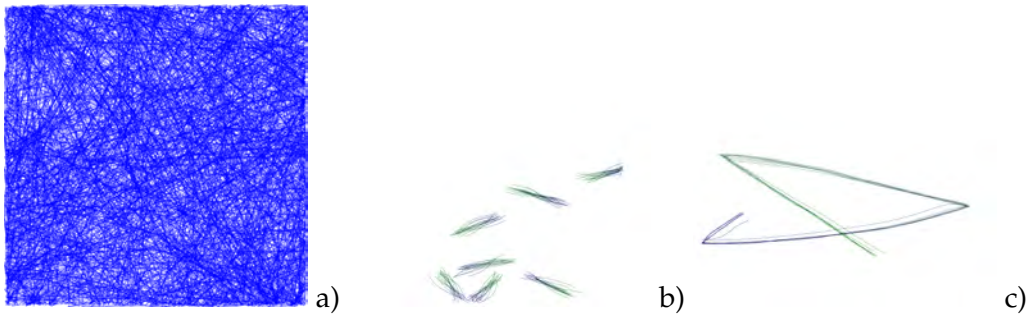

Figure 48: Results for uniformly distributed random families.

# 7

# FLOCK

*Design and programming are human activities;*
*forget that and all is lost.*
*– **Bjarne Stroustrup***

In this Chapter, we focus on the flock pattern, a movement pattern that identifies a group of entities moving close together during a given time interval. In Figure 49 we present a detailed example. Figure 49 a) shows the trajectories of 300 sheeps moving across a fence. It is a simulation generated with NetLogo [18]. In Figure 49 b) we show a set of flocks involving, between 10 and 16 entities, during 20 time steps. Meanwhile, in Figure 49 c) we show one flock with only 5 entities but during a long time interval. Notice that, while no information can be obtained from the original trajectories in Figure 49 a), once flock patterns are detected, useful and interesting information can be extracted from 49 b) and c), demonstrating that efficiently finding flock patterns is an interesting problem to solve.



a) b) c)

Figure 49: a) a set of trajectories. a) a set of short flocks involving many entities. b) a long flock with few entities

Laube et al. [3] defined several spatiotemporal patterns, including flock, and gave algorithms to compute them efficiently. In their paper a flock is a group formed at a time instance, which does not need to move together during any time interval. Later, Gudmundsson et al. [75] extended the algorithmic results. Benkert et al. [5] proposed a more elaborated and realistic definition of a flock, where a minimum number of consecutive time steps are considered, and presented an efficient approximation algorithm for detecting and reporting flocks. Using this model, Gudmundsson and van Kreveld [46] showed that computing the longest and largest duration flocks is NP-hard and presented approximation algorithms. Vieira et al. [55] gave a characterization of the potential flocks for every time step and proposed several heuristic strategies to discover maximal flock patterns with a predefined time duration. Finally, an approach for finding moving flock patterns among pedestrians is presented in [76].

In this Chapter, we present an efficient parallel GPU-based algorithm, designed under CUDA architecture, for reporting three different variants of the flock pattern: 1) all maximal flocks, 2) the largest flock, 3) the longest flock. The experimental results obtained with the implementation of our algorithm show the significance of the presented approach according to its performance.

## 7.1 THE FLOCK PATTERN

Let $E = \{e_0, \ldots, e_{n-1}\}$ be a set of $n$ moving entities. The trajectory $T_i$ of the entity $e_i$ is a sequence of $\tau$ points in the plane $T_i : e_0^i, \ldots, e_{\tau-1}^i$, where $e_j^i$ denotes the position of $e_i$ at time $t_j$ with $0 \leqslant j < \tau$. We assume that the positions are sampled synchronously for all the entities, and that entity $e_i$ moves between two consecutive positions $e_j^i$, $e_{j+1}^i$ with constant speed and without changing direction. Consequently, the trajectory $T_i$ is described by the polygonal line, which may self-intersect, whose vertices are the trajectory points.

Flocks identify groups of entities whose trajectories are close together during a minimum period of time. Formally, according to [5], given a set $E$ of entities, a minimum number of entities $\mu \in \mathbb{N}$, a number of time-steps $\delta \in \mathbb{N}$, a time interval $I_j^\delta = [t_j, t_{j+\delta-1}]$, and a distance $\epsilon \in \mathbb{R}$:

**Definition 3.** *A flock* $f_A(\delta, \mu, \epsilon)$ *in a time interval* $I_j^\delta$*, consists of at least* $\mu$ *entities such that for every instance of time within* $I_j^\delta$ *there is a disk of radius* $\epsilon$ *that contains all the* $\mu$ *entities.*

Benkert et al. [5] present the following alternative and algorithmically simpler definition of a flock, and they prove that both are equivalent.

**Definition 4.** *A flock* $f_B(\delta, \mu, \epsilon)$ *in a time interval* $I_j^\delta$*, consists of at least* $\mu$ *entities such that for every discrete time step* $t_\ell \in I_j^\delta$*, there is a disk of radius* $\epsilon$ *that contains all the* $\mu$ *entities.*

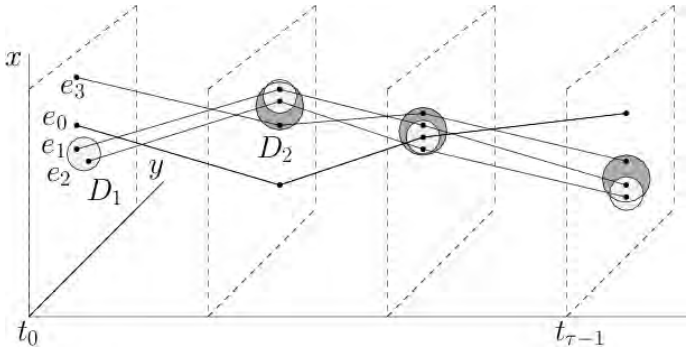



Figure 50: Assuming that the entities are close enough if they are inside the represented disks, for the disk $D_1$, entities $\{e_1, e_2\}$ form flock during time steps $\{t_0, \ldots, t_3\}$. Meanwhile, for disk $D_2$, the entities $\{e_1, e_2, e_3\}$ form flock during the time steps $\{t_1, t_2, t_3\}$.

In the remainder of the Chapter, we refer to $f_B(\delta, \mu, \epsilon)$ flocks whenever we discuss flocks.

The number of reported or determined flocks is a critical issue that can adversely affect the response time and the proper interpretation of the final results. For a time-step $t_\ell \in I_j^\delta$ there may be several circles with radius $\epsilon$ that yield to a flock with the same subset of entities of $E$, so we consider that two flocks are different if they involve two different subsets of entities. Since there can be $\Theta(n^2)$ combinatorially distinct ways to place a circle of radius $\epsilon$ among $n$ points in the plane, at each time-step there can be $\Theta(n^2)$ flock candidates involving different subsets of entities [4]. An algorithm having as output the position of the entities involved in a flock, would have an output size of $\Theta(n^3)$ per time-step. In [5] this problem is skipped by transforming the trajectories into a higher dimensional space and computing all the time steps together. Their solution reports, approximately, the number of flocks, but the entities involved in the flocks are not reported. Moreover, it is easy to observe that a set of entities could be present in many flocks, and even one single entity can be involved in several flocks. For example, a flock of $\mu + 1$ entities contains $\mu + 1$ flocks of $\mu$ entities. To overcome this problem, as in [55], instead of finding all the existent flocks, we are interested only in maximal flocks, so that the entities of one flock can not be a subset of the entities of another flock. Even in the case of finding maximal flocks, at every time-step there may still asymptotically be $\Theta(n^2)$ flock candidates, although in many real situations the number of candidates decreases considerably.

**Definition 5.** *A maximal flock* $\mathfrak{mf}(\delta, \mu, \epsilon)$ *in the time interval* $I_j^\delta$*, is a flock* $f(\delta, \mu, \epsilon)$ *of* $I_j^\delta$ *which is maximal in the family of flocks* $I_j^\delta$*, with respect to the subset inclusion.*

We will denote $\mathcal{M}_j^\delta$ the family of maximal flocks in the time interval $I_j^\delta$, and $\mathcal{M}^\delta$ the family of all maximal flocks, that is, the flocks of $\mathcal{M}_j^\delta$, for $j = 0, \ldots, \tau - 1$.

### 7.1.1 *Characterization*

Viera et al. [55] show that, at a given time step, there is a limited number of locations where the center of a disk determining a potential flock can exist. They prove that, for each pair of entity locations, there exists two such disks. Thus, $2n^2$ disks per time step must be tested.

In this Chapter, we prove that, since we are only interested in maximal flocks, keeping only one of the two disks is enough. Consequently, only $n^2$ disks per time step must be tested, thus the number of tests is reduced by half.

Given two points $s_1, s_2$, let $m = (m_1, m_2)$ be the midpoint of the line segment $s_1 s_2$ and D be a disk of center $c = (c_1, c_2)$ whose boundary passes through $s_1, s_2$ . We say that disk D is located at the "right side" of points $s_1, s_2$ if $c_1 - m_1 \geqslant 0$ and $c_2 - m_2 > 0$, and we say that is located at the "left side" otherwise (see Figure 51.a).

If $d(s_1, s_2) < 2r$, $r \geqslant 0$, there are exactly two disks of radius $r$ whose boundary passes through $s_1$ and $s_2$, one located at the right side of $s_1, s_2$ and the other at the left side (see Figure 51.b).

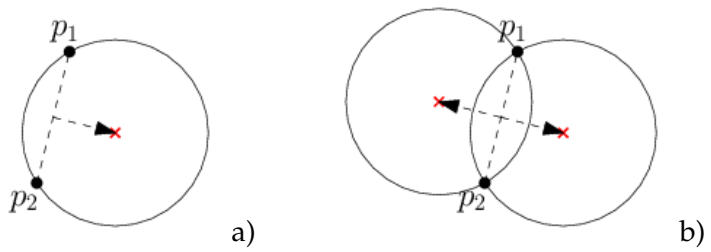



Figure 51: a) a disk located at the right side of $s_1, s_2$. b) right and left side disks whose boundary passes through $s_1$ and $s_2$.

Next, we prove a lemma, similar to one presented in [55], that allows us to bound the number of disks to be tested at each time step.

**Lemma 1.** *Let S be a set of* $n$ *points and* $D'$ *be a disk of radius* $r$ *that covers a subset* $S' \subseteq S$, $|S'| = k \leqslant n$. *There exist another disk* $D''$ *of radius* $r$ *such that: a) has at least two points* $s_i, s_j \in S'$ *on its boundary; b) covers a superset* $S''$ *of* $S'$, $S' \subseteq S'' \subseteq S$; *c) it is located at the right side of points* $s_i, s_j$.

We prove Lemma 1 by construction. We find the disk $D''$ by using a combination of a translation and a rotation of the disk $D'$ (see Figure 52).

**Translation**. If there does not exist any point of S on the (closed) left boundary of the disk $D'$, we translate the disk $D'$

in the positive X-direction until its left boundary meets a point $s_i$ of $S'$. During the translation no point of $S'$ leave the moving disk and maybe some points of $S - S'$ enter or leave the disk (see Figure 52b). In any case, we denote $D_t$ the disk having a point $s_i$ of $S'$ located on its left boundary.

**Rotation**. If necessary, the translated disk D is rotated clockwise or counter-clockwise until its boundary meets another point $s_j$ of $P'$ so that we have a disk $D''$ located to the right side of $s_i$ and $s_j$. During the rotation no point of $P'$ leave the moving disk and maybe some points of $S - S'$ enter or leave the disk (see Figure 52c). Next, we prove that this is always possible. Denote $l_i$ the horizontal straight line through $s_i$. Denote $s_j^+$ the first point reached during a clockwise rotation. If $s_j^+$ is below or on $l_i$ then, the rotated disk is located to the right side of $s_i$ and $s_j^+$ and we are

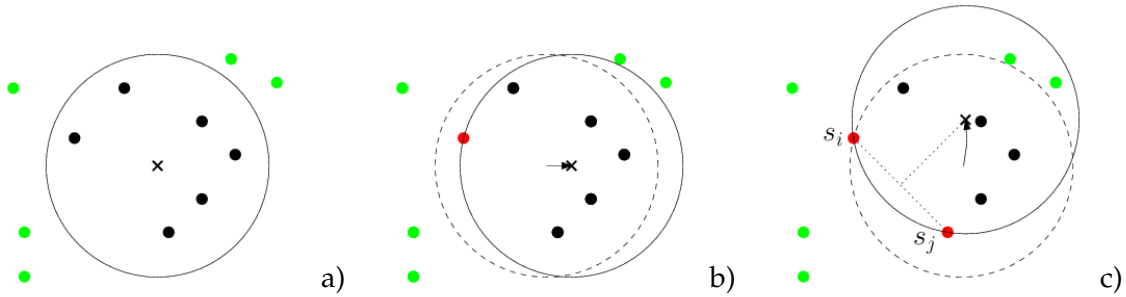



Figure 52: a) disk covering $k = 6$ points; b) translated disk; c) rotated disk.

done. Otherwise, $s_j^+$ is located above $l_i$ and the rotated disk is located to the left side of $s_i$ and $s_j^+$. In this case, we must rotate counter-clockwise. Denote $s_j^-$ the first point reached during the counter-clockwise rotation. Since during the rotation we reach $s_j^-$ before or at the same time that $s_j^+$, point $s_j^-$ is also located above $l_i$. Consequently, the rotated disk is located to the right side of $s_i$ and $s_j^-$ and we are done (see Figure 53).

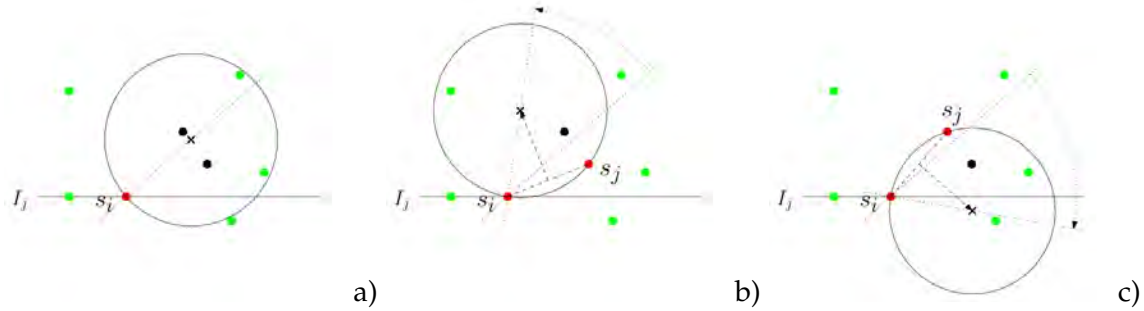



Figure 53: a) Translated disk covering $k = 3$ points. Between the counter-clockwise b) and clockwise c) rotations we select the clockwise one because the obtained disk is located at the right side of points $s_i, s_j$.

### 7.1.2 *Variants of the flock pattern*

What to report and how it should be reported when working with behavioral movement patterns, is a common topic of discussion. In the special case of the flock pattern, one may want to know, for example, which flocks there exist in a fixed interval, or whether a constant given subgroup of entities forms or not flock during a time interval. Some of these variants are discussed in [46], [5] and [55]. In this Chapter, we deal with three variants of the flock pattern, specifically we report:

1. The family $\mathcal{M}^\delta$ of all maximal flocks.
2. The largest flock: the maximal flock $mf(\delta, \mu, \epsilon)$ with highest $\mu$.
3. The longest flock: the maximal flock $mf(\delta, \mu, \epsilon)$ with highest $\delta$.

To compute $\mathcal{M}^\delta$, we find the family $\mathcal{M}_j^\delta$ in the time intervals $I_j^\delta$, for $j = 0, \ldots, \tau - 1$, and the algorithms for determining the largest and the longest flocks are based, again, on the algorithm that computes the family $\mathcal{M}_j^\delta$. Thus, the key element to solve the three variants of the flock pattern is how the family $\mathcal{M}_j^\delta$ is obtained, which is explained in the following Section.

## 7.2 COMPUTING THE FAMILY $\mathcal{M}_j^\delta$ IN THE GPU

We start describing the two main steps of the algorithm used to compute the family of maximal flocks $\mathcal{M}_j^\delta$ which concerns the time interval $I_j^\delta$.

First step. At each time step $t_i$ in the time interval $I_j^\delta$, we compute the family $\mathcal{F}_i$ of potential flocks containing at least $\mu$ entities. Each family $\mathcal{F}_i$ is obtained by using the characterization given in Lemma 1. Next, we find the maximal sets of each family $\mathcal{F}_i$. Abusing notation, we will still denote $\mathcal{F}_i$ the subfamily of potential maximal flocks. At the end of the step, we obtain the array of families of potential maximal flocks $\mathcal{F}_j^\delta = [\mathcal{F}_j, \ldots, \mathcal{F}_{j+\delta-1}]$.

Second step. We intersect the $\delta$ families of potential maximal flocks in $\mathcal{F}_j^\delta$, obtaining a new family of sets which are the intersection of $\delta$ sets, one of each $\mathcal{F}_i$ in $\mathcal{F}_j^\delta$. Finally, the subfamily of maximal sets of at least $\mu$ elements is $\mathcal{M}_j^\delta$.

Next, we describe in detail the algorithm to compute $\mathcal{M}_j^\delta$ which, in order to handle $\delta$ time steps in parallel, uses a multi-grid structure to encode proximity. Accordingly, this Section is organized as follows. In Section 7.2.1 we explain how we construct the multi-grid structure, and in Section 7.2.2 we explain how the potential maximal flocks are obtained.

### 7.2.1 *The multi-grid structure $\mathcal{G}$*

We denote by $E_i = \{e_i^0, \ldots, e_i^{n-1}\}$ the set of locations of the entities of $E$ at time step $t_i$, $i \in \{0, \ldots, \tau - 1\}$. In [55], a single grid is used to perform quick disk range searching queries over each set $E_i$. Instead of using a grid for each time step, we build a multi-grid structure, denoted $\mathcal{G}$, in parallel in the GPU, that allows us to search simultaneously in each one of the $\delta$ sets of $E_j^\delta = [E_j, \ldots, E_{j+\delta-1}]$.

The multi-grid structure contains $\delta$ regular grids, where the edges of the grid cells have length $\epsilon$. Each grid $j$ contains the set of points corresponding to $E_j$. In order to maximize the parallel performance, the top-right corner is defined so that the grid is a square, that is, it has the same number of rows and columns (see Figure 54).

The multi-grid structure $\mathcal{G}$ is composed of 7 arrays allocated in the GPU global memory. In Figure 54 we can see an example for $\delta = 3$. The bottom-left and the top-right corners of each grid are stored in the array $b_\delta$ of size $\delta$, where $b_\delta[j]$ stores the two corner points of the grid containing $E_j$. The number of cells per row (or per column) of each grid is stored in $\widetilde{c_\delta}$ and its prefix sum is stored in $\widetilde{p_\delta}$. Both arrays are integer arrays of size $\delta$.

To represent the $\delta$ grids, we use the array of integers $c_\delta$, whose size depends on the size of the different grids which directly depend on the distribution of the points. For a given grid cell, we store in the corresponding $c_\delta$ position the number of points contained on that cell. The array $p_\delta$ of the same size of $c_\delta$ is obtained as the prefix sum of $c_\delta$. Note that, for a better understanding, in Figure 54 the arrays $c_\delta$ and $p_\delta$ are showed as several 2D grids, but in the GPU memory they are stored as an 1D array, by storing one grid after the other starting from left to right and from top to bottom, thus, the first grid corresponds to the time step $t_i$ and the last one to the time step $t_{i+\delta}$. With this structure we can easily access, in a parallel way, to any element of the $\delta$ grids, by using the fact that the set of points of a cell $w$ starts at position $p_\delta[w]$ of $e_\delta$, and is stored in $c_\delta[w]$ consecutive positions of $e_\delta$. In each $e_\delta$ position we store the two floats of the corresponding location. Finally, $\widetilde{e_\delta}$ is an integer array of size $n\delta$ containing at position $j$ the entity id of the $e_\delta[j]$ location.

To construct the structure $\mathcal{G}$ in parallel we start by allocating CPU arrays $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$, $b'_\delta$ of size $\delta$, and $p'$ and $\widetilde{e_\delta}'$ of size $n\delta$. While we load the input $E_j^\delta$ into $p'$, the arrays $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$ and $b'_\delta$ are accordingly filled. Then $\widetilde{c_\delta}$, $\widetilde{p_\delta}$, $b_\delta$ and $p$ are allocated in GPU memory and the $\widetilde{c_\delta}'$, $\widetilde{p_\delta}'$, $b'_\delta$ and $p'$ values are transferred to GPU memory to $\widetilde{c_\delta}$, $\widetilde{p_\delta}$, $b_\delta$ and $p$ respectively.

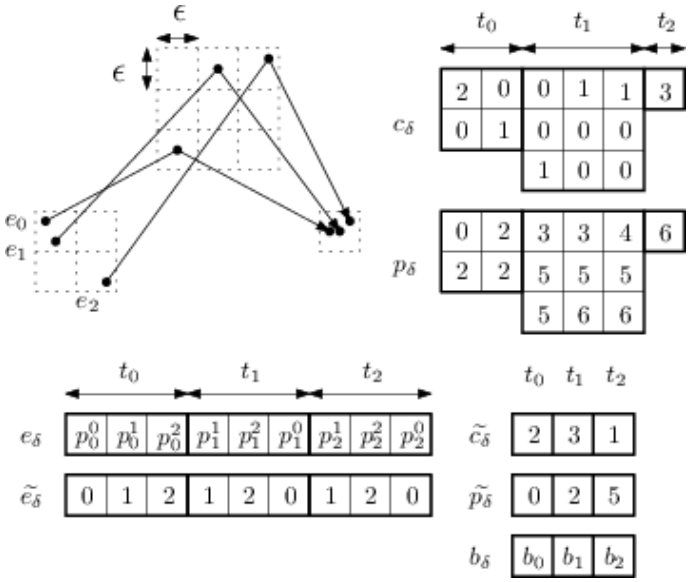



Figure 54: Example of multi-grid $\mathcal{G}$ for $\delta = 3$.

Then, $c_\delta$ and $p_\delta$ are allocated in the GPU memory according to the values of $\widetilde{c_\delta}$ and $\widetilde{p_\delta}$, respectively. Additionally, the arrays $e_\delta$ and $\widetilde{e_\delta}$ are allocated with size $n\delta$ in GPU memory. Because GPU has no dynamic memory we have to construct $\mathcal{G}$ in two steps.

First, we fill $c_\delta$ by counting the number of points contained in each cell. This is done in parallel by launching a kernel with $n\delta$ threads, that is, a thread per point and per time step. Each thread $\mathtt{idx}$ reads its $p[\mathtt{idx}]$ value and determines its position in the grid by using its position on the plane, the grid corresponding to the time step the point belongs to, and the arrays $\widetilde{c_\delta}$ and $\widetilde{p_\delta}$. The corresponding $c_\delta$ position is incremented by 1. Because many threads may be dealing with points corresponding to the same $c_\delta$ position, we use atomic operations to ensure no thread interferences.

Once $c_\delta$ is computed, $p_\delta$ is obtained as the prefix sum of $c_\delta$ and we allocate an auxiliary array $v$ with the same size of $c_\delta$, initialized to zeros. Now, we launch a parallel kernel with $n\delta$ threads where each thread $\mathtt{idx}$ reads its $p[\mathtt{idx}]$ value, determines its position $w$ in the grid and stores $p[\mathtt{idx}]$ at $e_\delta[p_\delta[w] + v[w]]$. Every time a thread stores its associated point into $e_\delta$, the corresponding $v$ value is incremented by one by using an atomic operation.

### 7.2.2 *Finding the family* $\mathcal{M}_j^\delta$

The computation of the array of potential flocks $\mathcal{F}_j^\delta$ of a given interval $I_j^\delta$ is done in two main steps. First, we find the center of the disks which are obtained using the characterization given in Lemma 1, and second, we report the entities contained in those disks. Additionally, because we do not have dynamic memory in the GPU, we must split each of these processes in several steps.

To find the disk centers, we first count how many disks with there exist according the given characterization, and next we store them. Note that each of the $n$ points in $E_j$ can be paired with at most the other $n-1$ points of $E_j$ to define a disk of radius $\epsilon$. In fact two points define such a disk, and thus are paired, whenever they are at distance smaller than or equal to $2\epsilon$. Since we compute $\delta$ time steps in parallel, we analyze the $n\delta$ points of $E_j^\delta$ at once. Thus, we allocate an array of integers, $D_c$, of size $n\delta$ to store the number points with whom each point $e_j^i$ can be paired with. This is done in parallel by launching a kernel with $n\delta$ threads, that is, a thread per point in $E_j^\delta$. Using the multi-grid structure $\mathcal{G}$, each thread $\mathtt{idx}$ reads the point $e_\delta[\mathtt{idx}] = e_j^i$ it corresponds to, and locates it in the corresponding grid structure. Then, the thread counts how many points the point $e_j^i$ can be paired with, by counting the points of $E_j$ that are at distance at most $2\epsilon$ from $e_j^i$. This is done

by exploring only the cell of $e_j^i$ and its eight neighboring cells. The obtained number is stored in $D_c[idx]$. Note that each paired pair of points $e_j^i$, $e_j^{i'}$ is counted twice, first when $e_j^i$ checks $e_j^{i'}$ and then when $e_j^{i'}$ checks $e_j^i$. To avoid this, we use the entities id. We force the thread corresponding to $e_j^i$ to check only the points $e_j^{i'}$ with $i' > i$. Accordingly, the tread idx checks the pair $e_\delta[idx]$, $e_\delta[k]$ whenever $\tilde{e}_\delta[idx] < \tilde{e}_\delta[k]$. Next, $D_p$ of size $n\delta$ is computed as the prefix sum of $D_c$, and the array D, where we will store the centers of disks, is allocated with size $D_p[n\delta - 1] + D_c[n\delta - 1]$. Finally, the same process is repeated, but instead of counting, the center of the right disk defined by each paired pair of points is stored.

In the second step, we actually report the potential flocks. To do this we use $\mathcal{G}$ to perform the range searching queries, using the points stored in the array D as the centers of the disk queries with radius $\epsilon$. In order that a regular grid structure handles range searching queries properly, any query point has to be located in the grid. It may happen that some disk centers were placed outside the grid corresponding to their time step. Thus, before performing the range searching queries we reconstruct $\mathcal{G}$ so that each grid, of the multi grid $\mathcal{G}$, contains the disk centers of its corresponding time step. We could modify $\mathcal{G}$, but it is faster to rebuild it than to modify it.

To count the number of points contained in each disk we first allocate $P_c$, the array of integers of size equal to the total number of disks reported, which is $D_p[n\delta - 1] + D_c[n\delta - 1]$, initialize it to zeros and launch a kernel with one thread per disk. Each thread idx reads the center of its disk $D[idx]$, locates $D[idx]$ on the grid and checks the distance between the center of the disk and the points of its neighboring cells. Each thread counts how many points are at distance $\leqslant \epsilon$ to its disk center, and if the number of points is $\geqslant \mu$, it is stored in $P_c[idx]$. Once we know the number of entities per disk, we compute $P_p$ as the prefix sum of $P_c$.

Note that it may happen that no entities were reported for a time step. In such a case, if we are interested in reporting the flocks of a given time interval, we have finished, because no flock can exist.

If there is at least one disk per time step with at least $\mu$ entities, we continue to report the potential flocks. The last position of $P_p$ plus the last position of $P_c$ gives us $\omega$, the size of the array used to store the potential flocks. Thus, the array P is allocated as an integer array with size $\omega$ and is filled in parallel using a thread per disk. The process is the same as before but this time, instead of counting, each thread stores in P the entities id contained inside each disk by using $P_p$.

Note that, $P_c$, $P_p$ and P denote a structure containing $\delta$ families of sets. This is the array of potential flocks $\mathcal{F}_j^\delta$, which stores $\delta$ families of sets containing the potential flocks of $\delta$ consecutive time steps starting at time step $t_j$. We denote $\mathcal{F}_i$ the family of potential flocks of the time step $t_i$.

The last step is to reduce each $\mathcal{F}_i \in \mathcal{F}_j^\delta$ so that it only contains the sets which are maximal, with respect to the partial order induced by the subset relation, in each family. To this end, we use the parallel algorithm presented in Chapter 5. In the Chapter we explained how the algorithm can be modified in order to discard the duplicated sets. We use this last variant of the algorithm and we additionally discard those sets $S_k$ of $\mathcal{F}_i$ with $|S_k| < \mu$.

When the process finishes, for each time step of the interval $I_j^\delta$, we have in $\mathcal{F}_j^\delta$ the families of maximal potential flocks with at least $\mu$ entities.

## 7.3 REPORTING FLOCK PATTERNS

In this section, we explain how we report the three variants of the flock pattern that we study. In Section 7.3.1 we explain how the family $\mathcal{M}^\delta$ with all the maximal flocks is obtained and next we describe how the largest and longest flocks are obtained in Sections 7.3.2 and 7.3.3, respectively.

### 7.3.1 *Reporting the family of all maximal flocks $\mathcal{M}^\delta$*

We want to report all the families $\mathcal{M}_j^\delta$ for all $I_j^\delta, j = [0, \ldots, \tau - \delta]$. We could perform the same algorithm over each interval $I_j^\delta$, but many information computed for a given time step within a time interval can be reused for the following time intervals. The idea is to compute the family of

potential flocks for a given interval and compute their intersections in a way that we can reuse the information for the next time interval. To this aim we proceed as follows.

To exploit GPU parallel capabilities, we compute the families of potential flocks in groups of $\delta$ time steps. We first compute $\mathcal{F}_0^\delta$, then $\mathcal{F}_\delta^\delta$, next $\mathcal{F}_{2\delta}^\delta$, and so on, according to the algorithm explained in Section 7.2.2. We compute $\mathcal{F}_{g\delta}^\delta$, when the family $\mathcal{F}_{g\delta}$ is needed for the first time. When we know that a family of potential flocks $\mathcal{F}_i$ will no longer be needed, we delete it.

We have designed an algorithm for computing the families $\mathcal{M}_j^\delta$ that uses the fact that $\mathcal{M}_i^1 = \mathcal{F}_i$ and that $\mathcal{M}_k^{\delta'+1}$ can be obtained from $\mathcal{M}_k^{\delta'}$ and $\mathcal{M}_{k+1}^{\delta'}$ (see Figure 55).

We first find $\mathcal{M}_0^\delta$, once $\mathcal{F}_0^\delta$ has been computed, as follows. Starting at $\mathcal{M}_{\delta-1}^1 = \mathcal{F}_{\delta-1}$, we intersect $\mathcal{M}_{\delta-2}^1 \cap \mathcal{M}_{\delta-1}^1 = \mathcal{F}_{\delta-2} \cap \mathcal{F}_{\delta-1}$, and then we compute the maximal sets and discard those maximal sets containing less than $\mu$ entities, thus, we obtain $\mathcal{M}_{\delta-2}^2$. Next, we intersect $\mathcal{M}_{\delta-3}^1 \cap \mathcal{M}_{\delta-2}^2 = \mathcal{F}_{\delta-3} \cap \mathcal{M}_{\delta-2}^2$ and determining the maximal sets with at least $\mu$ entities we obtain $\mathcal{M}_{\delta-3}^3$. We continue intersecting $\mathcal{M}_{\delta-4}^1 \cap \mathcal{M}_{\delta-3}^3 = \mathcal{F}_{\delta-4} \cap \mathcal{M}_{\delta-3}^3$ and determining the maximal sets with at least $\mu$ entities, to obtain $\mathcal{M}_{\delta-4}^4$. We repeat the process $\delta$ times until we obtain the family $\mathcal{M}_0^\delta$, which contains the maximal flocks with at least $\mu$ entities of $I_0^\delta$, which are added to $\mathcal{M}^\delta$.

Then we proceed to compute $\mathcal{M}_1^\delta$. Instead of recomputing all the intersections, we reuse the previous computations. Starting at $\mathcal{M}_\delta^1 = \mathcal{F}_\delta$, we intersect $\mathcal{M}_{\delta-1}^1 \cap \mathcal{M}_{\delta-1}^1 = \mathcal{F}_{\delta-1} \cap \mathcal{F}_\delta$, compute the maximal sets and discard those containing less than $\mu$ entities, obtaining $\mathcal{M}_{\delta-1}^2$. Next, we intersect $\mathcal{M}_{\delta-2}^2 \cap \mathcal{M}_{\delta-1}^2$, and again determining the maximal sets with at least $\mu$ entities, we obtain $\mathcal{M}_{\delta-2}^3$. We continue intersecting $\mathcal{M}_{\delta-3}^2 \cap \mathcal{M}_{\delta-2}^3$, computing the maximal sets and discarding those containing less than $\mu$ entities, obtaining $\mathcal{M}_{\delta-3}^4$. The process is repeated until we obtain the family $\mathcal{M}_1^\delta$, which contains the maximal flocks of $I_1^\delta$, and we add it to $\mathcal{M}^\delta$.

We repeat the process for $j = 2, \ldots, \tau - \delta$, obtaining at step $j$ the family $\mathcal{M}_j^\delta$ containing the maximal flocks of $I_j^\delta$, which are add to $\mathcal{M}^\delta$.

We want to observe that our technique does not reduce the number of intersections done, in comparison with the trivial strategy which computes $\mathcal{M}_j^\delta$ as the intersection of the $\delta$ initial families $\mathcal{M}_i^1 = \mathcal{F}_i$. However, in our strategy, the used families $\mathcal{M}_i^j$, with $j > 1$, are considerably smaller than the initial families $\mathcal{M}_i^1$, and consequently our strategy leads to faster intersections.

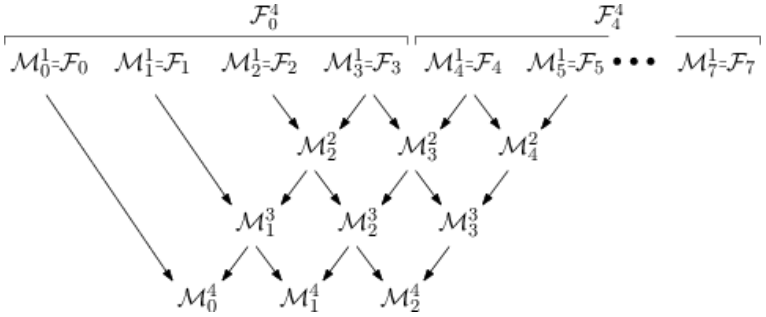



Figure 55: Potential flocks intersection process example

It is important to remark, that Viera et al. in [55], during the process of finding the families of potential flocks do not have into account those entities which for any given time step $t_j$ do not have at lest $\mu$ entities at distance $2\epsilon$. Because we compute the families of potential flocks in groups of $\delta$ time steps, this can not be appliad to our algorithm.

The intersection process is also performed in parallel using the GPU algorithm presented in Chapter 6.

### 7.3.2 *Reporting the largest maximal flock*

To find the largest maximal flock, that is the maximal flock $mf(\delta, \mu, \epsilon)$ with highest $\mu$, we use an approach based on the algorithm to compute $\mathcal{M}_j^\delta$.

Initially, we consider $\mu = 2$, and we compute $\mathcal{M}_0^\delta$ for the time step $j = 0$. We store, among the obtained flocks, the one with the biggest number of entities. In the next time step, $\mu$ is updated to the number of entities of the largest currently found flock and we look for $mf(\delta, \mu, \epsilon)$ flocks, with this new $\mu$ value. In the case that we found a flock with more than $\mu$ entities, the previous largest flock stored is overwritten by the new largest flock, and the $\mu$ value is accordingly updated. We repeat the process until $j = \tau - \delta$. In the case that at a given time step the biggest number of entities is archived in more than one flock, we only keep one of them.

### 7.3.3 *Reporting the longest maximal flock*

The algorithm to find the longest maximal flock is also based on the algorithm for computing $\mathcal{M}_j^\delta$.

Initially, $\delta = 2$ and we compute $\mathcal{M}_0^2$. Whenever we found one or more flocks within the studied interval, we extend these flocks to check for how long their groups of entities remain together. In order to check this condition, we use the algorithm to find $M_j^\delta$, but filtering the input set of entities $E$ to those entities belonging to the flocks we want to extend. Note that, for each flock, the same entities must remain together. When the groups can not be extended anymore, we stop and report one of the remaining flocks. In the next step, $\delta$ is updated to the length of the current longest flock, and we keep on finding $mf(\delta, \mu, \epsilon)$ with the current maximal $\delta$ value. We repeat the process until $j = \tau - \delta$, for the last updated $\delta$ value.

## 7.4 EXPERIMENTAL RESULTS

The experimental results have been obtained using an Intel Core2 CPU 6400 with a Nvidia GTX 480. Each running time has been calculated as the average of 10 executions.

We split the running time into 2 main steps, the finding potential flocks process and the intersection process. The accumulated value, corresponding to the columns height, gives the total running time of the algorithm. The needed GPU-CPU memory transfer times are also included.

We tested our algorithm with the following data sets:

- **Trucks:** Consists of 145 trajectories of 2 school buses collecting (and delivering) students around Athens metropolitan area in Greece for 108 distinct days with a total of 66,096 time step. Extracted from [28].
- **Buses:** Consists of 276 trajectories of 50 trucks delivering concrete to several construction places around Athens metropolitan area in Greece for 33 distinct days with a total of 112,203 time steps. Extracted from [28].
- **Human mobility traces:** Human mobility traces of 92 people from the campus of the KAIST University in Korea with a total of 135,055 time steps. Extracted from [29].

The algorithm has been tested under different situations to see its response. The 'Trucks' and 'Buses' data sets have been tested with the same input parameters $\mu, \epsilon, \delta$ as in [55]. The results show that our implementation runs up to 150 times faster. For instance, the case for the 'Trucks' data set when $\epsilon = 1500, \mu = 5$, and $\delta = 10$, their best algorithm reports running times close to $10^3$ seconds while our algorithm takes less than 7 seconds.

### 7.4.1 $\mathcal{M}^\delta$ *computation varying* $\epsilon$

In this test, we vary $\epsilon$ while we maintain $\mu = 5$ and $\delta = 10$. The results show that the number of flocks reported increases as long as we increase $\epsilon$. It also increases the running times, but note

that the most affected part is the intersection process. This is because the more flocks we report the more intersections we have to compute. Note that the running time of the 'Buses' data set and the 'Trucks' data set are similar despite the number of entities and times steps are clearly bigger in the 'Trucks' data set. This is because the number of flocks reported is bigger than in the 'Trucks' data set where the entities are closer and more flocks exist.

The 'Human mobility traces' data set has a very big number of time steps. Because $\mathcal{M}_j^\delta$ is computed sequentially for $j = [0, \ldots, \tau - \delta]$, the running times increase as long as the number of time steps becomes bigger.

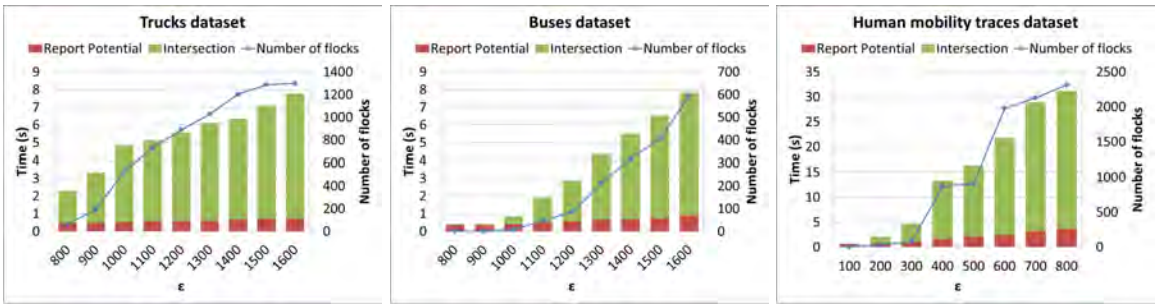



Figure 56: Running times varying epsilon

### 7.4.2 $\mathcal{M}^\delta$ *computation varying* $\mu$

In this case, the parameter $\mu$ is varied, while $\delta = 10$ and $\epsilon = 1200$. As long as we increase $\mu$, the condition to form flock is more restrictive. That is, $mf(\delta, \mu_1, \epsilon)$ is a $mf(\delta, \mu_2, \epsilon)$ whenever $\mu_1 \leqslant \mu_2$. Thus, as long as we increase $\mu$, the number of flocks decreases and, consequently, the running times too (Figure 57). It is remarkable that, for the 'Human mobility traces' data set, the number of flocks decreases significantly fast. This data set is recollecting data about students moving in a campus. These results suggest that all the students meet or divide at the same time, which makes sense from a human behavior point of view.

Note that, in some cases, the intersection process running times may be zero. This is because, even when the restriction is so hard that we do not find any flock, the potential flock test must be done. However, in such a case, the intersection will provably be avoided after applying the maximal sets algorithm, because we only maintain those potential flocks with at least $\mu$ entities. Thus, after finding the maximal sets, the potential flocks will provably be reduced to zero.

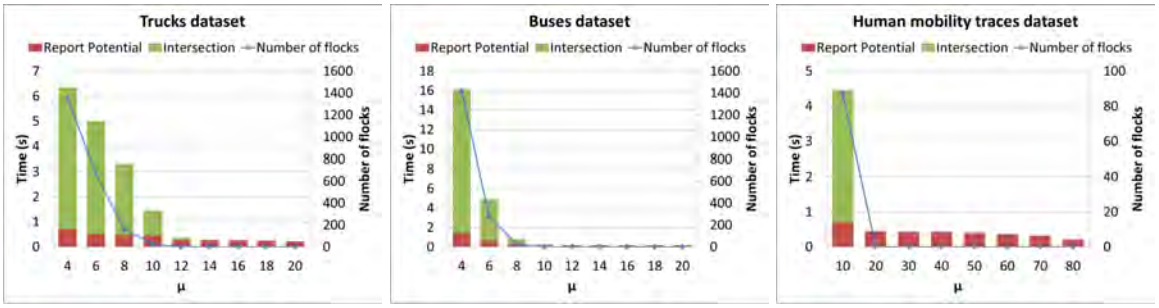



Figure 57: Running times varying $\mu$

### 7.4.3 $\mathcal{M}^{\delta}$ *computation varying* $\delta$

Now, the parameter $\delta$ is varied while $\mu = 5$ and $\epsilon = 1200$. We are reporting more restrictive flocks as long as we increase $\delta$. That is, a $mf(\delta_1, \mu, \epsilon)$ flock is a $mf(\delta_2, \mu, \epsilon)$ flock whenever $\delta_1 \leqslant \delta_2$. Thus, as long as we increase $\delta$, the number of flocks decreases, and consequently, the running times also decrease (Figure 58). However, for the 'Buses' data set the running times and the number of flocks do not seem to have a correlation. Note that, although the total number of flocks decreases, the running times are bigger for the medium values of $\delta$. This because of its input nature, depending on the points distribution they can lead into many small flocks, in terms of size. Thus, it may happen that the maximal sets algorithm discards a lot of sets, and the intersection process is then faster.

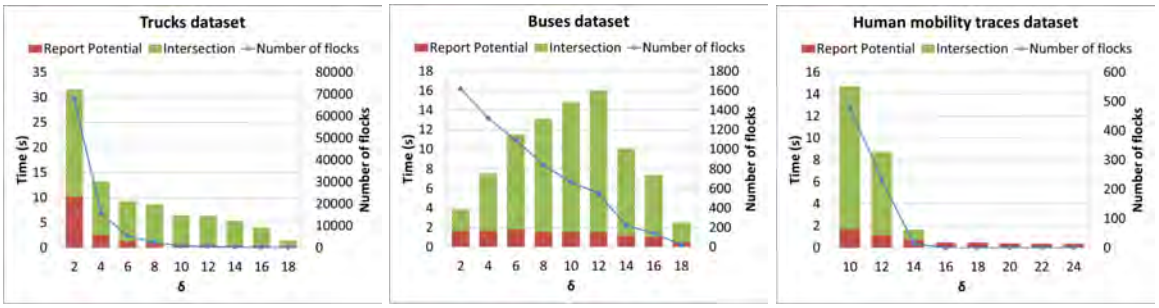



Figure 58: Running times varying $\delta$

### 7.4.4 *Computing the largest flocks*

Figure 59 shows the progressive increase of the $\mu$ value as long as we process a new time step. Additionally, it also shows the total running time accumulated at each time step. The last value of the 'total running time' curve is the total time taken to compute the largest flock.

The results show that the running times are constant when $\mu$ is not updated. This is because we just do the intersection when we have a potential flock with a number of entities bigger than the current $\mu$. When the curve has a lineal behavior means that we find potential flocks, which contain exactly $\mu$ entities, and when the time is almost not increased, we find potential flocks with less than $\mu$ entities.. When $\mu$ needs to be updated the intersection process has to take place and consequently the running times increase.

For the specific case of the 'Human mobility trace' data set, the largest flock is found at the very beginning, an for the rest of the time steps the running times are practically zero.

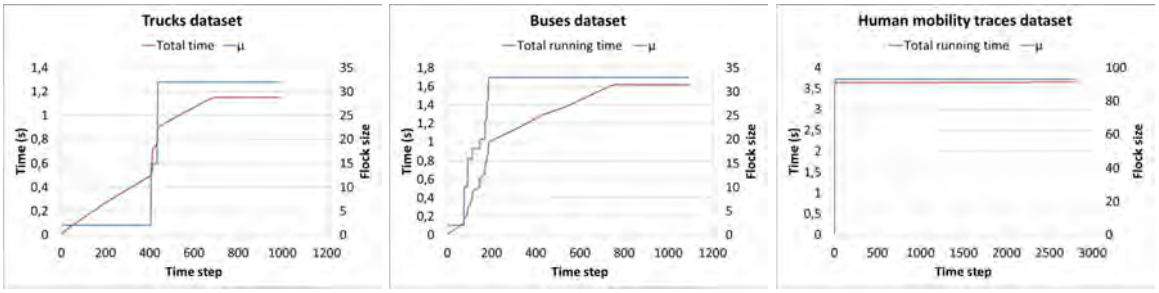



Figure 59: Results for finding the largest flock

### 7.4.5 *Computing the longest flocks*

The test for finding the longest flock (Figure 60) have a different behaviour. For the given datasets, the length restriction prune many flocks as long as δ is increased. Thus, as soon as δ is too restrictive, we do not find any potential flock of this length. The algorithm speeds up so much because the intersection does not take place and the potential flocks process is very fast.

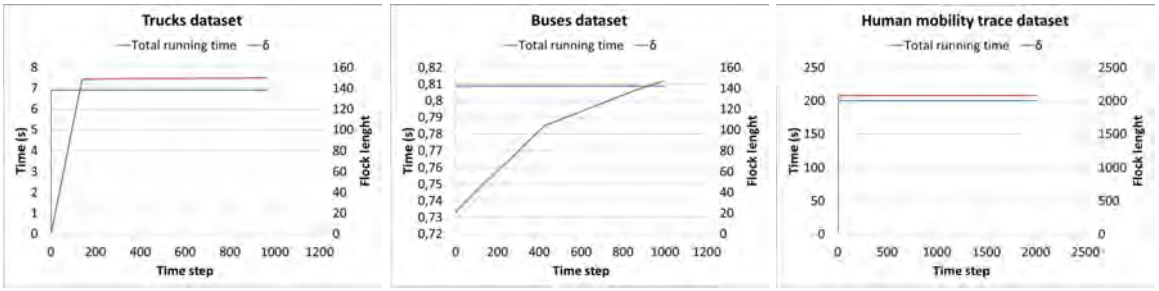



Figure 60: Results for finding the longest flock

# 8

# CONCLUSIONS

*The gap between theory and practice*
*is not as wide in theory as it is in practice.*
***– Unknown author***

In this dissertation we addressed the reporting of movement pattern behaviour in spatio-temporal databases. We proposed a set of new and efficient GPU parallel algorithms that represent a sustancial improvement with the actual algorithms or even new approaches to unthreaded problems.

First, we studied the problem of finding popular places, regions that are visited by at least a given number of entities. We provided two models, depending on whether knowing the number of different entities that have visited a place or knowing the number of times the entities have visited a place, by counting each entity as many times as different visits it has done, is required. We also introduced the notion of the popularity map, a planar structure from which it is easy to report popular places. By working towards practical solutions and in order to obtain good running times, we presented algorithms, to report popular regions and their schematization, that take advantage of the parallel computing capabilities of the Graphics Processing Units. The algorithms presented have no restrictions in either the number of entities or times steps, and let the user specify a desired maximal error in the solutions obtained. Although our proposed approach means reported solutions are approximated, this is acceptable since data of moving entities are approximated too. We also described how we can visualize and obtain a schematization of the reported solutions which in turn, facilitates the understanding of trajectory path data. To allow a deeper study of the trajectory sets, our implementation allows constraints to be added on both the input and output paths. In the case of the input trajectory paths, rather than considering whole trajectories, only those time steps in a specified time slot are taken into account, and for the output, this is done by deleting those parts of the extracted popular places which are considered too small. Finally, we reported and discussed experimental results obtained with our implementation of the algorithms presented showing the efficiency and scalability of our approach.

Next, we presented a parallel algorithm to compute subtrajectory clusters using the continuous Fréchet distance between curves as a measure of proximity. We showed how the steps in the original algorithm by Buchin et al. [30] are mostly parallelizable, which results in a large improvement in running times. Previous implementations of the subtrajectory clustering algorithm have all used the discrete Fréchet distance instead of the continuous. The main advantage with using the continuous is that it allows us to apply compressing techniques to the input data. Various studies have shown that a trajectory can be compressed to 5%-10% of its original size without loosing much information. Thus our implementation gives us the ability to handle much larger data sets than previously possible.

Then, we presented two GPU algorithms two solve two problems. The maximal sets and the family intersection problems. Both algorithms are necessary for solving the flock pattern. In the maximal sets Chapter, we presented a parallel approach for finding extremal sets within a family $\mathcal{F}$. We focused on the problem of finding the maximal sets of $\mathcal{F}$, and then we described the minor changes the algorithm needs so we can find the minimal sets. The proposed algorithm has no restriction on the input data. The presented approach finds the extremal sets of a family $\mathcal{F}$, even the repeated ones. That is, repeated extremal sets are found as many times as exists in $\mathcal{F}$. The algorithms for finding the maximal and the minimal sets can be slightly modified so the repeated sets of the output are discarded. We presented experimental results studying the algorithm response

depending on some characteristics of the input and output families for synthetic and real data sets. We also provided a comparison between the running times of our GPU algorithm against Bayardo et al. CPU algorithm, that showed that our algorithm is faster in all the cases.

In Chapter 6, we presented a parallel GPU-based approach, designed under CUDA architecture, for computing the intersection between two families of sets. This was, to the best of our knowledge, the first implementation of the problem even only using the CPU. The complexity analysis together with some experimental results were presented. Experimentally, we studied the algorithm response on different characteristics of the input and output families. Two very different kinds of synthetic families were considered, ones simulating the flocks problem, and the others with random uniformly distributed sets. The presented experimental results show the efficiency and scalability of the approach and the computational and memory savings reached with the different steps of the algorithm: the input domain reduction, and the empty and duplicated sets removal.

Finally, we presented a GPU-based algorithm, to compute flock patterns. We presented a new lemma which reduces the number of potential flocks to the half, with respect to a previous lemma. We discussed three variants of the flock pattern and we explained our algorithms to compute them. In the experimental results we compared our running times against the paper which, for the best of our knowledge, was the fastest algorithm to report flock patterns. We ran our algorithms with real data sets showing that our GPU algorithm is up to 150 times faster than the previous algorithms.

[1] P. Laube and S. Imfeld, "Analyzing Relative Motion within Groups of Trackable Moving Point Objects," in *GIScience* (M. J. Egenhofer and D. M. Mark, eds.), vol. 2478 of *Lecture Notes in Computer Science*, pp. 132–144, Springer, 2002.

[2] J. Gudmundsson, P. Laube, and T. Wolle, "Movement Patterns in Spatio-temporal Data," in *Encyclopedia of GIS* (S. Shashi and X. Hui, eds.), pp. 726–732, Springer, 2008.

[3] P. Laube, M. van Kreveld, and S. Imfield, "Finding REMO - Detecting Relative Motion Patterns in Geospatial Lifelines," *Developments in Spatial Data Handling: 11th Int. Sympos. on Spatial Data Handling*, pp. 201–215, 2004.

[4] J. Gudmundsson, M. van Kreveld, and B. Speckmann, "Efficient Detection of Motion Patterns in Spatio-Temporal Data Sets," in *GIS* (D. Pfoser, I. F. Cruz, and M. Ronthaler, eds.), pp. 250–257, ACM, 2004.

[5] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle, "Reporting flock patterns," *Computational Geometry*, vol. 41, no. 3, pp. 111–125, 2008.

[6] M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle, "Reporting Leaders and Followers among Trajectories of Moving Point Objects," *GeoInformatica*, vol. 12, no. 4, pp. 497–528, 2008.

[7] S. Dodge, R. Weibel, and A.-K. Lautenschutz, "Towards a taxonomy of movement patterns," *Information Visualization*, vol. 7, no. 3-4, pp. 240–252, 2008.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃŒger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[9] N. Coll, M. Fort, N. Madern, and J. A. Sellarès, "Multi-visibility maps of triangulated terrains," *International Journal of Geographical Information Science*, vol. 21, no. 10, pp. 1115–1134, 2007.

[10] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, (New York, NY, USA), pp. 34–42, ACM, 2009.

[11] M. Fort, J. A. Sellarès, and N. Valladres, "Computing popular places using graphics processors," in *Proc. SSTDM'10 in cooperation with IEEE ICDM'10*, pp. 233–241, IEEE Computer Society, December 2010.

[12] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2d grid using cuda," *J. Parallel Distrib. Comput.*, vol. 71, no. 4, pp. 615–620, 2011.

[13] V. Kolonias, A. G. Voyiatzis, G. Goulas, and E. Housos, "Design and implementation of an efficient integer count sort in cuda gpus," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 18, 2011.

[14] K. Kato and T. Hosino, "Multi-gpu algorithm for k-nearest neighbor problem," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 1, pp. 45–53, 2012.

[15] C. Schulz, "Efficient local search on the gpu - investigations on the vehicle routing problem," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 14–31, 2013.

[16] F. Pinel, B. Dorronsoro, and P. Bouvry, "Solving very large instances of the scheduling of independent tasks problem on the gpu," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 101–110, 2013.

[17] G. L. Andrienko and N. V. Andrienko, "A visual analytics approach to exploration of large amounts of movement data," in *VISUAL* (M. Sebillo, G. Vitiello, and G. Schaefer, eds.), vol. 5188 of *Lecture Notes in Computer Science*, pp. 1–4, Springer, 2008.

[18] U. Wilensky, "NetLogo." `http://ccl.northwestern.edu/netlogo`, 1999.

[19] M. Benkert, B. Djordjevic, J. Gudmundsson, and T. Wolle, "Finding popular places," *International Journal of Computational Geometry and Applications (IJCGA)*, vol. 20, no. 1, pp. 19–42, 2010.

[20] K. Siddiqi and S. Pizer, *Medial Representations: Mathematics, Algorithms and Applications*. Springer Publishing Company, Incorporated, 2008.

[21] A. Gajentaan and M. H. Overmars, "On a class of $o(n^2)$ problems in computational geometry," *Comput. Geom.*, vol. 45, no. 4, pp. 140–152, 2012.

[22] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., 2008.

[23] P. J. S. Leite, J. M. X. N. Teixeira, T. S. M. C. de Farias, V. Teichrieb, and J. Kelner, "Massively parallel nearest neighbor queries for dynamic point clouds on the gpu," in *SBAC-PAD*, pp. 19–25, IEEE Computer Society, 2009.

[24] X. Li, J. Han, J.-G. Lee, and H. Gonzalez, "Traffic density-based discovery of hot routes in road networks," in *SSTD* (D. Papadias, D. Zhang, and G. Kollios, eds.), vol. 4605 of *Lecture Notes in Computer Science*, pp. 441–459, Springer, 2007.

[25] D. Sacharidis, K. Patroumpas, M. Terrovitis, V. Kantere, M. Potamias, K. Mouratidis, and T. K. Sellis, "On-line discovery of hot motion paths," in *EDBT* (A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, and I. Manolescu, eds.), vol. 261 of *ACM International Conference Proceeding Series*, pp. 392–403, ACM, 2008.

[26] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory pattern mining," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pp. 330–339, ACM, 2007.

[27] A. T. Palma, V. Bogorny, B. Kuijpers, and L. O. Alvares, "A clustering-based approach for discovering interesting places in trajectories," in *SAC* (R. L. Wainwright and H. Haddad, eds.), pp. 863–868, ACM, 2008.

[28] Y. Theodoridis, "R-Tree portal." `http://www.rtreeportal.org`, 2011.

[29] C. Dartmouth, "CRAWDAD." `http://crawdad.cs.dartmouth.edu/index.php`, 2008.

[30] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo, "Detecting commuting patterns by clustering subtrajectories," *Int. J. Comput. Geometry Appl.*, vol. 21, no. 3, pp. 253–282, 2011.

[31] O. W. H. Cao and G. Trajcevski, "Spatio-temporal data reduction with deterministic error bounds," *The VLDB Journal*, vol. 15, no. 3, pp. 211–228, 2006.

[32] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle, "Compressing spatio-temporal trajectories," *Computational Geometry - Theory and Applications*, vol. 42, no. 9, pp. 825–841, 2009.

[33] K. Buchin, M. Buchin, and J. Gudmundsson, "Constrained free space diagrams: a tool for trajectory analysis," *International Journal of Geographical Information Science*, vol. 24, no. 7, pp. 1101–1125, 2010.

[34] H. Alt and M. Godau, "Computing the fréchet distance between two polygonal curves," *Int. J. Comput. Geometry Appl.*, vol. 5, pp. 75–91, 1995.

[35] H. Alt, C. Knauer, and C. Wenk, "Comparison of distance measures for planar curves," *Algorithmica*, vol. 38, no. 1, pp. 45–58, 2003.

[36] M. Vlachos, D. Gunopoulos, and G. Kollios, "Discovering similar multidimensional trajectories," in *Proceedings of the 18th International Conference on Data Engineering*, pp. 673–682, 2002.

[37] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory clustering: a partition-and-group framework," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 593–604, 2007.

[38] M. Nanni and D. Pedreschi, "Time-focused clustering of trajectories of moving objects," *J. Intell. Inf. Syst.*, vol. 27, no. 3, pp. 267–289, 2006.

[39] J. Gudmundsson, P. Laube, and T. Wolle, "Movement analysis," in *Handbook of Geographic Information* (W. Kresse and D. M. Danko, eds.), ch. 22, pp. 725–741, Springer, 2012.

[40] H. Alt, *In Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, ch. The computational geometry of comparing shapes, pp. 235–248. Springer-Verlag, 2009.

[41] A. Driemel, S. Har-Peled, and C. Wenk, "Approximating the fréchet distance for realistic curves in near linear time," in *Symposium on Computational Geometry*, pp. 365–374, 2010.

[42] P. K. Agarwal, R. B. Avraham, H. Kaplan, and M. Sharir, "Computing the discrete fréchet distance in subquadratic time," *CoRR*, vol. abs/1204.5333, 2012.

[43] S. Gaffney, A. Robertson, P. Smyth, S. Camargo, and M. Ghil, "Probabilistic clustering of extratropical cyclones using regression mixture models," *Climate Dynamic*, vol. 29, no. 4, pp. 423–440, 2007.

[44] S. Gaffney and P. Smyth, "Trajectory clustering with mixtures of regression models," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 63–72, 1999.

[45] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. Cheung, "Mining, indexing, and querying historical spatiotemporal data," in *Proceedings of the 10th International ACM Conference On Knowledge Discovery and Data Mining*, pp. 236–245, ACM, 2004.

[46] J. Gudmundsson and M. J. van Kreveld, "Computing longest duration flocks in trajectory data," in *GIS* (R. A. de By and S. Nittel, eds.), pp. 35–42, ACM, 2006.

[47] N. Corporation, "Nvidia cuda 4.1 sdk samples." `http://developer.nvidia.com/gpu-computing-sdk`, 2012.

[48] Amisco, "www.amisco.eu." [Online; accessed 23-Jan-2013].

[49] www.tracab.com, "[online; accessed 23-jan-2013]."

[50] ProZone, "www.prozonesports.com." [Online; accessed 23-Jan-2013].

[51] R. J. Bayardo and B. Panda, "Fast algorithms for finding extremal sets," in *SDM*, pp. 25–34, SIAM / Omnipress, 2011.

[52] P. Pritchard, "Opportunistic algorithms for eliminating supersets," *Acta Inf.*, vol. 28, no. 8, pp. 733–754, 1991.

[53] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," in *SAT* (F. Bacchus and T. Walsh, eds.), vol. 3569 of *Lecture Notes in Computer Science*, pp. 61–75, Springer, 2005.

[54] T. Mielikäinen, P. Panov, and S. Dzeroski, "Itemset support queries using frequent itemsets and their condensed representations," in *Discovery Science* (L. Todorovski, N. Lavrac, and K. P. Jantke, eds.), vol. 4265 of *Lecture Notes in Computer Science*, pp. 161–172, Springer, 2006.

[55] M. R. Vieira, P. Bakalov, and V. J. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *GIS* (D. Agrawal, W. G. Aref, C.-T. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, and O. Wolfson, eds.), pp. 286–295, ACM, 2009.

[56] D. M. Yellin and C. S. Jutla, "Finding extremal sets in less than quadratic time," *Inf. Process. Lett.*, vol. 48, no. 1, pp. 29–34, 1993.

[57] P. Pritchard, "A simple sub-quadratic algorithm for computing the subset partial order," *Inf. Process. Lett.*, vol. 56, no. 6, pp. 337–341, 1995.

[58] H. Sheni and D. J. Evans, "Fast sequential and parallel algorithms for finding extremal sets," *International Journal of Computer Mathematics*, vol. 61, no. 3-4, pp. 195–211, 1996.

[59] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in *SC*, pp. 666–675, 1990.

[60] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[61] B. Hoffmann, M. Lifshits, Y. Lifshits, and D. Nowotka, "Maximal intersection queries in randomized input models," *CoRR*, vol. abs/1004.0092, 2010.

[62] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA* (D. B. Shmoys, ed.), pp. 743–752, ACM/SIAM, 2000.

[63] J. Barbay and C. Kenyon, "Adaptive intersection and t-threshold problems," in *SODA* (D. Eppstein, ed.), pp. 390–399, ACM/SIAM, 2002.

[64] J. Barbay, "Optimality of randomized algorithms for the intersection problem," in *SAGA* (A. A. Albrecht and K. Steinhöfel, eds.), vol. 2827 of *Lecture Notes in Computer Science*, pp. 26–38, Springer, 2003.

[65] R. A. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *CPM* (S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, eds.), vol. 3109 of *Lecture Notes in Computer Science*, pp. 400–408, Springer, 2004.

[66] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Experiments on adaptive set intersections for text retrieval systems," in *ALENEX* (A. L. Buchsbaum and J. Snoeyink, eds.), vol. 2153 of *Lecture Notes in Computer Science*, pp. 91–104, Springer, 2001.

[67] R. A. Baeza-Yates and A. Salinger, "Experimental analysis of a fast intersection algorithm for sorted sequences," in *SPIRE* (M. P. Consens and G. Navarro, eds.), vol. 3772 of *Lecture Notes in Computer Science*, pp. 13–24, Springer, 2005.

[68] J. Barbay, A. López-Ortiz, and T. Lu, "Faster adaptive set intersections for text searching," in *WEA* (C. Àlvarez and M. J. Serna, eds.), vol. 4007 of *Lecture Notes in Computer Science*, pp. 146–157, Springer, 2006.

[69] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang, "A batched gpu algorithm for set intersection," in *ISPAN*, pp. 752–756, IEEE Computer Society, 2009.

[70] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu, "Efficient lists intersection by cpu-gpu cooperative computing," in *IPDPS Workshops*, pp. 1–8, IEEE, 2010.

[71] R. R. Amossen and R. Pagh, "A new data layout for set intersection on gpus," in *IPDPS*, pp. 698–708, IEEE, 2011.

[72] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *PVLDB*, vol. 4, no. 8, pp. 470–481, 2011.

[73] R. Paige, "Efficient translation of external input in a dynamically typed language," in *IFIP Congress (1)*, pp. 603–608, 1994.

[74] F. Henglein, "Generic top-down discrimination for sorting and partitioning in linear time," *J. Funct. Program.*, vol. 22, no. 3, pp. 300–374, 2012.

[75] J. Gudmundsson, M. J. van Kreveld, and B. Speckmann, "Efficient Detection of Patterns in 2D Trajectories of Moving Points," *GeoInformatica*, vol. 11, no. 2, pp. 195–215, 2007.

[76] M. Wachowicz, R. Ong, C. Renso, and M. Nanni, "Finding moving flock patterns among pedestrians through collective coherence," *International Journal of Geographical Information Science*, vol. 25, no. 11, pp. 1849–1864, 2011.