Efficient large-scale graph processing: optimisations for storage, performance and evolving graphs

Thèse N° 9921

Présentée le 20 décembre 2019

à la Faculté informatique et communications
Laboratoire de systèmes d'exploitation
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Jasmina MALICEVIC

Acceptée sur proposition du jury

Prof. J.-D. Decotignie, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Prof. R. Chen, rapporteur
Prof. E. Yoneki, rapporteuse
Prof. A. Ailamaki, rapporteuse

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

"Yesterday I was clever, so I wanted to change the world.
Today I am wise, so I am changing myself."

"Two there are who are never satisfied:
the lover of the world and the lover of knowledge."
— Rumi

To my dearest, wonderful and supporting family...

Acknowledgements

I am very fortunate to have had amazing people by my side during this journey. First, I want to thank my advisor, *Prof. Willy Zwaenepoel*, for all his advise, support and guidance throughout the years. He shaped the way I approach problems, presentations and paper writing. I learned how to argue for my ideas, how to defend my results, the importance of being precise and concise when expressing my ides. Most of all, I am thankful to him for always knowing exactly when I needed space to find my path, and when I needed a little push.
I wish to thank my thesis committee, *Prof. Anastasia Ailamaki, Prof. Rong Chen and Prof. Eiko Yoneki* for their work in reviewing this thesis. All their comments during and after the defence have strengthen the thesis. I would also like to thank *Prof. Jean-Dominique Decotignie* for presiding over my thesis, and all the advise and help he was always happy to offer.

I will be forever indebted to *Amitabha Roy*. Amitabha guided me through my first year and introduced me to the world of systems research. His patience, support and faith in my projects were a boost to my self-confidence. He was always there to discuss ideas and problems, even after he left EPFL.

In addition to Amitabha, I want to thank my entire LABOS family for all the lab lunches, pétanque and board games, hikes, and in general their open doors for both problems and celebrations. Thank you *Pamela, Kristina, Maria, Laurent, Calin, Maciej, Peter, Baptiste, Florin and Oana.* And of course, Madeleine, who always had a solution to all my complicated requests. Thank you *Mia* for making conferences and EPFL fun, and to my friends in Lausanne who made Switzerland feel more like home, especially Zarko and Ana.

Throughout this journey I had support not only from people at EPFL or in Switzerland. These years were filled with love and support from my friends from Serbia, my in-laws, and my entire family. Thank you for cheering for me, for sending me delicious food, and being understanding when I could not make it to all the gatherings.

I am deeply grateful to my parents, *Behdzida* and *Nermin* for teaching me the true values of life, for always supporting me in all my decisions. Thank you for letting me take any path I wanted, no matter how far it was from where you are. Being a parent now, I see how hard that is, yet I never saw it on you. Thank you mom, for showing up before all my deadlines to take care of Uma so I can concentrate on work. I wish to thank my *siblings*, my best friends and

biggest cheerleaders. I am grateful and happy to have been able to share my life with you.

There are no words to thank my husband, *Edin*, for his love, for being my partner throughout the years, for holding down the fort in these last months, his endless understanding, our daughter *Uma*. Her presence and joy made all the hard days look easy. Being a mother to her gives a much deeper meaning to everything I do. Thank you both for everything.

Lausanne, December 5, 2019 Jasmina Malicevic

Abstract

Graph processing systems are used in a wide variety of fields, ranging from biology to social networks. Algorithms to mine graphs incur many random accesses, and the sparse nature of the graphs of interest, exacerbates this. As DRAM sustains high bandwidth in the presence of random accesses, traditionally, it was the chosen medium to store and process graphs from.
Many in-memory systems were presented at top tier conferences, and every system compared to the previous in algorithm execution time. What is not clearly evaluated are the overheads of pre-processing the input in order to support particular optimisations. More critical, for a systems designer, it is very hard to reason about what are the fundamental techniques that lead to performance gains.
We implement the proposed optimisations of state-of-the-art systems within one system to answer this question. We found that the pre-processing cost often dominates the cost of algorithm execution, calling into question the benefits of proposed algorithmic optimisations that rely on extensive pre-processing. The design of a system has to be carefully guided by the characteristics of the algorithms, graphs and hardware.
Furthermore, in-memory systems are often limited by the amount of available memory on a single machine. When the graph cannot fit in the available DRAM, many systems scale up to secondary storage on one machine, or in the cloud. As storage is cheaper than memory, they trade off performance for more space, at a lower cost. Emerging non-volatile technologies such as 3D XPoint, offer a new opportunity for bridging this performance gap. Designing a system that fully utilises these storage devices is not straightforward. Traditional I/O optimisations, designed for SSDs, underutilise the device, and systems end up being CPU bound on fast storage. By removing the dedicated I/O layers, and combining pre-processing approaches for in-memory and out-of-core systems, we are able to switch graph representations to benefit a particular algorithm. This improves the end-to-end execution time up to 2×.
However, in the presence of updates to the graph structure, the data structures used by static algorithms need to be re-created on every update, and the algorithm executed from scratch. The high latency of this process is not acceptable for critical applications such as fraud detection.
We address this problem by developing two systems to compute on evolving graphs, using an update friendly graph representation. Each system targets a different group of applications: graph analytics and graph pattern mining (GPM). For graph analytics we trade sub-millisecond latencies for consistency. For GPM, we use re-computation and backtracking to handle millions of updates per second, outperforming state of the art by up to 5×.

Abstract

Keywords: *graph analytics, storage, dynamic graphs, Big Data, non-volatile memory, cloud computing, survey*

Résumé

Les systèmes de traitement de graphes sont utilisés dans une grande quantité de domaines, de la biologie aux réseaux sociaux. Les algorithmes utilisés pour l'exploration des graphes encourent beaucoup d'accès aléatoires, et la nature clairsemée des graphes d'intérêt, exacerbent ce problème. Étant donné que la mémoire vive dynamique (DRAM) est capable de maintenir une bande passante élevée en présence d'accès aléatoires, il s'agit, traditionnellement, du milieu de choix pour stocker et traiter les graphes.

Beaucoup de systèmes en mémoire ont été présentés à des conférences de premier plan, et chaque système a été comparé avec les précédents en termes de temps d'exécution d'algorithme. Ce qui n'est pas clairement évalué sont les frais supplémentaires liés au prétraitement de l'entrée afin de pouvoir supporter certaines optimisations. Encore plus critique, pour un concepteur de systèmes, est qu'il est très difficile de raisonner sur les techniques fondamentales qui conduisent à des gains de performance.

Nous implémentons les optimisations proposées par les systèmes de l'état de l'art à l'intérieur d'un seul système afin de répondre à cette question. Nous avons constaté que les coûts de prétraitement dominent souvent les coûts d'exécution d'algorithmes, ce qui remet en question les bénéfices des optimisations algorithmiques proposées qui reposent sur un prétraitement approfondi. La conception d'un système doit être soigneusement guidée par les caractéristiques des algorithmes, graphes, et du matériel.

De plus, les systèmes en mémoire sont souvent limités par la quantité de mémoire disponible sur une seule machine. Lorsque le graphe ne peut pas tenir dans la DRAM disponible, beaucoup de systèmes évoluent vers le stockage secondaire d'une seule machine ou vers le cloud. Comme le stockage est moins cher que la mémoire, ils échangent leur performance pour plus d'espace à un moindre coût. Les technologies non-volatiles émergentes telles que 3D XPoint offrent une nouvelle opportunité de combler cet écart de performance. Concevoir un système efficace qui utilise pleinement ces périphériques de stockage n'est pas simple. Les optimisations d'entrées/sorties (E/S) traditionnelles, conçues pour les disques durs à état solide (SSDs), sous-utilisent le périphérique et les systèmes finissent par limités par l'unité centrale de traitement (CPU) si le stockage est suffisamment rapide. En supprimant les couches d'E/S dédiées et en combinant des approches de prétraitement pour les systèmes en mémoire et les systèmes hors cœur, nous sommes en mesure de changer la représentation du graphe pour tirer parti d'un algorithme particulier. Cela permet des gains pouvant aller jusqu'à 2x dans le temps d'exécution.

Cependant, en présence de mises à jour de la structure du graphe, les structures de données

utilisées par les algorithmes statiques doivent être recréées à chaque mise à jour, et l'algorithme doit être exécuté en partant de zéro. La latence élevée de ce processus n'est pas acceptable pour des applications critiques telles que la détection de fraude.
Nous abordons ce problème en développant deux systèmes de traitement de graphes en évolution, en utilisant une représentation du graphe conviviale pour les mises à jour. Chaque système cible un groupe d'applications différent : le traitement analytique de graphes et l'exploration de modèles dans les graphes (GPM). Pour le traitement analytique de graphes, nous échangeons des latences inférieures à une milliseconde contre la consistance des données. Pour GPM, nous utilisons le recalcul et le retour en arrière pour gérer des millions de mises à jour par seconde, dépassant les performances de l'état de l'art jusqu'à 5x.
Mots-clés : *traitement analytique de graphes, stockage, graphe dynamiques, mégadonnées, mémoire non-volatile, informatique en nuages, étude*

Contents

List of Figures

List of Tables

1 Introduction

Over the years, graph analytics have emerged as an important sub-problem of the big data problem. Graphs are an intuitive and natural way to represent arbitrary relationships and connections between objects, people, sensor data, cities, etc. As they are present in many different areas, general purpose big data systems such as Map-reduce have added support for graph analytics. However, graphs have different characteristics than the data typically processed by these systems. The API of general purpose big data systems such as Spark [137] or Apache Hadoop is not expressive enough for graph analytics, making graph-specific optimisations hard to implement. This often leads to poor performance of these systems when the input is structured as a graph, resulting in a different representation of relationships naturally captured by a graph.

As the amount of data being generated grows, the great challenge is not only how to store this data, but to extract meaningful information out of it. To put things in perspective, according to a study in [15], the amount of data copied and created by 2020 will be 44 zettabytes.

The publication of Google's Pregel system [86] in 2010 was a shifting moment for using graphs in large-scale analytics. For the first time, it was shown that it is possible to store and query a graph with billions of edges and vertices, at scale. In fact, at the time, Pregel was the backbone of Googles' search engine. Graphs soon found their place at the heart of recommender systems (Netflix), social networks (Facebook, Twitter), road networks, web analytics, fraud detection, and many more.

Graph analytics have been an important topic in the research community as well, with many publications at top-tier conferences [18, 34, 35, 37, 55–57, 59, 61, 78, 85, 114, 119, 121, 144, 149].

The graphs of interest in these communities are so called natural graphs, with the following features:

- They are sparse; not every vertex is connected to every other vertex.
- A small number of vertices is connected to a very large part of the graph, while most of the vertices have few connections. The former are called high degree or hot vertices.

Figure 1.1 shows the power-law distribution of out-degrees in the Twitter follower graph. This is why they are also called power-law graphs.

- "Six degrees of separation" [22, 123]: A vertex can reach any other vertex in at most six hops. The papers demonstrate that any Facebook user, connects to any other random user by at most six hops, while popular accounts are four hops away from any other user.

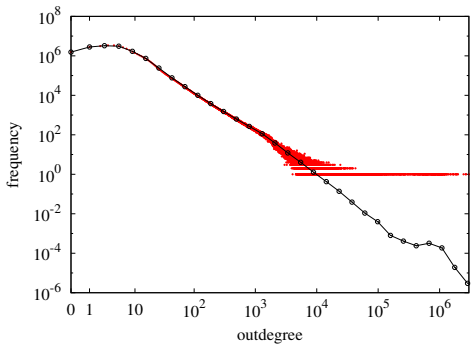


Figure 1.1 – The out degree distribution of the Twitter 2010 follower graph [28, 77]

The arbitrary connections between objects in a graph result in a random and unpredictable access pattern of graph algorithms. The skewed power-law distribution of edges only exacerbates this behaviour, making graph analytics a very challenging problem.

Due to the random accesses in graph algorithms, traditionally, many systems used DRAM to store and process graphs. When the size of the graph exceeds the available DRAM, in-memory systems scale out to many machines [34, 55, 147].

An alternative to scaling out is scaling up to secondary storage. For this to be feasible, the programming model and graph algorithm have to be adapted to reduce the random accesses. Some techniques used in state-of-the-art out-of-core systems [18, 78, 85, 113, 114, 149] are streaming big chunks of the graph at a time, at the cost of reading in more data than needed; introducing a caching layer between the storage and CPU, that merges and schedules small requests; asynchronous computation to reduce the number of iterations of the algorithm.

At a high level, all out-of-core systems assume the storage to be the bottleneck, and try to optimise disk I/O at the expense of more computation or limited data access patterns.

However, new storage technologies such as the 3D XPoint have significantly lower latencies and higher bandwidth than traditional storage devices. In this thesis we look to leverage these features for graph processing, with the goal to reduce cost, speed up graph processing and process larger graphs on one machine. More specifically, we look at two different classes of these technologies: PCIe NVMe SSDs and byte addressable NVM technologies.

PCIe NVMes are able to read and write data at high speed (2.6GB/s), and are able to maintain that speed doing random IO at a page granularity (see Section 5.1). NVMes are now closer

in performance to DRAM than to hard drives. This observation begs the following question: should NVMes be treated as storage or would they benefit more from optimisations proposed for in-memory (DRAM) graph processing?

On the other hand, byte addressable NVMe technologies are closer to the CPU, and faster than PCIe attached storage. However, they are less durable than DRAM, and are expected to offer a tradeoff between lower latency but lower throughput, or higher throughput at the cost of an increased latency. It is important to understand how this impacts the memory requirements of graph analytics.

Evolving graphs

Pregel was introduced with the purpose to run the Google website ranking algorithm - Pagerank [29]. The algorithm was run offline, periodically, on the entire webgraph. The same principle was used in recommender systems such as Netflix or the Pandora music engine. The results of the analytics were then used to correct and update rankings for a user. When a new movie was added, or the user changed its ranking, it would not be immediately reflected, until the analytics were run again, in the background.

However, with the increase of available graph data, the size of the graphs grows as well, and re-running analytics on the entire graph for every change in the graph is very costly. A more latency sensitive example is fraud detection in online shopping or banking. People trying to commit fraud typically form some kind of connection through third parties. Detecting this connection and flagging suspicious activity as soon as possible is crucial.

A few systems have attempted incremental analytics over graphs [35, 66, 95, 116] for algorithms such as Breadth-first search and Pagerank. They typically batch updates to the graph and then apply the whole batch of updates at once. This allows them to handle reasonable update ingestion rates, albeit at the expense of increased latency while the batch is being collected. In an orthogonal development, systems for general-purpose incremental processing have been proposed [31, 95, 108], and some have been applied to the analysis of graphs, but because of their general-purpose nature, they cannot take advantage of the specifics of graph processing. We present a system that processes graphs with a significantly lower latency, by applying the updates as they arrive, while the algorithm is being executed. We track dependencies between vertices to determine which parts of the graph are affected by an update.

In addition to graph analytics, another set of algorithms is of equal importance: graph pattern mining algorithms.

Graph pattern mining (GPM) algorithms help discover complex structural patterns in graphs, enabling wide-ranging applications, such as discovering chemical interactions or 3D protein structures [38, 109], analysing communities in social networks [53], mining frequent motifs in networks [93], analysing semantic data [96], or detecting suspicious credit card transactions [107]. Examples of GPM algorithms include motif counting [19], frequent subgraph

mining (FSM) [63], graph keyword search [68, 128], and clique or diamond mining [30, 58].

Several systems have been developed for running these mining algorithms on single nodes [26, 49, 60], on distributed infrastructure [118, 122, 127, 133], or using out-of-core processing [129]. These systems are designed to perform offline processing of *static* graphs. However, real-world graphs are large, and mining algorithms are computationally expensive because the number of possible subgraphs matching a pattern can be exponential in the size of the original graph. For example, a graph that takes 500MB to store, results in 148GB of 4-cliques. As a result, executing static algorithms from scratch on an updated graph, especially when a small subset of the graph has changed, is prohibitively expensive. Furthermore the dependency tracking applied to graph analytics applications is unfeasible for this class of algorithms.

These algorithms require a different programming model compared to graph analytics algorithms. To this end, this thesis presents a system that supports incremental graph pattern mining on evolving graphs, without explicit dependency tracking, but rather by entirely re-computing the patterns impacted by an edge. As the update propagates only locally, this approach has superior performance to re-computing on the entire graph. In this scenario we batch the updates to avoid expensive re-computation in the same neighbourhood, and perform a staged execution of first applying the updates, and then executing the algorithm.

1.1 Motivation and challenges

Due to the large number of graph processing systems presented, it is very hard to reason about which system to use when. A new system typically compares its algorithm execution time with the algorithm execution time of what is believed to be the state-of-the-art at the time.
We observe that this simplistic approach is not enough to clearly understand why is one system better than the other. They all use different data representations, programming models, rely on different hardware, run different algorithms etc.
Furthermore, many of the optimisations that are the key contributor to the good performance of graph processing systems, come at an expensive pre-processing step. Some systems rely on the availability of a large amount of DRAM, or storage space. It is not clear from their evaluation what are the limitations of their systems and when another system should be used.

This was the key motivation behind the work in Chapters 3 and 5. The work presented in the chapters demonstrate how important it is to understand what techniques should be applied when, and when do the resources available to the system designer have to guide the design of the system.
Thorough understanding of the optimisations and the execution environment becomes even more important when running on specialised hardware like byte addressable NVMes (Chapter 6) or network attached storage in the Cloud (Chapter 4).
The insights gained from evaluating the state-of-the-art enabled us to design a more efficient graph engine, that utilises the underlying hardware more efficiently.

However, many of these optimisations were implemented and evaluated only in the context of static graphs. The graph representations used by the systems are not easily updated, and the algorithms do not support incremental computation by default. The dynamic nature of the graphs of interest, and ever growing user expectations for a quick result, require graph processing systems to support processing evolving graphs quicker and more efficiently.

We extend the scope of the thesis to dynamic graph processing, implementing two systems, each for a different group of graph algorithms. We improve on the state-of-the-art by achieving lower latency, using less DRAM, and computing on much bigger graphs using fewer resources.

While implementing both the static and dynamic systems we faced a number of challenges presented in the following section.

1.1.1 Challenges

Load imbalance during computation. When assigning work to threads, the amount of work done by a thread can differ vastly. Consider for example a graph represented as an adjacency list, where each thread is assigned an equal number of vertices to process. Processing a vertex involves reading its neighbours and updating them. Since the number of neighbours per vertex varies by orders of magnitude, this simple division of work will leave many threads idle. In memory, with bad work partitioning, for the Twitter graph, threads were idle 70% of the time. In Chapter 3 we show how this can be mitigated by an adaptive work assignment scheme.
This problem is exacerbated in distributed and out-of-core systems running on fast storage. In distributed systems, load imbalance has a higher cost as idle workers are more expensive than idle threads on one machine. Furthermore, with bad data distribution the amount of network traffic is increased as well.
In out-of-core systems, fetching the neighbours of a vertex translates to issuing I/O requests. Load imbalance means that there are fewer outstanding requests at one point, leaving the device under-utilised. Chapter 5 demonstrates the significant impact load balancing has on performance and device utilisation.

To address this issue, systems need to start with a fair division of work among threads/workers based on the number of edges they need to process. However, in many algorithms, different parts of the graph are active at different times, leading to the initial load balancing to be wrong. Thus, there is a need for adaptive load balancing, or work stealing during computation.

Matching graph representation and algorithm. Graph algorithms have different characteristics. Traversal algorithms, like Breadth-first search (BFS), only look at a few vertices per iteration. On the other hand, global algorithms, like Pagerank, update the entire graph at every iteration.
For traversal algorithms, a system should be able to access only the relevant parts of the graph. For example, when BFS starts running, it requires the neighbours of only one vertex - the BFS root, followed by their neighbours in the second iteration. This is less than 1% of the graph.

Streaming the entire graph in this case is a huge overhead. Adjacency lists provide a way to access the graph at such fine granularity. With this representation, the edges are sorted by source (or destination), and each vertex has a pointer to the list of its neighbours. Fetching the neighbours of only one vertex is now trivial.

On the other hand, when the algorithm reads most of the graph, there is no need for this pointer chasing as we anyways read all the edges. For these algorithms, any delay in fetching the data increases the running time of the algorithm.
In DRAM, this extra level of indirection leads to a higher cache miss rate. Out-of-core, on fast storage devices, it delays I/O requests, and is an overhead amortised only in traversal algorithms.
We address the different implications of various graph representations on algorithm running time in Chapters 3 and 5.

Pre-processing is not free. Different systems and programming models rely on different graph representations. A particular graph layout can enable many optimisations specific to this representation, leading to its superior performance. However, the cost of creating this representation from raw input can often be higher than the algorithm execution time over the raw input (Chapter 3). This is especially visible out-of-core, where the penalty for bad design decisions is much higher (Chapter 5).

Different programming models needed for different hardware. Graph analytics were initially ran from DRAM, and the programming models designed accordingly. However, we show in Chapters 5 and 6 that simply running this code out-of-core is not feasible. It is important to understand the specifics of the underlying hardware such as durability (Chapter 6), bandwidth and latency (Chapters 4,5, 6).

In-memory graph representation that supports evolving graphs. Graphs that change over time are not easily supported by the data layouts used in state-of-the-art graph processing systems. The default representations, being optimised for the static scenario, do not support all update operations.
Adding one edge to an edge array is trivial. But deleting a particular edge requires scanning the entire graph in order to find this edge. While the lookup is quick when the graph is represented as an adjacency list, adding an edge requires reallocating all edges succeeding the added edge and updating offsets to all vertices with an ID higher than the source ID of that edge. In Chapter 7, we present an update-friendly graph representation with comparable performance to a highly optimised static representation.

Algorithm correctness when graphs evolve. In addition to updating the actual graph data structure, it is important to trigger *re-computation only on the part of the graph impacted by an update*. In Chapter 7 we address this issue for graph analytics applications. Based on their execution model, we differentiate between two groups of graph analytics algorithms: monotonic (traversal) and sparse-matrix multiplication (always converging) algorithms. For traversal algorithms, it is essential to keep track of dependencies in order to determine the

path along which an update is to be propagated. For always converging algorithms, tracking dependencies is more complex, but without dependencies, the intermediary state of the algorithm is inconsistent.
However, keeping track of dependencies in graph pattern mining applications is completely unfeasible due to the large number of patterns that need to be tracked. We show how to maintain patterns matching a query in case of updates in Chapter 8.

Many updates can impact the same subgraph. In graph pattern mining, a pattern is a subgraph of the graph formed by a set of edges or vertices. This pattern can be discovered from any of the edges/vertices it contains. If more than one of them is updated within one batch, simply triggering re-computation on both will lead to a pattern being discovered twice. We introduce a unique exploration order via update canonicality rules (Chapter 8) to resolve conflicts without synchronisation among threads.

1.2 Thesis statement and contributions

The research contributions in this thesis can be divided in two parts - static and dynamic graph processing.

We address the challenges faced by static graph processing systems, both in memory and out-of-core and contribute the following:

A roadmap for designing an efficient in-memory graph analytics system. We answer the question of what techniques used by state-of-the-art graph processing systems work for what algorithms and what graphs. Additionally, we evaluate whether hardware specific performance optimisations are always beneficial. We implement all the techniques within one system in order to perform an apples-to-apples comparison.

Improved pre-processing techniques. Different graph representations allow for optimisations in the algorithm execution time, but the cost to create them is often not amortised. We improve on the pre-processing time of existing systems, and provide an analysis on which data representation should be used in different scenarios.

An analysis on the feasibility of graph analytics from secondary storage in the Cloud. We analyse the cost benefits of running graph analytics applications in the Cloud, when the graph is stored on secondary storage, when processing on a single machine, and in a distributed setting. We identify the network to be the bottleneck and offer the following to mitigate this problem: compression of I/O requests, network provisioning, and combining messages destined for the same vertex, to reduce the amount of I/O and network packages in distributed systems.

A system that uses state-of-the-art PCIe NVMe devices efficiently. State-of-the-art out-of-core systems are designed for HDDs or SSDs. Emerging devices such as PCIe NVMes have different characteristics, warranting a rethinking of the design choices of previous systems. We design Optimus, which outperforms state-of-the-art systems, by using the device efficiently and sustaining a much higher bandwidth. Applying the lessons learned from in-memory systems, we optimise the pre-processing step for out-of-core graph processing as well. This allows for transformations of the graph into different layouts, depending on the algorithm.

A study on the feasibility of byte addressable NVMe for graph processing. Emerging byte addressable non volatile memories, are a promising alternative to storing large amounts of data out-of-core. They provide latencies closer to DRAM while offering persistence. We analyse the impact of their design features on state-of-the-art graph processing systems, and offer directions on future development of systems for this type of storage.

To support changes in the graph this thesis provides the following contributions:
An update friendly graph representation. To support graph updates, we optimise for the most common operation - edge lookup. When using a dynamic graph representation, our system is $1.2\times - 2.7\times$ slower than an optimised static solution. However, the static computation

requires a pre-processing step, which in end-to-end time results in it being 3× slower. As the data structures of the system are not updatable, this step is required whenever there is an update to the graph.

An incremental graph engine to support graph analytics. For a response with low latency, we allow the system to change the graph structure as the algorithm executes. We then update the affected parts of the graph, or invalidate previous results to provide the latest results to the user.

A distributed incremental graph processing engine for graph mining applications. The system has a low memory footprint, uses re-computation to output the changes to the user, and offers coordination free parallel graph mining ensuring there are no duplicates.

In light of these contributions this thesis makes the following statements:

Thesis statements

Systems overlook the cost of pre-processing, and focus on the benefits achieved by optimisations in the compute time. However, by improving the pre-processing time as well, they can adapt the data structure to better suit different algorithms and graphs, use the underlying resources more efficiently and in end-to-end time significantly improve the performance of in-memory and out-of-core systems.

For graph analytics applications, applying updates while the algorithm is executed achieves low latency while only introducing brief inconsistencies.

We use backtracking and re-computation to support high throughput updates for graph pattern mining applications. We define a total order of exploration to prevent duplicate discoveries and prune the search space to reduce the number of explorations.

1.3 Publications

Some of the results described in Section 1.2 have been previously published:

1. J. Malicevic; A. Roy; W. Zwaenepoel : Scale-up Graph Processing in the Cloud: Challenges and Solutions. 2014. *CloudDP'14: Fourth International Workshop on Cloud Data and Platforms*, Amsterdam, Netherlands, April 13-16,2014 (Chapter 4)

2. (*)A. Roy; L. Bindschaedler; J. Malicevic; W. Zwaenepoel : Chaos: Scale-out Graph Processing from Secondary Storage. *25th Symposium on Operating Systems Principles*, Monterey, California, USA, October 3-7, 2015 (Chapter 4)

3. J. Malicevic; S. Dulloor; N. Sundaram; N. Satish; J. Jackson et al. : Exploiting NVM in Large-scale Graph Analytics. *3rd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, Monterey, California, USA, October 3-7 (Chapter 6)

4. J. Malicevic; B. J. E. Lepers; W. Zwaenepoel : Everything You Always Wanted to Know about Multicore Graph Processing but Were Afraid to Ask. *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, California, USA, July 12-14, 2017 (Chapter 3) **Best paper award**

The work presented in Chapter 5 is under submission:

- J. Malicevic, B. Lepers, S. Dwarkadash, W. Zwaenepoel. Optimus: Transforming for efficient single machine NVMe based out-of-core graph processing. *18th USENIX Conference on File and Storage Technologies (FAST 2020)*

and the work in Chapter 8 is currently under submission:

- (**) J. Malicevic, L. Bindschaedler, B. Lepers, A. Goel, W. Zwaenepoel. Tesseract: Fast, Scalable Graph Pattern Mining on Evolving Graphs (EuroSys 2020).

[*]In Chaos, I contributed to the implementation and evaluation of the compression of network packages, and I/O requests. I also implemented message combiners, to reduce the number of messages being sent through the network, and to storage.

[**]Tesseract was done in equal parts by Laurent Bindschaedler and myself. I contributed to the following: the storage backend of Tesseract, prevention of the discovery of the same pattern from multiple edges added in **the same batch**, the pre-filter to prune search space during explorations, and implemented the re-computation on updates, thus removing the need to store the patterns.
Laurent defined the new exploration algorithm and new update canonicality rules. These rules enable quick retrieval of all patterns from one edge without the need to test all possible combinations, and guarantee unique discovery **between batches**. He defined the pattern cache, the streaming interface and API, and integrated Tesseract with Spark, to provide distributed execution and fault tolerance.

[***] The work presented in Chapter 7 was done in collaboration with Baptiste Lepers, a postdoc in LABOS. He helped shape the work into a paper. He also occasionally helped with coding, especially with improving the performance of the update-able graph representation.

In addition to the work presented in this thesis, I also worked on the following publication, where I implemented the graph algorithms on top of the system, and ran the experiments on our servers:

1. L. Bindschaedler; J. Malicevic; N. Schiper; A. Goel; W. Zwaenepoel : Rock You like a Hurricane: Taming Skew in Large Scale Analytics. 2018-04-23. *The European Conference on Computer Systems (Eurosys '18)*, Porto Portugal, April 23-26, 2018

1.4 Thesis outline

The first part of the thesis describes techniques to process static graphs stored on different types of storage.

In **Chapter 3** presents techniques used by state-of-the-art in-memory graph processing systems implemented in one system. We present the trade-offs between gains in the algorithm execution phase, achieved by different optimisations, and the overhead these optimisations cause in the pre-processing time.

In **Chapter 4** we show how graph processing can be scaled up into the Cloud, when the graph does not fit in the memory of a single machine.

Chapter 5 describes how fast NVMe technologies can be efficiently used to scale up to storage attached physically to a single machine. NVMe technologies can also be byte addressable, and we explore their potential in graph analytics using a hybrid memory emulator in **Chapter 6**.

The second part of the thesis presents two systems that support changes to the graph structure, without the need to re-compute from scratch.

Chapter 7 presents how to incrementally update the algorithm state and graph structure for graph analytics, while we tackle the same problem for graph pattern mining applications in **Chapter 8**.

In **Chapter 9** we present systems and papers related to the work presented in this thesis. Finally in **Chapter 10** we conclude and present directions for future work.

2 Background

In this chapter, we present common graph representations, the graph algorithms used in this thesis, and computation models used in state-of-the-art graph processing systems. We analyse their benefits and drawbacks, and discuss how the choice of representation and computation model depends on the underlaying storage.

Graph processing involves loading the graph as an edge array from storage, pre-processing the input to construct the necessary data structures, executing the actual graph algorithm, and storing the results. Most papers focus solely on the algorithm phase, but we demonstrate that there is an important trade-off between pre-processing time and algorithm execution time.

2.1 Graph representation

The graph can be stored in different formats, and state-of-the-art systems do not use the same input format. Transforming a raw input into a desired format is considered to be a pre-processing step, while the algorithm execution time is the actual compute time.

Edge array. A simple way to represent the graph is as a *list of edges*. Each edge is identified by a source and destination vertex. During computation, this list is streamed in, if the state of the source has changed, the state of the destination is updated. The benefit of this layout is that the graph needs no additional processing before the algorithm starts. However, the input is often unsorted, as for example data dumped by a web crawler, thus making it hard to extract the neighbours of a single vertex without streaming in the entire graph.
It is easy to add new edges to the edge list, they are simply appended to the end. However, performing lookup operations to delete/update a particular edge, or edges of one vertex, requires a pass over the entire graph.

Adjacency list. Edges can be fully sorted by source, or destination vertex. Each vertex has a pointer to the beginning of its neighbour list. The neighbour lists of all vertices are stored in a contiguous array. To access its list of neighbours, a vertex simply reads the edges at a particular offset. Since the source is the same for all neighbours of a particular vertex, it can be

removed from the edge list. This reduces the amount of data stored by two. This data structure is called an *adjacency list*, or more precisely, a Compressed Sparse Row (CSR) representation of the graph.
To update the adjacency list, individual edges are found quicker than in edge arrays. To add or delete an edge, the array storing the neighbours has to be expanded and data copied forward to make room for the new edge, or backwards to replace a deleted edge. This is inherently expensive.

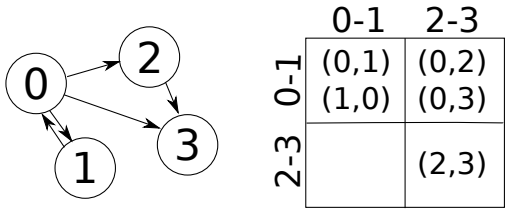


Figure 2.1 – Transforming a graph into a grid representation.

2-D Grid. Since sorting edges fully can be expensive, it is also possible to group, or bucket them into grid cells. This data structure is inspired by the grid data structure first introduced in GridGraph [149], which aims to maximise reuse of data read from disks. Computation then iterates over cells. Mosaic [85] uses a variation of the grid representation. These systems trade coarse grained accesses and sequentiality for an increased amount of I/O compared to an adjacency list, for a subset of algorithms.

GridGraph partitions the graph into a $P \times P$ grid. Figure 2.1 shows how a graph with four vertices is represented as a grid. The vertices are partitioned in $\frac{nVertices}{P}$ ranges where $nVertices$ is the number of vertices in the graph. The corresponding cell is filled with all edges whose source vertex is in that row, while the column is identified by the destination vertex. The smallest unit of computation is a cell. This data layout has also been shown to significantly reduce the cache miss rate in memory [87], due to the fact that it always accesses a limited subset of vertices. It naturally supports the case when the vertex state does not fit in DRAM. For a given P, it suffices that the state of $\frac{nVertices}{P}$ fits in DRAM. We discuss this in more detail in Section 5.5.2.

Mosaic is specifically designed for use on NVMe devices. The system relies on the presence of accelerators such as XeonPhis to offload parts of the computation while running at higher bandwidth. Mosaic uses a more complex version of the grid and compresses the graph, further reducing the amount of I/O done. The grid itself can have an imbalanced number of edges within a cell, leading to load imbalance during computation. Mosaic addresses this by merging grid cells into tiles. The grid cells are traversed in Hilbert order, their edges added to a tile, until the number of edges reaches a cut off point. In addition to better load balancing, Hilbert ordering further increases cache locality [85, 90]. To compress the graph, Mosaic cuts off the number of edges as soon as the number of unique vertex IDs within a tile reaches 2^{16}, after which the vertices are relabelled to 16-bit values (given that the original IDs were 32-bit values).

While the tiles in Mosaic offer many benefits compared to the original grid data structure, the pre-processing time is much higher in this case. We demonstrate how these benefits can be achieved without creating tiles with reduced pre-processing time.

2.2 Graph algorithms

In this thesis we differentiate between two types of algorithms, which we classify into graph analytics and graph pattern mining algorithms. While they have many similarities, we differentiate between the two groups by the locality of explorations from a vertex, and the amount of data generated during the algorithm execution.
Graph analytics compute various graph-wide properties, usually through iterative matrix vector multiplication. Examples of such problems include PageRank and Connected Components.
However, their design is based on the "think like a vertex" approach, and iterative matrix vector multiplication, which makes them unsuitable for mining algorithms that require searching and enumerating subgraphs within a larger graph [122]. Graph analytics update the state associated with vertices, while during graph pattern mining we output instances of a pattern found in the graph. The output is often exponentially larger than the input graph.

2.2.1 Graph analytics

Examples of **graph analytics** algorithms are breadth first search, Pagerank, shortest path, connected components. They can potentially explore the entire graph. In this group, algorithms differ in the number of vertices and edges processed during an iteration.

Traversal algorithms

As their name suggests, traversal algorithms, often explore (traverse) a given path from a vertex, such as the shortest path from one vertex to all other vertices in the graph.

Breadth first search (BFS) is a traversal algorithm that discovers all vertices of a graph reachable from a root vertex, building a breadth first tree. At every iteration, only the vertices added to the tree in the previous iteration are active. Threads iterate through their neighbour lists and add vertices not already in the tree.
Single Source Shortest Path (SSSP) is similar to BFS, but computes on a weighted graph. A weight is associated with each edge representing the length of the path between two vertices. The algorithm picks edges with a smaller weight. If a vertex has already been reached from the root, but a new , shorter path is discovered, it is reactivated to be computed on in the next iteration. This way, it propagates the new distance to its neighbours.
Weakly connected components (WCC) computes the connected components in an undirected graph. We implement WCC using label propagation. All vertices start with distinct labels. They propagate their label to their neighbours, and the neighbour with the higher label changes its label. At the end, when no vertex changes labels anymore, the algorithm

ends. Vertices inside the same component will have the same labels. If the input graph is directed, it is still possible to compute WCC on it, but all the vertices have to be active at every iteration, and the algorithm can activate both source and destination to be processed in the next iteration.

Global, sparse matrix multiplication (SpMV) algorithms

These algorithms always process the entire graph, and are built around a SpMV kernel.

Pagerank [103] (PR) is a website ranking algorithm. All vertices are active at every iteration, and the algorithm executes for a fixed number of iterations. The initial rank of a vertex is computed based on its out-degree. This rank is then sent to all its neighbours. Each vertex, sums up all the ranks it receives from neighbours, and adapts its own rank based on that.
Collaborative filtering is a widely used technique in machine learning for building recommender systems. The input is a bipartite graph of users and their ratings for a subset of items. The goal is to recommend items to a user based on their previous rankings. There are two different algorithms used in the literature to perform collaborative filtering: Stochastic gradient descent and Alternating least squares [146]. In this thesis we use the latter.
Belief Propagation (BP) [70] is a well-known machine learning algorithm to infer the state of vertices based on previously observed values. Vertices read the state of all it's neighbours, and infer a new belief based on that. They then propagate their new belief to their neighbours. The algorithm can either stop when the change in belief is below a threshold or after a fixed number of iterations.

Non iterative algorithms

The following two algorithms are non-iterative. They require one iteration over the entire graph. Other than that, they have the same characteristics as global SpMV-based algorithms.

Triangle counting (TC) is a technique to discover the number of triangles formed by vertices in the graph. A triangle exists when a node is connected to two other nodes that are mutually connected. Graph analytics systems implement this algorithm such that vertices look for intersections in their neighbour lists.
Sparse matrix vector multiplication (SpMV) multiplies the adjacency matrix of the graph with a vector of values. The matrix entries are stored as edge weights.

2.2.2 Graph pattern mining

Graph pattern mining applications explore the characteristics of subgraphs within the graph, and typically explore the immediate neighbourhood of a vertex.

k-clique enumeration (k-C) finds all cliques in the graph (fully connected subgraphs) of a given size within a graph. We also consider an extended version of this algorithm in which all

vertices in the clique must have distinct labels (k-**CL**).

Graph keyword search (k-GKS) finds all possible minimal subgraphs whose vertices contain all k labels of interest. We define the maximal number of non matching words that are accepted within a subgraph.
Motif counting (MC) counts the number of times each motif appears in a graph. Motifs are *isormorphic* pattern instances, i.e. subgraphs that are identical if their vertices were to be relabeled. We refer to motif counting with motifs of size $\leq k$ as k-MC.

2.3 Graph shape

The graphs of interest to research and industry have different shapes with respect to their connectivity. We differentiate between *power-law*, *high diameter* and *bipartite* graphs in this work.

Most of the graphs of interest fall into the first category, and have a so-called power-law degree distribution. They are thus referred to as **power-law** or **natural** graphs. Examples of such graphs are social network graphs like the Twitter [77] follower graph, the LiveJournal [1] or randomly generated RMAT [33] graphs.

In these graphs, only a few vertices are highly connected, having direct edges to many vertices in the graph. Most of the vertices have only a few neighbours. This is a characteristic that often leads to load imbalance in computation and graph partitioning. However, a full traversal of the graph is typically possible in very few iterations, as any vertex can be reached from another vertex in just a few hops [22].

High diameter graphs, on the other hand, have vertices with a few direct connections and require many iterations for a full traversal of the graph. For example, to compute the Breadth First tree on the US-Road graph [2], we require over 1400 iterations (compared to 7 for a medium-sized power-law graph).

The third group, **bipartite** graphs, is of interest in recommender systems, where the edges in the graph connect vertices belonging to two distinct groups. For example, the Netflix [146] graph connects users, on one side, to the movies they ranked, on the other side.

2.4 Computation model

The "Think like a vertex" model was widely adopted after the introduction of Pregel. The state of the algorithm is stored as a property of the vertex. For example, the distance of a vertex from the root of a BFS tree, or the rank of a website (represented as a vertex) in Pagerank. A graph algorithm executes in iterations (super-steps), following a block synchronous computation model. During each iteration, a set of vertices is considered active if there is a change in their state. They propagate this new state to their neighbours.

The Gather-Apply-Scatter(GAS) model was introduced by Powergraph [55], where each of the three steps (gather, apply and scatter) is executed within one superstep. This model was widely adopted in many systems that followed, and is a variation of the Think like a vertex model.

Vertices *gather* messages from their neighbours. Once all messages are collected, they *apply* the aggregated value to their own state, and update it. The new state is then *scattered* - propagated, to the neighbours of a particular vertex.

Different systems implement these steps differently. At a high level, the implementation of the model depends on the graph representation. We differentiate between vertex-centric, edge-centric and grid-centric computation.

Vertex-centric computation. In memory graph processing systems store and compute from DRAM. Many single machine [98, 119] and distributed in-memory graph processing systems [34, 37, 55, 56] have been presented in recent literature. The graph is represented as an adjacency list. In case the system is distributed, the graph is further partitioned across the machines. Graph partitioning in itself is an NP-hard problems, and many papers were published on this topic alone [41]. In general, every system partitions for good load balance. As the major bottleneck in graph analytics is fetching the lists of neighbours, and the number of neighbours varies among vertices, partitioning often aims to balance out the number of edges per machine, rather than the number of vertices.

In every iteration, active vertices are put in a workqueue. The workqueue is read in the next iteration. Threads fetch work from the queue, and apply a compute function to the vertex state. If the state of the vertex has changed, the IDs of its neighbours are placed into the workqueue for the next iteration.
This process is repeated until there are no vertices in the queue, or after a fixed number of iterations. This model of computation is referred to as a **push** model, since information is *pushed* from source to destination.

The state of the neighbours can be updated directly by the thread that activates it for processing in the next iteration, using atomic operations. This effectively merges the *Scatter* and *Apply* steps. To avoid atomic operations, some systems use either message passing, or a **pull** model. In the *pull* model, vertices traverse their incoming edges, read the state of their upstream neighbours, and update their own state, thus *pulling* information in.

Even though the graph itself can outgrow the amount of available DRAM, the state of the vertices most often fits in DRAM. A number of semi-external [57, 80, 144] systems leverages this insight, by storing the vertex state in memory, and the adjacency list on SSD storage. These systems perform vertex-centric computation even though the adjacency list is on external storage.
To avoid fine grained access to storage, they merge small requests, or explicitly cache frequently accessed edges to avoid more expensive I/O.

Edge-centric computation. If the state of the vertices fits in DRAM, the edges can be simply memory mapped and streamed in from storage. The state of the destination vertex is updated based on the state of the source, in place. Unless otherwise stated, in this thesis we assume that the state fits in DRAM.
When this is not the case, the edge array is partitioned. In this section, we present two techniques used in state-of-the-art systems designed for SSDs and HDDs, and show how to efficiently support this scenario on fast PCIe NVMes in Chapter 5.

GraphChi [78] was the first system to process a large graph on a single machine out of core. The graph is pre-processed, such that the edges are grouped into shards. The vertices of the graph are split in P shards. The number of edges across shards is balanced, and one shard fits in the DRAM of the machine. Each shard contains edges whose destination vertex is in its interval. The edges in the shard are sorted by source. To update all neighbours of a vertex, one shard is first loaded into memory. For every vertex in range P, we update its state based on the state of its incoming neighbours, via the edges in the same shard. Since no other shard contains edges whose destinations are in P, GraphChi reads the edges from other shards, whose source vertex is in P, thus updating the state of all neighbours of vertices in P.
X-Stream [114], partitions the edges into a much simpler data structure - streaming partitions. Without balancing the number of edges across partitions, X-Stream splits the vertices in P equal intervals. Edges whose source vertex belongs to an interval Pn, belong to partition n. X-Stream follows a scatter-gather model of computation. During scatter, the state of P vertices is loaded into DRAM, and the edges from the partition are streamed in. If the destination vertex needs to be updated, X-Stream appends the new value to the update file for partition Pm , whose source vertex is in the range m. Once all partitions have generated updates, X-Stream gathers the updates. The state of P vertices is loaded into memory, and the update file for the corresponding partition is streamed in.
This way, X-Stream does fully sequential reads and writes, maximising the bandwidth. X-Stream outperforms GraphChi, especially when taking into account the more expensive pre-processing step of GraphChi. The time to sort the edges and create the shards is not insignificant, often being orders of magnitude higher than the actual algorithm execution time.

Grid-centric computation. On graphs represented as grids, computation happens in a row- or column-orientated manner. Threads iterate over all cells within a row, or all cells within a column. When processing the cells of a row, the fact that each cell has edges for a disjoint set of vertices, is leveraged to remove locks.

By making the grid cell the unit of computation, the I/O exhibits higher sequentiality, thus using the device more efficiently compared to adjacency lists. During computation, after updating the state of a vertex, the row to which all its outgoing edges belong, is marked as active. When no source vertex within a row is active, a row can be completely skipped, decreasing the total amount of I/O done compared to edge arrays.

Part I Static graph processing

3 In-memory graph processing

In this chapter we focus on single-machine in-memory graph processing systems. With the recent increase in main memory size and number of cores, such machines can now process very large graphs in a reasonable amount of time.

With few exceptions [32, 115], most papers on graph processing systems present a new system and compare its performance (and occasionally its programmability) to previous systems. While interesting, these comparisons are often difficult to interpret, because systems are multi-dimensional, and therefore a variety of features may contribute to observed performance differences. Variations in hardware and software infrastructure, input formats, algorithms, graphs and measurement methods further obscure the comparison.

In this chapter we compare different techniques used in graph processing systems. We implement various techniques proposed in different papers in a single system.

We structure our investigation of algorithm execution time along two dimensions. In a first dimension, we distinguish between a vertex-centric approach, in which the algorithm iterates over vertices, and an edge-centric approach, in which the algorithm iterates over edges. In addition, we propose a new iteration approach, adapted from out-of-core systems [149], in which the algorithm iterates over grids, with improved cache locality as a result. In a second dimension, we distinguish between algorithms that push information to their neighbours, or pull information from them. We also consider algorithms that dynamically choose between push and pull.

To illustrate through a simple example the importance of an end-to-end view, we analyse the push-pull approach to Breadth First Search (BFS)[1]. Earlier papers [23, 24, 119] have demonstrated that, for BFS, a push-pull approach results in better algorithm execution time than the conventional push approach. Figure 3.1 shows the end-to-end execution time of BFS on the well-known Twitter follower graph [77] using both approaches. While the algorithm execution time is indeed 3× smaller for push-pull, the overall execution is completely dominated by

[1] see Section 3.5 for a precise definition of push-pull

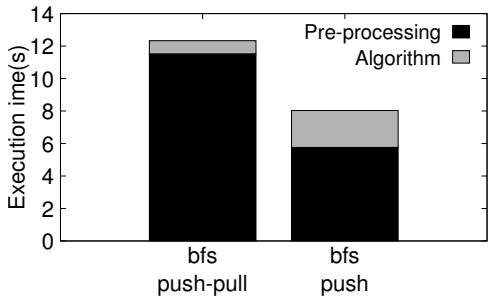


Figure 3.1 – Example of the trade-off between pre-processing and algorithm execution time for BFS on the Twitter graph: push-pull improves algorithm execution time, but the required pre-processing time leads to overall worse end-to-end execution time (measured on Ligra [119]).

pre-processing. The pre-processing time is 2× larger for push-pull, resulting in 1.5× worse overall end-to-end time.

In addition to different methods of iteration and information flow, various optimisations have been proposed to take advantage of memory locality on NUMA machines. These optimisations often take the form of partitioning data structures during pre-processing, such that most accesses during algorithm execution are local to a NUMA node. Continuing the theme of the trade-off of pre-processing versus algorithm execution times, we investigate whether such pre-processing pays off for graph processing.

The main results presented in this chapter are:

- An illustration of the fundamental trade-off between pre-processing and algorithm execution time in graph processing.
- An evaluation of different techniques for building adjacency lists, showing that radix sort provides the best performance when the graph is in memory or when it is loaded from a fast storage medium.
- An evaluation of the pre-processing vs. algorithm execution time trade-off for vertex-centric vs. edge-centric computation, showing that the construction of adjacency lists for vertex-centric processing may or may not pay off, depending on the algorithm execution time.
- An evaluation of a push vs. pull information flow, illustrating the benefits of reduced computation for push vs. reduced synchronisation for pull.
- An evaluation of the pre-processing vs. computation trade-off for combined push-pull information flow, showing that the extra pre-processing costs associated with this combination outweigh gains in algorithm execution time.
- The adaptation of an out-of-core technique for improving the cache locality and the synchronisation overhead of an in-memory graph processing system.
- An evaluation of the pre-processing vs. computation tradeoff for NUMA-aware optimisations, demonstrating that their large pre-processing times can be compensated by

gains in algorithm execution time only on large NUMA machines and only for certain algorithms.

The outline of this chapter is somewhat unusual. We start in Section 3.1 with an overview of the hardware and software used. We discuss data structures and pre-processing costs in Section 3.2. In Section 3.3 we look at the relationship between the data layout and vertex-centric or edge-centric computation. Section 3.4 discusses methods for improving cache locality. In Section 3.5 we evaluate the choice between push and pull approaches and its implications for algorithm execution time, pre-processing time and synchronisation overhead. Section 3.6 evaluates graph partitioning approaches to take advantage of NUMA characteristics. Section 3.7 summarises results on graphs and algorithms not discussed in previous sections. Section 3.9 provides an overview of all the results in one place and concludes the chapter.

The code used for the experiments in this chapter and instructions on how to run them is available at: https://github.com/epfl-labos/EverythingGraph.

3.1 Experimental setup

Experimental environment. We evaluate the pre-processing and algorithm execution times on two machines, each representative of a large class of machines. Machine A has 2 NUMA nodes, and is less sensitive to NUMA effects than machine B, which has 4 NUMA nodes. More precisely, machine A has 2 Intel Xeon E5-2630 processors, each with 8 cores (16 cores in total) and a 20MB LLC cache, and 128GB of DRAM. Machine B has 4 AMD Opteron 6272 processors, each with 8 cores (32 cores in total) and a 16MB LLC cache, and 256GB of DRAM. Unless otherwise stated, all experiments are run on Machine B.

The pre-processing times, unless otherwise stated, assume the graph is already loaded in memory. The costs of loading the graph into memory and its implications on pre-processing are discussed separately.

The subset of vertices or edges to be processed during a computation step is kept in a work queue. Threads take work items from the queue in large enough chunks to reduce the work distribution overheads. We parallelise both pre-processing and computation using the Cilk 4.8 parallel runtime system. When needed, Cilk balances the work among threads by allowing threads to steal work items from one another. Threads start by fetching a small chunk of vertices at a time (1024). If the number of vertices to be processed decreases, the chunk size is decreased as well. We measure the effectiveness of this load-balancing scheme. For Pagerank, threads were idle only 1% of the time, compared to 70% with a static partitioning. For BFS, the idle time was reduced from 85% to 10%. The higher percentage of idleness is due to the fact that there are iterations where there are fewer vertices than threads. In this scenarios, it would be more beneficial to have more threads process the neighbours of one vertex, rather than assign one vertex to a thread. We leave this for future work.

Our experiments using OpenMP and PThreads show comparable execution times and are

therefore not reported.

Algorithms. We select six algorithms with different characteristics in terms of functionality (traversal, machine learning, ranking), vertex metadata, as well as the number of vertices active during computation steps (iterations).

We evaluate the following three traversal algorithms. **Breadth-first search (BFS)**, **Weakly connected components (WCC)** and **Single source shortest path (SSSP)**. We also evaluate two algorithms that compute over the entire graph: **Pagerank (PR)** [103] and **Sparse matrix vector multiplication (SpMV)**.

Datasets. Table 3.1 gives an overview of the graphs used along with their number of vertices and edges. We use both synthetic and real-world datasets. The synthetic datasets are power-law graphs generated by the RMAT graph generator [33]. We generate graphs of different sizes to evaluate the scalability of optimisations in terms of graph size. RMAT26 is the biggest RMAT graph that we can fit on all machines for all approaches. As a real-world power-law dataset, we use the Twitter follower graph [77], which is the largest real-world dataset that fits on all machines for all approaches.

In addition to these two graphs, we also use the US-Road graph from the DIMACS challenge [2]. This graph has a different shape than power-law graphs: it has a high diameter, and all vertices have a small in/out degree. We use it to study the impact of the shape of the graph on different computation approaches. Finally, for ALS we use the bipartite Netflix graph [146].

Graph	Vertices	Edges
RMAT-N	2^N	2^{N+4}
Twitter	62M	1468M
US-Road	23.9M	58M
Netflix	0.5M	100M

Table 3.1 – Graphs used in the evaluation, with their number of vertices and edges.

For brevity, in Sections 3.2 to 3.6, we primarily present results for BFS and Pagerank (with 10 iterations). These algorithms represent opposite ends of the spectrum, both in terms of the percentage of the graph that is active during each step of the computation and in terms of computation complexity. Furthermore, we report results primarily for the RMAT26 graph. We include results for other algorithms and graphs only when they provide additional insights that depend on the algorithm or the shape of the graph. Section 3.7 completes the picture by presenting data on the combinations of algorithms and input graphs not discussed in earlier sections.

3.2 Data layouts and pre-processing costs

In this section we first present the pre-processing costs associated with the adjacency list and edge array layouts, presented in Section 2.1.

3.2.1 Pre-processing costs

Edge array. The layout of edge arrays matches the format of the input file, and it suffices to map the input file in memory to be able to start computation. As such, edge arrays incur no pre-processing cost.

Adjacency lists. We explore two techniques to build adjacency lists.

The simplest technique consists of reading the input file and *dynamically* allocating and resizing the edge arrays of vertices as new edges are discovered.

The second technique avoids reallocations by loading the graph as an edge array and then sorting it by source vertex. Vertices use an index in the sorted edge array to point to their outgoing edge array. The incoming edge array is created by sorting the edge array by destination vertex. This way the edges are stored contiguously in memory, corresponding to compressed sparse row format (CSR). The performance of this approach depends on the sorting algorithm.

The most common approach to sort edges is to use a *count sort*. In a first pass over the edge array, we count the number of outgoing (incoming) edges for each vertex. In a second pass over the edge array, we place edges at the correct location in the sorted edge array. Most existing graph analytics frameworks use this approach, as it is optimal in terms of complexity (the input array is only scanned twice).

An alternative approach is based on *radix sort*. Radix sort treats keys as multi-digit numbers, and sorts the keys into buckets one digit at a time. In the parallel version, each thread recursively sorts a subset of edges into a small number of buckets [136]. The advantage of radix sort is that buckets are written sequentially, and therefore have good locality. The complexity of the sort is relatively low. We use a radix size of 8 bits (256 buckets) which only requires $log_2(\#vertices)/8$ recursions to sort the edge array (e.g., 4 recursions for a graph with 4 billion vertices, 8 recursions with 2^{64} vertices).

3.2.2 Evaluation

Table 3.2 presents, for all three approaches (dynamic, count sort and radix sort), the execution times for creating outgoing per-vertex edge arrays and for creating both incoming and outgoing per-vertex edge arrays, for the Twitter graph and assuming the graph is already in memory. Using a radix sort is 4.8× faster than count sort. Surprisingly, sorting using a radix sort is also 4.9× faster than dynamically building the per-vertex edge arrays. Radix sort is faster, because

Adj. list pre-processing variation	Twitter out	Twitter in-out	LLC misses
Dynamic	20.0	27.2	69%
Count sort	19.5	23.9	71%
Radix sort	4.0	8.5	26%

Table 3.2 – Adjacency list creation cost (in seconds) and percentage of LLC misses on machine B when the graph is in memory.

it has better cache locality than the other solutions. Both the dynamic approach and count sort sequentially read the input edge array, but the subsequent steps have poor cache locality. The dynamic approach requires jumping between per-vertex arrays to insert a newly read edge. Count sort requires jumping between vertices as well in order to count their degree. It then does another scan of the input to place edges at their corresponding offsets in the sorted edge array. This step jumps between distant positions in the array.

Figure 3.2 presents the evolution of the pre-processing time for RMAT graphs depending on the graph size. All approaches scale as the graph size increases. The radix sort approach is always faster than the count sort and the dynamic sort approach (3.3× and 3.8×, respectively, on RMAT26).

For smaller graphs, count sort is slower than both the dynamic and radix approaches. The approach requires reading the edge array twice (once for counting, and then once to place edges in the sorted array). As the graph grows, however, the fact that the second pass in count sort does no reallocations makes it slightly better than the dynamic approach (e.g. there are 32 million reallocations for an RMAT26 graph).

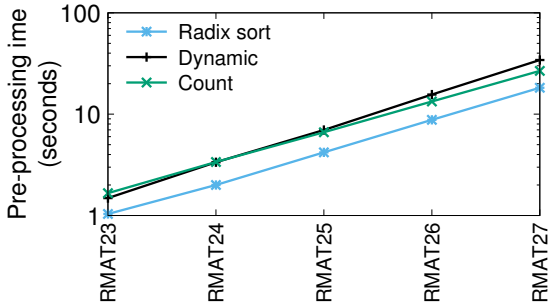


Figure 3.2 – Scaling of pre-processing methods for adjacency list creation. All methods scale linearly with the graph size. RMAT-(N+1) is double the size of RMAT-N, and so is the pre-processing time.

3.2.3 Loading and pre-processing

The previous discussion assumes that the graph is already loaded into memory. Conclusions are different when the graph is to be read from storage or over the network. Indeed, doing a radix sort can only be partially overlapped with loading the graph in memory. In contrast, the dynamic approach of allocating and resizing per-vertex edge arrays can be fully overlapped with loading. For count sort, only the first pass can be overlapped with loading.

3.2.4 Evaluation with loading included

Table 3.3 presents the combined loading and pre-processing time when the graph is loaded from an SSD (380MB/s maximum bandwidth) and from a regular hard drive disk (100MB/s).

If we take loading speed into account, dynamically allocating per-vertex edge arrays becomes faster than radix sort when the storage medium is slow. On the SSD the total time for the radix

sort approach is shorter than or more or less the same as the dynamic approach. The results for count sort are, as before, inferior, and are not included for that reason.

Pre-processing approach	RMAT26 out	RMAT26 in-out
Dynamic, loaded from SSD	20.7	40.0
Radix-sort, loaded from SSD	21.2	27.0
Dynamic, loaded from disk	61.0	61.1
Radix-sort, loaded from disk	65.0	71.0

Table 3.3 – The cost of pre-processing for adjacency list creation with loading time included. Results show the time when building only the outgoing per-vertex edge arrays, and when building both the outgoing and incoming per-vertex edge arrays. The pre-processing is overlapped with loading when the adjacency list is created dynamically.

Summary. Costs associated with loading and building data structures in memory are non-negligible, and different approaches shine in different situations. Surprisingly, using radix sort to build adjacency lists is the fastest approach when the input file is in memory or loaded from a fast medium. When the graph is loaded from a slow medium, building adjacency lists dynamically is a better option, because it can be overlapped with loading.

3.3 Data layout and graph traversal

3.3.1 Vertex-centric vs. edge-centric

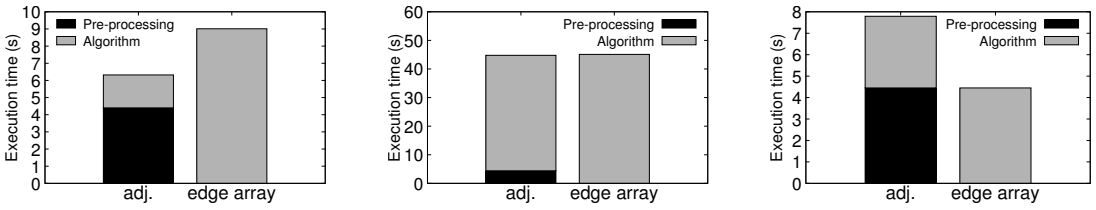
The choice of data layout impacts the decision of how to traverse the graph. In this section, we show that the best performing data layout and corresponding traversal model depend on the algorithm.

Computation on edge arrays happens in an **edge-centric** manner, and is quite simple: at every iteration of the computation the whole edge array is scanned, and the graph algorithm is called on every edge. This computation model is efficient, because scanning an edge array is cache-friendly: most of the accessed data is prefetched before being used. The drawback of this layout is that it offers no easy way to work on a subset of the vertices: a full scan of the edge array is required to find the edges of a vertex.

Adjacency lists are a natural solution to this problem. They enable **vertex-centric** computation, in which work is only performed on the subset of active vertices.

3.3.2 Evaluation

To illustrate the impact of data layout and traversal model on the end-to-end execution time, we show in Figure 3.3 the pre-processing and algorithm execution times of BFS, Pagerank, and SpMV on RMAT26. For BFS, vertex-centric computation performs the best, because during an iteration BFS only works on a limited subset of the graph. Edge arrays are not well suited for this type of computation, as all edges of the graph are read at every iteration.



(a) BFS on RMAT26 (b) Pagerank on RMAT26 (c) SPMV on RMAT26

Figure 3.3 – Pre-processing and algorithm execution time for BFS, Pagerank and SpMV on RMAT26, using vertex-centric computation on an adjacency list or edge-centric computation on an edge array.

In contrast, Pagerank accesses the entire graph in every iteration. Looking only at algorithm execution time, vertex-centric computation still performs a bit better, because it has better cache locality (all edges from a vertex are processed on the same core). When taking into account the pre-processing time, however, the end-to-end execution time is the same as for edge-centric computation.

Finally, SpMV is an algorithm that makes only a single pass over the graph. Here, edge-centric computation produces the best end-to-end result, since the cost of building adjacency lists for vertex-centric execution is not amortised by any gains in algorithm execution time.

3.4 Cache-locality

Due to their irregular access patterns, graph algorithms usually exhibit poor cache locality. Last-level cache (LLC) misses may happen during three key steps of the computation: fetching an edge, fetching the metadata associated with the source vertex of the edge, and fetching the metadata associated with the destination vertex of the edge. In this section, we study how to lay out the data in memory to reduce the number of LLC misses, and we explain the pre-processing costs associated with creating those layouts.

3.4.1 Impact of the data layout

Edge array. In edge-centric computation, since edges are streamed, they are prefetched efficiently and do not incur cache misses. Fetching the metadata of the vertices, however, leads to random accesses with poor spatial and temporal locality.

Adjacency lists. In adjacency lists, computation is performed from the point of view of a vertex: a core iterates over all edges of a given vertex before processing another vertex. As a consequence, the metadata of the source vertex is read only once, after which it is cached. This is beneficial for vertices that have a large number of edges. Fetching edges may introduce a cache miss for the first edge, but subsequent edges are prefetched, as with the edge array. Also similar to the case of the edge array, the metadata of the destination vertices exhibits poor cache behaviour.

Grids: optimising edge arrays. To improve the cache locality of edge arrays, data is laid-out as a grid of cells. Each cell contains the edges from a range of vertices to another range of

vertices.

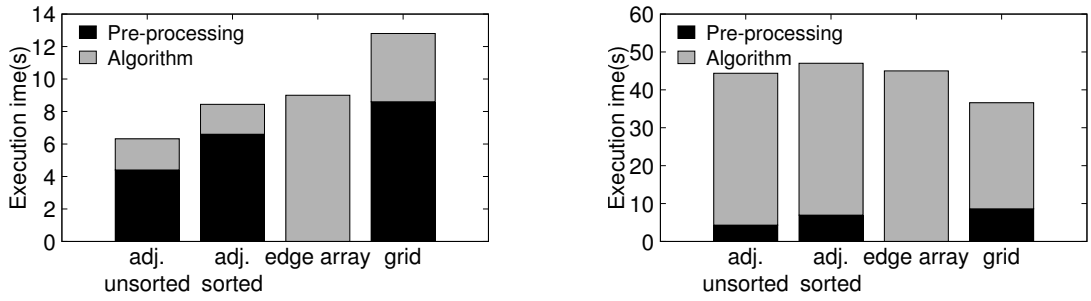
We construct the grid using the same radix sort approach as for building adjacency lists. Instead of bucketing edges by source vertex, we bucket them by the cell to which they belong. The optimal number of cells in the grid depends on the graph shape and size. We experimentally find that a grid of 256x256 cells performs best on the Twitter and RMAT26 graphs. Building a grid is slightly more expensive than building an adjacency list (the number of cells in the grid is equal to $(\#vertices/256)^2$, which is higher than the number of vertices for large graphs).

We compare using radix sort with a dynamic approach for building the grid, and the conclusions regarding different pre-processing approaches made in Section 3.2.1 are applicable to grids as well: radix sort is faster when the graph is in memory or loaded from a fast medium, while dynamically building the grid is faster otherwise.

optimising adjacency lists. An intuitive idea to improve cache locality in adjacency lists is to sort the per-vertex edge arrays by destination. Indeed, the metadata of vertices with contiguous IDs is also contiguous in memory, thus when accessing vertex 0 and then vertex 1, the metadata of vertex 1 is likely to be present in cache. Of course, sorting the per-vertex edge arrays increases the pre-processing cost.

3.4.2 Evaluation

Figure 3.4 compares the pre-processing and algorithm execution times of BFS and Pagerank on RMAT26, on the unsorted adjacency list, the sorted adjacency list, the edge array and the grid. Table 3.4 presents the cache miss rate for these four data layouts.



(a) BFS on RMAT26

(b) Pagerank on RMAT26

Figure 3.4 – Impact of cache-related optimisations on pre-processing and algorithm execution time for BFS and Pagerank on RMAT26.

Data layout	BFS	Pagerank
Edge array	57%	83%
Grid	23%	35%
Adjacency list	63%	78%
Adjacency list sorted	63%	78%

Table 3.4 – Cache miss ratio for BFS and Pagerank on RMAT26.

BFS. For BFS, the unsorted adjacency list remains the solution with the best end-to-end execution time. Looking at algorithm execution time alone, BFS is 2.4× faster with a grid than with unsorted per-vertex edge arrays. However, creating the grid adds significant pre-processing time (9s), making the grid the slowest solution overall for BFS. Sorting the per-vertex edge arrays also leads to end-to-end performance inferior to unsorted adjacency lists. The pre-processing time increases, and the algorithm execution time does not decrease. Table 3.4 shows that sorting the per-vertex arrays does not significantly impact the cache miss rate. The destination vertices are accessed in order, but in practice a cache line only contains the metadata associated with very few vertices (64 in the case of BFS). Even when sorted, the destination vertex identifiers in the per-vertex edge arrays are sufficiently far apart for their metadata to fall in different cache lines, which explains the limited impact of this optimisation on the number of cache misses and therefore on algorithm execution time. The increased pre-processing time for sorting the per-vertex arrays increases end-to-end execution time.

Pagerank. Even with the added pre-processing cost, the grid outperforms all other data layouts for Pagerank: it is 1.4× faster than an edge array and 1.3× faster than an unsorted adjacency list. This improvement is a direct result of the reduced cache miss rate when using a grid. As shown in Table 3.4, the cache miss ratio for the grid is less than half of that for the other data layouts. As for BFS, sorting the per-vertex edge arrays provides no benefit for Pagerank, for the same reasons. A cache line can fit at most 6 vertices for Pagerank, leading to an even smaller improvement in spatial locality than for BFS.

Summary. Creating a grid improves cache reuse and has a significant impact on algorithm execution time. Yet, this comes at the cost of an extra pre-processing, which is not always amortised. Different layouts also shine in very different situations. For instance, the grid is the best solution for Pagerank, but the slowest on BFS.

3.5 Information flow: Push and Pull

One of the core design decisions for a graph processing system is the information flow model it adopts. Information propagates through the graph in one of two ways: a vertex either **pushes** data along its out edges, writing to the state of its neighbours, or it **pulls** data along its incoming edges and updates its own state. These two approaches have important implications on computation, synchronisation and pre-processing that we detail in this section.

3.5.1 Impact on end-to-end execution time

Impact on algorithm execution time

The **push** and **pull** approaches have different impact on the number of vertices and edges that need to be accessed during an iteration.

First, the **push** approach allows working on a subset of the vertices, while the **pull** approach does not. When pushing, vertices that do not need to propagate their value can be safely

ignored. In contrast, the pull approach requires a vertex to scan all its incoming edges for neighbours that could potentially propagate a value. It also requires a pass over all vertices to check whether they need to look at their incoming edges (e.g., whether they have already been discovered in BFS).

Second, for some algorithms, the **pull** approach allows stopping the computation for a vertex in the middle of an iteration, while the **push** approach does not. Indeed, while **pulling** data a vertex may stop pulling before exploring all its incoming edges. For instance in BFS, if a vertex marks itself as discovered in the middle of an iteration, it stops exploring its remaining incoming edges. This guarantees that the vertex is discovered only once. In the **push** approach, vertices need to check that all their neighbours have been discovered, which leads to redundant work if multiple vertices have the same neighbours.

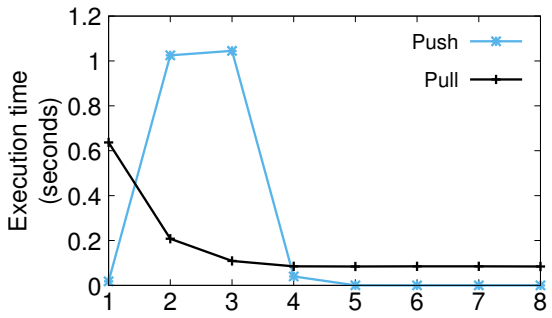


Figure 3.5 – Per-iteration algorithm execution time for push vs. pull for BFS on RMAT26.

Figure 3.5 shows the per-iteration execution time of pushing vs. pulling for BFS on an RMAT26 graph. During the first iteration and after the third iteration, pushing is faster than pulling. During iterations 2 and 3, pulling is faster than pushing. This difference is explained by the percentage of the graph that is accessed during the iterations: most vertices in the graph are discovered during iterations 2 and 3. When pushing data, lots of redundant work is done in these iterations.

Because pushing data and pulling data perform best at different phases of the computation, some frameworks dynamically switch between pushing and pulling, depending on the number of active vertices in an iteration [23, 24, 119].

Impact on synchronization

A significant part of the algorithm execution time may involve synchronisation. For example, in Pagerank on an RMAT26 graph with 16 cores, 40% of the algorithm execution time is spent in code protected by locks. The goal of this section is to evaluate the possibilities for lock removal, how they depend on the data layout and the information flow, and what if any pre-processing costs they induce.

In push mode, a vertex pushes updates to all its neighbours, and thus needs to lock them to

update their metadata. In pull mode, a vertex only updates its own state. Thus, lock removal with adjacency lists requires execution in pull mode.

The grid offers a natural partition of the graph: edges in different rows have different source vertices, and edges in different columns have different destination vertices. To perform computation without locks in push mode, it suffices to assign different columns to different cores. To perform computation without locks in pull mode, it suffices to assign different rows to different cores.

Impact on pre-processing

Adjacency lists. To use push-pull, a system needs to iterate over both outgoing and incoming edges. As a result, when the graph is directed, we need to build both the outgoing and incoming per-vertex edge arrays. In contrast, for push we only need to build the outgoing, and for pull only the incoming per-vertex edge arrays. As a result, for directed graphs push-pull comes with an increased pre-processing cost, compared to push or pull, as seen in Section 3.2.1. When the graph is undirected, it suffices to build the outgoing per-vertex edge arrays (outgoing and incoming edges are the same), and push-pull induces no extra pre-processing cost.

Edge array. Computation over an edge array always requires scanning all the edges in the graph, so there is no advantage to using either push or pull. Furthermore, since the computation is edge-centric and not vertex-centric, locks need to be acquired for all updates. For these reasons, edge arrays are not considered any further in this section.

Lock removal. Lock removal does not require any additional pre-processing, beyond what is otherwise necessary for adjacency lists and grids, but it cannot be used with edge arrays, which have zero pre-processing cost.

3.5.2 Evaluation

BFS

Figure 3.6 presents the end-to-end execution times for BFS running on a directed RMAT26 graph, with adjacency lists, using push-pull, push (with locks) and pull (without locks). We do not show any results for edge array or grid for BFS, as we have shown in Section 3.4 that these approaches lead to inferior results compared to adjacency lists.

Push-pull is much faster in terms of algorithm execution time, but it is 1.5× slower than the push approach in terms of end-to-end execution time because of the extra pre-processing time. When taking pre-processing time into account, we find no combination of graphs, algorithms and machines in which push-pull is beneficial on directed graphs. On undirected graphs, push-pull does not add any pre-processing time, and is thus much faster than just pulling or pushing data. Furthermore, due to the fact that, on average, only a small percentage of vertices is processed per iteration, BFS in push mode performs 20% better than BFS in pull

mode, even though push uses locks and pull does not.

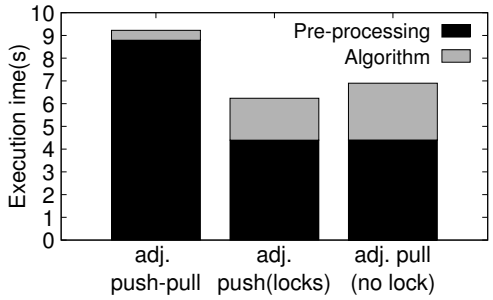


Figure 3.6 – Pre-processing and algorithm execution time for BFS on RMAT26 using push-pull, push (with locks) and pull (without locks).

Pagerank

Figure 3.7 shows the end-to-end execution times for Pagerank in push mode on an adjacency list (with locks), in pull mode on an adjacency list (without locks), in push mode on a grid (with locks), and in pull mode on a grid (without locks). Here, the advantages of removing locks can be clearly seen. On adjacency lists, the version without locks is 40% faster than the push version when looking at end-to-end time. On a grid, the version without locks shows a gain of 1.5× in end-to-end time when comparing to the version with locks.

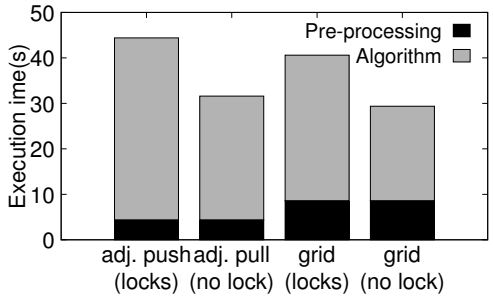


Figure 3.7 – Pre-processing time and algorithm execution time for Pagerank on RMAT26 for push (with locks) on an adjacency list (with locks), for pull on an adjacency list (without locks), for push on a grid (with locks), for pull on a grid (without locks).

Summary. Push and pull on adjacency lists have conflicting benefits. Push works better for algorithms that only access a subset of the vertices in a given iteration, while pull allows vertices to be updated without locks. With grids, locking can be avoided regardless of whether push or pull is used, but the advantage of push remains for algorithms that only access a subset of the vertices. Whether push or pull comes out ahead depends heavily on the nature of the algorithm. A combined push-pull approach requires extra pre-processing, which outweighs the benefits in terms of algorithm execution time.

3.6 NUMA-Awareness

We evaluate the trade-offs between the potential benefits of being NUMA-aware and the overheads it introduces in both the pre-processing and algorithm execution phase.

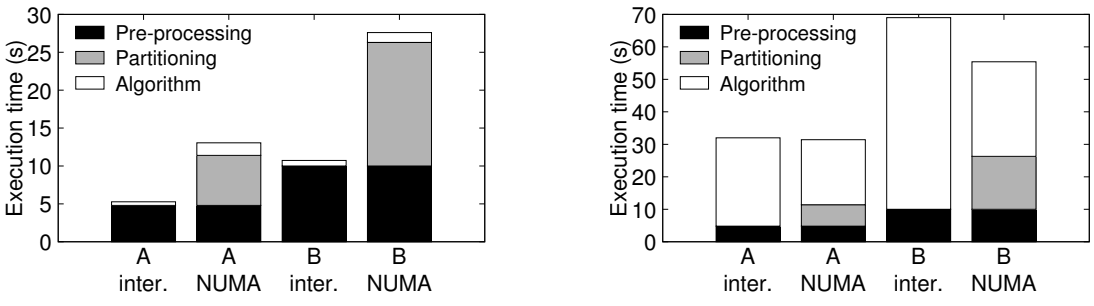
3.6.1 Data layout

In NUMA-aware solutions, the graph is *partitioned* across the NUMA nodes, and threads prioritise work from partitions that are local to their NUMA node. The partitioning scheme divides graph data evenly across NUMA nodes and places related data on the same NUMA node. Partitioning is performed so as to minimise the number of edges whose source and destination vertices are on different NUMA nodes, while still balancing the number of vertices and edges per NUMA node.

We evaluate in particular the partitioning schemes of Polymer [138] and Gemini [147]. The vertices are split into as many subsets as there are NUMA nodes. The outgoing edges of vertices are colocated with their target vertices. This approach avoids random remote writes and balances the number of edges across NUMA-nodes. Threads first process their local partitions. After that, they start working on remote partitions by updating the target vertices that are local to their NUMA node.

3.6.2 Evaluation

We evaluate the potential performance improvement of NUMA-aware data placement on the two machines presented in Section 3.1. Figure 3.8 shows the impact of NUMA-aware graph partitioning of an RMAT26 graph when running BFS and Pagerank. We compare NUMA partitioning to a solution that randomly interleaves the graph data on all NUMA nodes. We use, for each application, the best algorithm in terms of algorithm execution, as presented in the previous sections (push/pull for BFS and pull without locks for Pagerank). The end-to-end execution time is broken down into pre-processing, partitioning and algorithm execution.



(a) BFS - RMAT26

(b) Pagerank - RMAT26

Figure 3.8 – Impact of NUMA-aware partitioning on machines A and B. For each machine we show the pre-processing, partitioning and algorithm execution time for BFS and Pagerank on RMAT26 with memory interleaving vs. NUMA-aware data placement.

Looking at Figure 3.8b, the NUMA-aware data layout improves the algorithm execution time for Pagerank 1.3× on Machine A and 2× on Machine B. However, only on the machine B, with 4 NUMA nodes, does the end-to-end execution time benefit from being NUMA-aware.

In contrast, looking at Figure 3.8a, for BFS the NUMA-aware version is 3.5× slower on Machine A and 1.8× slower on Machine B. For BFS the time spent in partitioning dwarfs the algorithm execution time on both machines. More surprisingly, even when looking only at algorithm execution time, the NUMA-aware version performs worse than the interleaved version. In BFS, in a given iteration, only a small number of vertices is processed, and these vertices often share a common ancestor (e.g., during the first iteration, all processed vertices are the children of the root vertex). As a consequence, vertices processed during a given iteration often reside in the same partition. This leads to all cores accessing the same NUMA node, which creates memory contention [42]. This undesirable effect is even more visible on high-diameter graphs with low-degree vertices, as shown in Figure 3.9 when running BFS on the US-Road graph. The NUMA-aware version is 12× slower than the interleaved version.

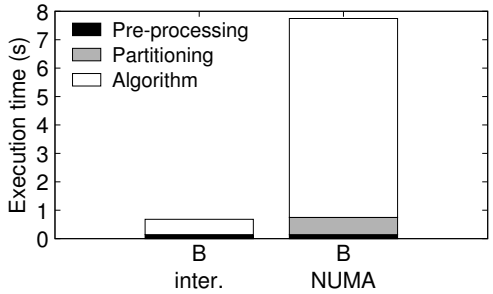


Figure 3.9 – Effect of contention on memory bus on high diameter graphs. Pre-processing, partitioning and algorithm execution time for BFS US-Road graph with memory interleaving vs. NUMA-aware data placement

Summary. NUMA-aware data partitioning has a high pre-processing cost. This cost is amortised for algorithms that run for a long time and that work on most of the data during every iteration. For algorithms that run only for a short time, this may not be the case. For algorithms that only work on a subset of the data, NUMA-aware partitioning may exacerbate memory contention.

3.7 Additional algorithms and workloads

Table 3.5 shows the best solutions for BFS and Pagerank for graphs not evaluated in previous sections. The Twitter graph has a degree distribution similar to that of RMAT, and benefits from the same approaches: using an adjacency list while pushing data for BFS, and using a grid for Pagerank. The US-Road graph leads to slightly different conclusions. The best approach on Pagerank is to use an edge array and not a grid. Since the graph has a lower per-vertex degree than the RMAT and Twitter graphs, the grid data structure reduces only slightly the cache miss ratio, and therefore its pre-processing cost is not amortised.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algo-rithm	Total
BFS	Twitter	Adj. list	Push	5.8	2.3	8.1
BFS	US-Road	Adj. list	Push	0.3	0.5	0.8
Pagerank	Twitter	Grid	Pull (no lock)	23.2	37.8	61.0
Pagerank	US-Road	Edge array	Pull	0.0	1.6	1.6

Table 3.5 – Best approaches in terms of end-to-end execution time for BFS and Pagerank on the Twitter and US-Road graph.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algo-rithm	Total
WCC	RMAT-26	Edge array	Push	0.0	11.0	11.0
WCC	Twitter	Edge array	Push	0.0	19.2	19.2
WCC	US-Road	Adj. list	Push	0.6	56.8	57.4
SpMV	RMAT-26	Edge array	Push	0.0	4.4	4.4
SpMV	Twitter	Edge array	Push	0.0	5.8	5.8
SpMV	US-Road	Edge array	Push	0.0	0.3	0.3
SSSP	RMAT-26	Adj. list	Push	4.4	2.8	7.2
SSSP	Twitter	Adj. list	Push	5.8	4.4	10.2
SSSP	US-Road	Adj. list	Push	0.5	30.7	31.2
ALS	Netflix	Adj. list	Pull (no lock)	0.8	7.7	8.1

Table 3.6 – Best approaches in terms of end-to-end execution time for SpMV, WCC and ALS on different graphs.

In Table 3.6 we report the best approaches for WCC, SpMV, SSSP and ALS, their end-to-end execution time and its breakdown over pre-processing and algorithm execution time.

SPMV is a very short algorithm, and edge arrays are always the fastest approach, as they induce no pre-processing cost.

Intuitively, **WCC** should perform best on adjacency lists, because it is a traversal algorithm (only a subset of the graph is processed during every iteration of the computation), but WCC runs on an undirected graph. We therefore first have to build an undirected version of the graph from the input file. In the case of adjacency lists, an edge has to be inserted in both the outgoing edge array of its source and its destination. Thus, the pre-processing cost for creating adjacency lists is increased. In contrast, no additional pre-processing is required for edge arrays and grids to perform computation on an undirected graph. As a consequence, on graphs with a low diameter, WCC works best with an edge array, because the pre-processing time of building adjacency lists is too high. On graphs that have a higher diameter, like the US-Road graph, WCC needs more iterations to converge, and an adjacency list works best.

SSSP is very similar to BFS, and previous conclusions regarding the trade-offs between algorithm execution time and pre-processing for BFS are applicable to this algorithm as well. The only difference is that BFS discovers a vertex only once, whereas in SSSP a vertex may update its path many times during the computation, leading to an increase both in the number of iterations and the number of vertices active in each iteration.

ALS computes recommendations from a bipartite graph. The left side of the graph represents

users and the other side items being rated. During every iteration, a subset of the graph (the left or right side) is active, and hence adjacency lists are the best data layout.

3.8 Related work

System	Data layout	Iteration model	Push or Pull	Without locks	NUMA-Aware
Ligra	Adj list	Vertex-centric	Push&Pull	Yes	-
Polymer	Adj list	Vertex-centric	Push&Pull	Yes	Yes
Gemini	Adj list	Vertex -centric	Push&Pull	Yes	Yes
X-Stream	Edge array	Edge-centric	Push	-	-
GridGraph	Grid	Grid-cell	Push	Yes	-

Table 3.7 – Overview of multicore graph processing systems that inspired this work and their features.

We cover here only those works that have directly inspired the techniques evaluated in this chapter. For a brief summary of the main features of these systems, see Table 3.7.

Beamer et al. [23, 24] are the first to propose push-pull for BFS. Ligra [119] extends this idea to other graph algorithms. It also uses radix sort for creating adjacency lists. X-Stream [114] introduces edge-centric graph processing in the context of out-of-core systems. GridGraph [149] improves on this idea by organising the edges into a grid. Polymer [138] and Gemini [147] optimise graph processing for NUMA machines. We use their data placement technique in Section 3.6. In addition to the techniques used in Polymer, Gemini adds work stealing to balance work across NUMA nodes.

3.9 Summary

We have presented an analysis of various techniques aimed at improving the algorithm execution time in graph processing systems, and we have explained their impact on pre-processing time. Our main observation is that pre-processing often dominates the end-to-end execution time of graph analytics. Therefore, it is often better to work with simple graph data layouts that induce less pre-processing than to invest time in elaborate pre-processing to speed up the algorithm execution phase. As seen in the previous sections, no approach fits every graph, algorithm or machine. In this section we try to provide a roadmap for choosing between different data layouts and computation approaches.

The first step consists of choosing an appropriate data layout. The layout is chosen based on the algorithm and graph characteristics. Short algorithms, such as SPMV, that complete in one iteration, should use an edge array, as it incurs no pre-processing cost. When the computation works only on a small subset of the graph at every computation step, adjacency lists in push mode improve algorithm execution time. The cost of building them is usually amortised compared to computation over edge arrays, especially on graphs with a high diameter. Other algorithms that run on graphs that have a large average per-vertex degree and iterate over most of the graph at every iteration, may benefit from using a grid, because the grid improves cache locality.

Second, if the machine is a large NUMA machine and the algorithm execution time is predicted to be large, then partitioning the graph to be NUMA-aware is beneficial (Figure 3.8b).

Third, if the data layout and computation approach chosen during the first step allow for execution without locking (e.g., pull mode in grids), then it is always beneficial to remove locks. We do not find any algorithm or directed graph for which switching between a pull mode without locks and push mode is beneficial when looking at end-to-end execution time.

Finally, when pre-processing cannot be avoided, it induces a non-negligible cost, and it should be optimised by using appropriate sorting techniques. We argue that future works on graph analytics frameworks must more carefully consider this trade-off between pre-processing and algorithm execution time.

4 Scale-up Graph Processing in the Cloud: Challenges and Solutions

Until recently, processing large graphs had required large clusters or expensive supercomputers. This has changed with systems such as GraphChi and X-stream that enable the processing of large graphs on a single machine. This chapter focuses on X-Stream, a state-of-the-art graph processing system that can handle a Facebook social network sized graph on a single machine [114].

In this chapter we analyse the performance of X-stream on the Amazon EC2 [9] and Windows Azure [10] cloud environments and demonstrate the following:

- The network becomes the performance bottleneck when processing graphs from remote storage.
- X-Stream's performance can be improved by
  - Using local instance storage for smaller graphs
  - Provisioning the network for better performance
  - Compression to reduce the amount of data transferred

The rest of this chapter is structured as follows. In Section 4.1 we describe the environment our experiments were set in as well as what our use case scenarios were. Section 4.2 covers our hypotheses and supporting experimental results. For generalisation, we provide results from a subset of our experiments when run on Windows Azure [10]. The results are shown in section 4.3. We expand our evaluation to a scale-out setting in Section 4.4, when running a scale-out system presented in [113] on multiple machines. Finally, we conclude in Section 4.5.

Our starting point for this chapter is the implementation described in [114]. We added the necessary environmental support for X-Stream to start up and execute in the Cloud. In addition, we implemented compression for the edge and updates lists. This was straightforward as we access both sequentially. We include results when two compression libraries: the zlib [3] and snappy [4] compression algorithms.

4.1 Experimental Environment

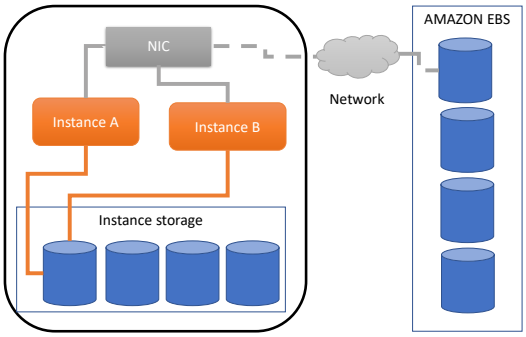


Figure 4.1 – Architecture of the Amazon EC2 cloud.

We used the Amazon EC2 cloud infrastructure for our experiments; the main components of which are shown in Figure 4.1. Each physical machine (host computer) runs multiple virtual machines (instances). The instances share a physical NIC to connect to the network through which they can access remote storage (EBS volumes). Each physical machine also has a number of attached local disks (called instance store) that can be accessed by the running instances. Instance storage is ephemeral, lasting only for the lifetime of the VM, while EBS volumes are persistent across invocations of the VM. The EBS volumes can easily be detached and attached to another instance. In some cases the performance of the EBS volumes can be improved by striping the volumes into a RAID-0 array.

We used Amazon EC2 m1.large instances with 7.5GB of RAM and eight virtual CPUs, each approximately equal to a 1-1.2Ghz 2007 Opteron or Xeon processor. The VM runs a paravirtualised 64-bit Ubuntu Precise (server edition).

Amazon provides a cost-performance trade-off for accessing the EBS volumes as follows:

- Since the network of the instance can be used for various purposes, I/O requests can get delayed. The solution is to run an instance as “EBS optimised”. This reserves 500Mb/s network throughput solely for communicating with the device.
- The disk holding the EBS volume is also shared between multiple tenants. EC2 therefore allows the user to pay for a “provisioned EBS volume”. One can provision 30 IOPS per gigabyte. One I/O operation cannot exceed 16KB. To reach peak provisioned IOPS requires the number of outstanding I/Os to be at least 5 per 200 provisioned IOPS.

There can be many combinations of storage types: instance and EBS, RAID/noRAID, provisioned/not provisioned. We pruned our experimental space down to the following types, eliminating those we knew would be outperformed by a different type on this list:

- Instance store (Instance)

- Two standard EBS volumes organised in a software RAID0 array. No provisioning for either the network nor the IO device (EBS_S2S).
- Two provisioned EBS volumes organised in a software RAID0 array accessed from a non EBS network optimised instance. (EBS_S2P)
- Two EBS standard volumes in a RAID-0 array and an EBS optimised instance (EBS_P2S)
- Two EBS provisioned volumes in a RAID-0 array and an EBS optimised instance (EBS_-P2P)

The number of provisioned IOPS was 1000 (500 for each volume).

4.2 Experiments

4.2.1 Characterizing the EC2 platform

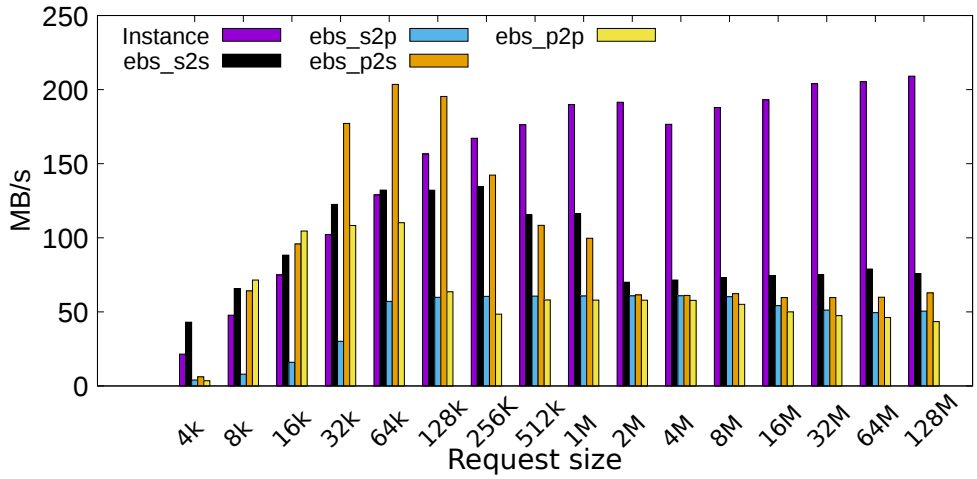


Figure 4.2 – Sequential read bandwidth when varying the request size and configurations.

We first characterised the EC2 platform from the perspective of X- Stream. We used the fio [14] tool to benchmark the bandwidth available to storage from the VM. In order the simulate the workload presented by X-Stream, we did sequential I/O by issuing a single synchronous request at a time varying the size of the request. This approximates X-Stream's disk access pattern that issues sequential I/O of constant configurable size to disk, using asynchronous I/O to ensure that there is always exactly one outstanding request to disk. The results are shown in Figures 4.2 for sequential reads and Figure 4.3 for sequential writes. We drew the following conclusions.

1. The fastest storage is the instance store. Moving to remote storage on EC2 provides persistence and increased space, in return for performance.

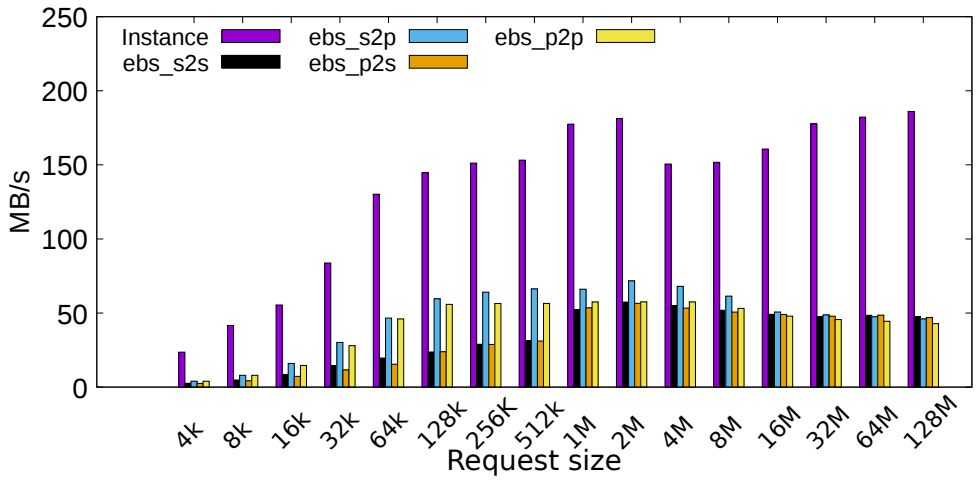


Figure 4.3 – Sequential write bandwidth when varying the request size and configurations.

2. In terms of peak achievable bandwidth for sequential reads, provisioning the network is more beneficial compared to provisioning storage.

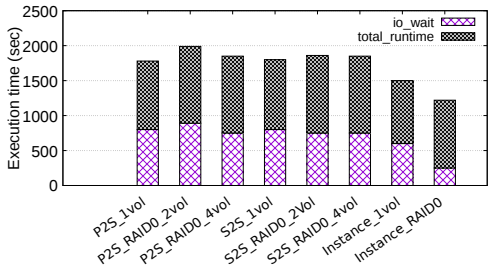
3. In terms of peak achievable bandwidth for sequential writes, provisioning the drives has a marginally better payoff than provisioning the network.

Our interpretation of the results is that for EC2 the EBS volumes appear to be faster than the network. Hence, in the case of reads, requesting large chunks of data becomes counterproductive because the responses are bottlenecked by the network, leaving the drives idle. In the case of writes, larger request sizes are better as there is no data to send back and there are less network overheads for larger request sizes.
A corollary of this is that for reads, provisioning IOPS on the EBS volumes is wasteful as the network is the bottleneck on the return path. More benefit is obtained by provisioning the network. On the other hand, for writes, the drives cannot keep up with the network. This is likely a consequence of the EBS volumes being stored on SSD that have poor write performance as compared to read performance.
Noting that the peak write bandwidth of provisioned volumes is only marginally higher, we conduct further experiments without provisioned volumes.

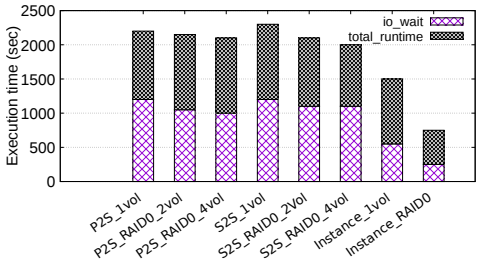
4.2.2 X-Stream baseline performance

For our tests we generated a synthetic undirected graph using the RMAT [33] generator. We used 25 as the scaling factor making the number of vertices 2^{25} (32Million) and the number of edges 2^{29} (512Million). The I/O included reading in the edge list and writing out the updates for the scatter phase as well as reading in the updates for the gather phase. Since the entire vertex set fits into memory, X-Stream creates only one partition. The vertex state is in memory during the entire run.

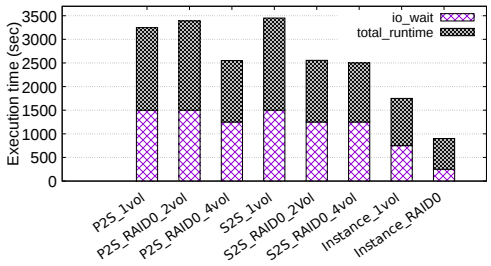
We ran BFS, Connected components (BFS forest) and Pagerank on the graph and the results are shown in Figure 4.4. In addition to the storage configurations described above, we also



(a) BFS

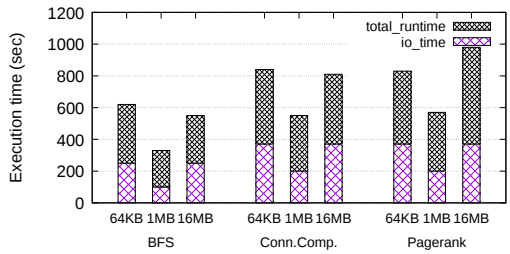


(b) Connected Components

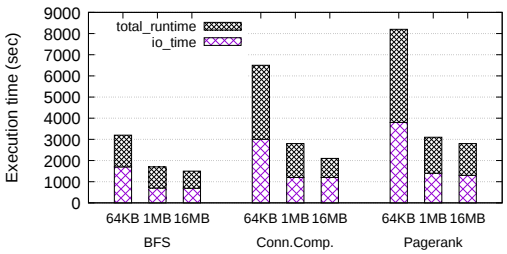


(c) Pagerank

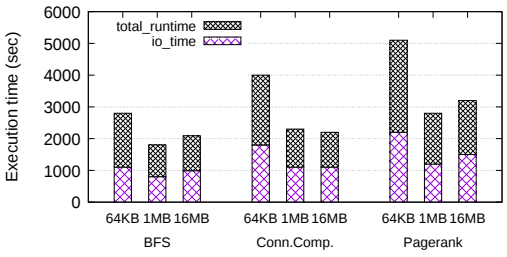
Figure 4.4 – X-Stream performance for BFS, Connected Components and Pagerank on Amazon.



(a) Instance store



(b) EBS_S2S



(c) EBS_P2S

Figure 4.5 – X-Stream performance when varying the request size on different configuraitons.

experimented with increasing the number of volumes in the RAID array. The sequential access nature of X-Stream means that it can take advantage of the extra bandwidth offered by RAID arrays on physical machines. We therefore wanted to explore whether the same was also true in the cloud. The relative performance of EBS_P2S, EBS_S2S and Instance storage are

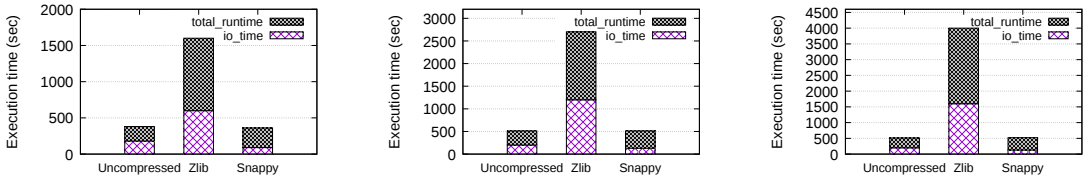
consistent with the fio rest results. Instance storage performs the best followed by EBS_P2S and finally by EBS_S2S. *Moving from local instance storage to remote storage on EC2 affects the performance of graph processing by as much as* 4×.

Another conclusion from the results is that while RAID helps with local instance storage, it has very little effect with remote storage. The extra bandwidth achievable by sequential access to the RAID volumes is unavailable at the X-Stream end, due to the bandwidth limitations of the intervening network. The improvements with RAID on local instance storage are consistent with X- Streams' performance on physical machines [114], with performance improvements of 50% when using two disks in RAID0.

X-Stream has a key configuration parameter that dictates the size of requests made to storage. By default (and in the results presented thus far) it is set to 16MB, as that was the optimum setting for physical machines with attached storage.

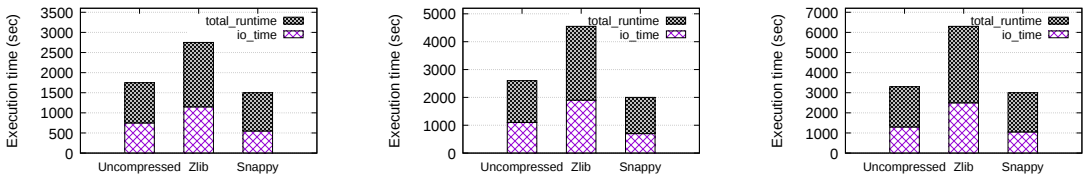
The fio test results however suggest that peak throughput is obtained using 1MB requests for sequential writes, and 64KB requests for sequential reads. Therefore, we decided to do another set of tests for each of the benchmarks with these I/O chunk sizes. The results are shown in Figures 4.5a, 4.5b and 4.5c. We can conclude that a 1M I/O chunk size gives the best results with EC2 infrastructure as opposed to 16M with physical infrastructure. It is possible that even better performance might be possible by having different chunk sizes for the input and output paths. Unfortunately, X-Stream does not support this at the moment. Using X-Stream in the cloud would be a key motivator for adding such support.

4.2.3 Compressed I/O



(a) BFS (b) Conn. Components (c) Pagerank

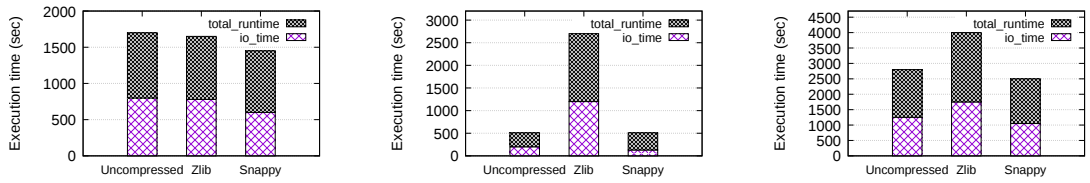
Figure 4.6 – Compression on instance store.



(a) BFS (b) Conn. Components (c) Pagerank

Figure 4.7 – Compression on EBS_S2S.

Provisioning the network comes with a trade-off of 9% in price increase for the instance for



(a) BFS (b) Conn. Components (c) Pagerank

Figure 4.8 – Compression on EBS_P2S.

12% of performance improvement in return.

Therefore we wanted to explore mitigating the bottleneck by storing and retrieving compressed sequences of edges and updates. We analysed the gain with compressed I/O on both, EBS_S2S and EBS_P2S.

The sequential access nature of X-Stream makes such compression possible. We added compression support to the X-Stream codebase from [114]. We experimented with two different compression algorithms: zlib [3] and snappy [4]. Zlib provides somewhat better compression ratios in return for increased latency to decompress and compress blocks.

In Table 4.1 we display the compression ratio of the synthetic graph we have used thus far in the chapter, as well as the Twitter [77] real world graph. The compression ratio is much better with the Twitter dataset as the edges in the graph are sorted by source. The synthetic graph and the Twitter dataset therefore represent opposite ends of the spectrum in terms of compressibility of edge lists.

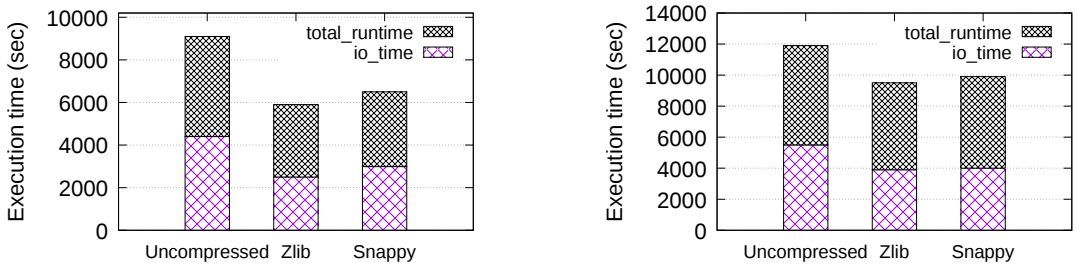
We ran experiments in this section with a 1M I/O chunk size, guided by our results from the previous section. We first consider the results displayed in Figures 4.6,4.7,4.8, obtained when running on the synthetic graph. From these results, we draw the following conclusions:

It is possible for I/O bandwidth to exceed the capability of in-memory compression and decompression. This is a somewhat counterintuitive result. However, sequential accesses to SSDs used in EC2 is extremely fast, providing as much as 180 MB/s. On the other hand we have observed in separate tests that, zlib is unable to handle streams at more than 150 MB/s. This is why compression provides no benefit with instance stores. Snappy is faster than zlib and is able to keep up with the EBS volumes thereby providing benefit.

Graph name	Uncompressed	Zlib	Snappy
Synthetic graph	6GB	5.2GB(13.3%)	6GB(0%)
Twitter	17GB	11GB(35.3%)	14GB(17.6%)

Table 4.1 – Zlib and Snappy compression ratios for different graph types.

Benefits from compression are visible even if the input edge list is not compressible. Snappy provides benefits for synthetic graphs on EBS volumes even though the edge list is not compressible. The reason behind this is that the updates are compressible. This reduces the time needed to write them out, and read them back in.



(a) BFS

(b) Conn. Components

Figure 4.9 – The impact of compression of the Twitter graph when running BFS, on EBS_S2S and EBS_P2S.

Next, we consider breadth-first search over the real-world Twitter graph in Figure 4.9. This graph has a far better compression ratio for the edge list as it is sorted by vertex ID. Therefore, even with zlib, the compute overhead caused by compression is amortised by reducing the amount of I/O done.

In general, compression is an effective technique for improving the performance of graph processing on EC2 using X-Stream. For our tests we were able to improve performance between 12%-30%.

Furthermore, we saw that when the compression is efficient, like with zlib on the Twitter graph, the provisioned instance is cheaper by approximately 40%.

4.3 Windows Azure

In order to generalise our findings we ran a subset of the tests performed on Amazon EC2 on Microsofts' cloud platform. The platform itself offers different options from EC2 and categorises instances and storage in a slightly different manner. The first difference is that there is no provisioning of either network or IOPS but rather, they guarantee 500 IOPS for each attached disk. In this section we do not go into the specific details of the offered options. We chose the environment that is most compatible with what we had on Amazon.

The tests were run on Ubuntu 12.04 on Extra Large instances that offer 8 CPU cores and 16GB of RAM. Since this is larger than what we had on Amazon, we restricted X-stream to use the same amount of RAM as on EC2. We ran BFS, Connected Components and Pagerank on the following combinations of storage devices:

- Local storage
- One external disk of 20GB
- Two external disks with 20GB each striped in a software RAID-0 array

Before running X-stream, we ran the same fio tests as on EC2 in order to determine the best

request size. The results are in agreement with the results on EC2 in the sense that the local storage is 6X faster for sequential reads, and 2X faster for sequential writes. The request size with which peak bandwidth was achieved is 32MB. In order for our comparison to be fair we chose this as our I/O chunk size on Azure.

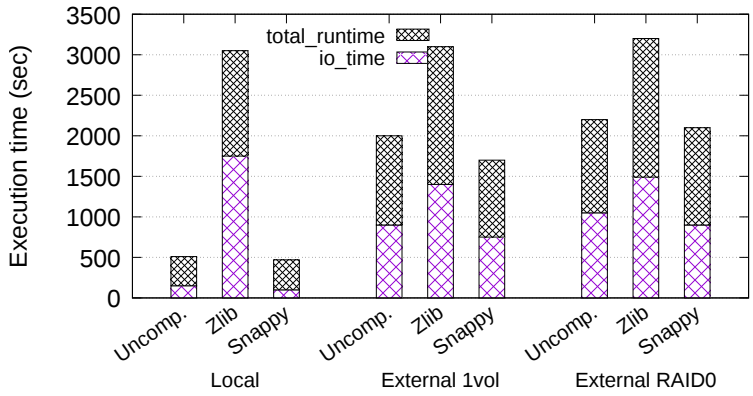


Figure 4.10 – X-Stream running on Windows Azure, when varying the storage type and compression schemes.

In Figure 4.10 we show the performance gain of compression, on both local and external storage. The conclusions are similar to those on EC2. Here too, the network is a limiting factor and the speedup gained by attaching new disks and striping them into RAID array is only marginal.

4.4 Scaling-out on secondary storage

To evaluate the benefits of reducing the amount of I/O in distributed systems, we implemented compression in Chaos [113]. Chaos is a scale-out system, that leverages the aggregated storage of many machines. It can process bigger graphs on fewer machines, reducing the overall cost in $.
In addition to compressing the I/O requests, we compress network packages and combine messages destined to the same vertex.

The conclusions when comparing the two compression algorithms were the same as for X-Stream. When compressing the network packages, zlib was always slower than snappy.
Combining requests destined to the same vertex improved the performance only marginally. The reason is that in practice, the message buffers containing messages to remote machines, or I/O requests, are small - 16MB. Since the graph is sparse, and the buffer itself is small, the chance of having many messages to the exact same vertex in these 16MB is very small.

Using more machines, does not only increase the scale of the graph that can be processed, but allows the user to process the same graph faster. Chaos can leverage the combined instance store of the two machines, while X-Stream is forced to scale-up to EBS storage due to the small

size of local Instance storage. X-Stream takes 7573s to run Pagerank on RMAT-28 from Amazon EBS, while Chaos takes 3360s when running on 2 machines, using instance store.

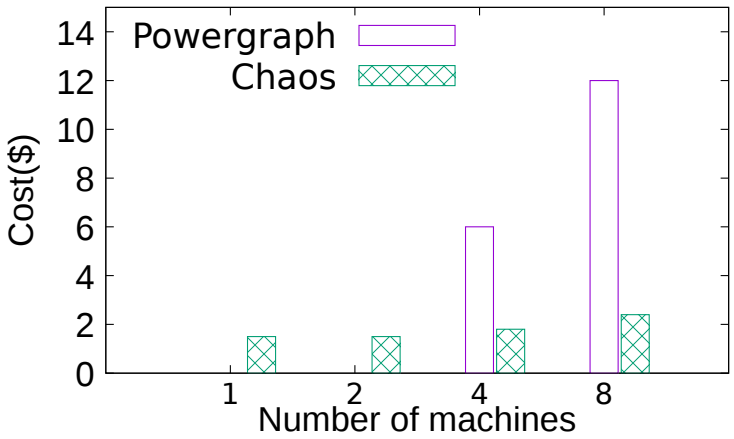


Figure 4.11 – Cost comparison of running in-memory and out-of-core analytics in the cloud. We run Pagerank on RMAT-28. The cost is equal to the running time of the application multiplied by the hourly price, rounded up. Missing entries for Powergraph mean that Powergraph could not process the graph.

In the cloud, Chaos is only 1.2× slower than Powergraph [55], a distributed in-memory system, while reducing the cost of computation.

Figure 4.11 shows the cost savings in $ when running in the cloud from secondary storage.

We also ran Chaos in a local rack, to evaluate the benefits of compression on a fast 10Gb/s network. In overall time, the CPU overhead of compression was not amortised by the reduced I/O.

4.5 Summary

In this chapter we have evaluated processing of large-scale graphs using X-Stream in the cloud and discussed the benefits and downsides of the cloud environment. Cloud storage is attractive due to the easy availability of large amounts of storage, something that is difficult to achieve with physical machines. One can therefore potentially scale the problem size tackled using the elasticity of the cloud.

Our most important conclusion is that the network is a serious bottleneck when accessing remote storage thereby limiting such scalability. This means that although cloud services can provide large amounts of storage for graphs, there is a penalty on performance due to the network to storage becoming a bottleneck.

For the specific case of EC2, we found a set of partial mitigations to this problem. Provisioning the network helps to improve performance by relieving the network bottleneck. A more cost-effective way of improving performance is through compression to reduce the amount of data moved on the network. We showed that it is necessary to use a compression algorithm with

adequate performance to keep up with streaming bandwidths available on EC2.

A more effective solution for scaling graph processing in the cloud is to distribute X-Stream execution across multiple virtual machines thereby gaining aggregate bandwidth to storage. We demonstrate this by adding compression to Chaos, and observe a significant reduction in the number of machines needed to run big graphs. However, these optimisations were beneficial only in the cloud, and their overheads were amortised in a local rack, with fast network.

5 Optimus: Transforming for efficient single machine NVMe-based out-of-core graph processing

Out-of-core graph engines usually assume that storage is the performance bottleneck and try to optimise disk IO at the expense of more computation or limited data access patterns.

In this chapter we discuss the implications state-of-the-art fast storage devices have on the design of a graph processing system.

We present an out-of-core system that processes large graphs faster than state-of-the-art out-of-core systems, when running on PCIe attached NVMes. We support fine grained access granularity by representing the graph in a compressed sparse row (CSR) format, while at the same time removing fine tuned I/O layers.

However, we demonstrate that no layout fits all algorithms. In addition to the adjacency list representation, Optimus provides implementations of all algorithms when using a variation of the grid layout. We show the benefits from using this data layouts for a subset of algorithms, and when the state of the graph does not fit in DRAM. Optimus is designed as a semi-external system, similarly to Flashgraph [144], storing the state of the vertices in DRAM, while the edges are streamed from storage. While in many cases the state always fits in the available DRAM, we add support for fully external mode, for algorithms for which the state cannot fit in DRAM.

The benefit of transforming the graph into a different layout can be substantial in the algorithm execution time. This benefit comes with the additional cost of time needed to create the chosen data layout [87]. As in Chapter 3, we evaluate the cost of creating the different data layouts out-of-core, when using techniques implemented in existing systems, and design an optimised solution, with 1.6× faster processing time.

Since most existing out-of-core systems assume that CPUs can process data much faster than the speed at which it is delivered from storage, they tend to be CPU bound rather than IO bound on NVMes, since this assumption no longer holds true.

We therefore present the following contributions and findings in this chapter:

- A system that provides fine grained access granularity for algorithms computing on small parts of the graph, while not impeding on the performance of algorithms that process the entire graph. Optimus is inspired by both in-memory and out-of-core engines, outperforming the state-of-the-art up to 2×. It is only 17% slower than a system which highly optimises the I/O accesses to run at higher bandwidth.

- An API that allows the user to transform the adjacency list representation of the graph into a different layout to maximise the sustained bandwidth. The layout also allows for vertex state to be stored on the NVMe, in case it cannot fit in the provided DRAM.

- An analysis on the cost of creating different graph representations using existing in-memory and out-of-core algorithms.

- Optimised implementations for all pre-processing algorithms which use the underlying NVMe more efficiently, outperforming existing approaches by 1.6×.

5.1 Motivation and background

In this section we first describe the two types of storage we used for evaluation, and evaluate their raw performance. We then show how existing out-of-core and in-memory systems perform when the data is stored on the NVMe. The goal of the section is to determine whether the NVMe behaves like the SSD or DRAM, or neither.

Compared to traditional SSDs, state-of-the-art PCIe NVMe devices have a much higher bandwidth. We evaluate the bandwidth of a 480GB Intel DC S3500 Series SSD and an 1.5TB Intel Optane 900P attached via PCIe. Both drives are attached to a dual socket Intel Xeon E5-2690 server with a 2.6GHz clock speed, and 28 threads per CPU (56 in total). Since Flashgraph cannot run with a number of threads that is not a power of 2, we run all experiments and all systems with 32 threads.

	4MB seq. bw (Queue depth)	**4KB rand bw (Queue depth)**	**4K bw / 4MB bw**
SSD	377MB/s (4)	272MB/s (32)	72%
NVMe	2495MB/s (4)	2220MB/s (32)	89%

Read bandwidth

	4MB seq. bw (Queue depth)	**4KB rand bw (Queue depth)**	**4K bw / 4MB bw**
SSD	360MB/s (4)	181MB/s (32)	50%
NVMe	2202MB/s (4)	2185MMB/s (32)	99%

Write bandwidth

Table 5.1 – Read and write bandwidth

Table 5.1 shows the performance of these two drives measured by fio [14] with libaio as engine and bypassing the page cache. We evaluate the sequential bandwidth when issuing

4MB requests. To evaluate the achievable bandwidth for random I/O, we issue smaller, 4KB, requests, as this is a common scenario in graph processing.

The NVMe is able to sustain 90% of its sequential bandwidth when doing random I/O with 4KB requests, contrary to the SSD. Due to its write-in-place technology, there is virtually no performance degradation when comparing random and sequential writes on the NVMe.

While the difference between random and sequential I/O on SSDs is much lower than it was on HDDs, it is still substantial. In the section below we discuss how this impacts the design of state-of-the-art out-of-core systems.

5.1.1 Existing systems and NVMe

Flashgraph [144], Graphene [57], Gridgraph [149] and Mosaic [85] all rely on different techniques to optimise I/O. In this section we compare their performance when running on the NVMe, with the performance of an in-memory implementation of the same data layout - RAMCode. We modify RAMCode to store and processes the edges from the NVMe by memory mapping them, instead of loading them into memory. The amount of DRAM given to the application depends on the algorithm, but it is always smaller than the size of the input (edges).

Flashgraph [144] represents the graph as an *adjacency list*. The degree of vertices are known and for a subset of vertices the pointers to their neighbours on disk are stored as well. For the remaining vertices, the starting position of their neighbour list within the edge array is computed based on its degree, and the pointers already stored.

As the size of the neighbour lists of individual vertices varies, Flashgraph has a separate I/O layer that receives requests for neighbours from compute threads, and merges neighbouring requests to increase the request size where possible.

RAMCode-adj represents the graph as an adjacency list, while RAMCode-grid uses a grid representation of the graph. In Figure 5.1, we plot the running times of Flashgraph, GridGraph and Mosaic on RMAT-29 when running BFS and PR. We run Mosaic without the accelerators. The size of the input is 64GB for Gridgraph and RAMCode-grid, while other approaches compress the data into adjacency lists, or tiles, reducing the input size by 2. The applications are given 8GB and 15GB of DRAM for BFS and Pagerank, respectively. The only exception is Mosaic, which requires more memory and runs on 20 and 40GB of DRAM for the two algorithms. We compare their performance to RAMCode, which shows the behaviour of in-memory optimisations when the input is read from storage.

For BFS, which traverses only a small part of the graph at a time, using an adjacency list leads to significantly shorter running time. Simply porting the in-memory implementation, without merging the I/O requests is orders of magnitude slower compared to all systems.

However, for Pagerank, which accesses the entire graph at every iteration, the I/O layer and the

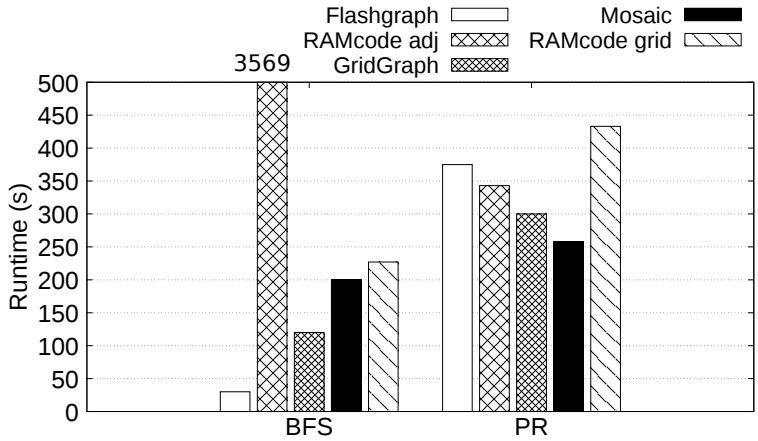


Figure 5.1 – Running time of BFS and Pagerank(10 iter), on different systems on RMAT-29. The graph is stored and processed from an NVMe. RAMCode-adj(-grid) are in-memory implementations of the data layouts ran out-of-core.

pointer chasing for adjacency lists, makes Flashgraph 1.3× slower compared to Mosaic. More surprisingly, simply porting the in-memory implementation of Pagerank over adjacency lists to storage, outperforms the I/O optimised implementation of the same layout. To understand why this happens we looked at the bandwidth sustained by the two approaches. Flashgraph sustains a bandwidth of 2GB/s for Pagerank, compared to 1400MB/s of RAMCode-adj.

Closer profiling of the applications revealed that the work done by the I/O layer in Flashgraph, introduced delays in issuing the requests. These were not amortised compared to RAMCode-adj. Flashgraph was no longer I/O bound on the NVMe.

SSD vs. NVMe . We ran the same experiments on the SSD to determine whether any of these observations are NVMe specific, and the result of Flashgraph not being I/O bound on the NVMe. As expected, the random access of RAMCode-adj only amplifies its slowdown compared to Flashgraph for BFS. But on Pagerank, we observe that RAMCode-adj is an order of magnitude slower compared to Flaghraph, taking 2100s to run 10 iterations of Pagerank, compared to 400s.

Summary The findings in this section show that there is no solution that fits all algorithms. We see that I/O friendly data layouts such as the grid, perform poorly on traversal algorithms, doing more I/O than needed. But, on the NVMe, the I/O layer required to make the adjacency list performant out-of-core, causes significant overhead for other algorithms. Since the penalty is higher for BFS, in the next section we show how the adjacency list can be made performant for sparse-matrix multiplication algorithms, while not impeding on the performance of BFS.

5.2 Adjacency lists in Optimus

In this section we describe how Optimus represents the graph using an adjacency list, the corresponding programming model and optimisations.

Graph representation The edges, consisting only of destination vertex IDs, are stored contiguously on disk. We store an index with a pointer to the first neighbour of each vertex. The vertex degree, and position of the last neighbour are computed on the fly. This array can be memory mapped from storage or fully loaded into DRAM. We observed a minimal 10% performance degradation when the index is memory mapped. The state of each vertex is stored in DRAM. When it cannot fit in DRAM, we provide an alternative data layout discussed in 5.5.2.

During algorithm execution time, the edges are memory mapped, and threads access the file as in-memory when accessing the neighbour list of a vertex. We give applications enough DRAM to place the state and up to 2GB for buffering I/O requests.

Programming model During one iteration of the algorithm, vertices are activated, and put in a workqueue to be processed in the following iteration. Threads fetch a number of vertices from the workqueue, read the position of their neighbours list from the index, and update the neighbours' state.
For example, BFS starts off with one active vertex - the root of the BFS tree. During the first iteration, all its neighbours are added into the tree and put in the workqueue for the next iteration. For each vertex in the queue, threads read all their neighbours, and, in case they are not already in the tree, append them to the workqueue of the next iteration. As threads discover and add vertices to the tree at random, they are placed into the queue without order. As a corollary, when fetching the neighbours of vertices in the queue, we have no locality of access.

The fundamental role of the I/O layer is to achieve I/O locality and increase the request size. The gap between RAMCode-adj and other approaches, shows the importance of having such locality, even on state-of-the-art storage devices. Flashgraph resolves this issue by sending requests for neighbours to an I/O layer, which orders them by vertex ID, and merges small neighbouring requests before issuing an I/O request to the filesystem. Furthermore Flashgraph relies on SAFS [143]. SAFS issues direct asynchronous I/O requests to fully utilise the parallelism of the NVMe, and provides a custom cache.

Optimus achieves this locality using much simpler techniques, without a specialised filesystem, and without the overhead of a dedicated I/O layer: *madvise*, sorting the workqueue, and replacing atomic operations to mark active vertices with a bitmap.
We first used *madvise* to hint the prefetchers that the graph is accessed at random. While the runtime improved, it was still one order of magnitude higher than the performance of Flashgraph. We therefore tried to achieve I/O locality simply by sorting the workqueue.
For traversal algorithms, such as SSSP, we further leveraged the insight from Ligra [119], that certain iterations of BFS traverse a large part of the graph to further improve the performance of these iterations.

Namely, when adding vertices that are to be explored in the subsequent iteration, a thread has to take a lock on the queue, in order to not add the same vertex twice. This overhead is clearly

RAMCode-adj	Madvise	QSort	RadixS	Optimus*
3925s	262s	112s	42s	37s

Table 5.2 – Running time BFS in seconds in Optimus when using different optimisations on RMAT-29 with 8GB of DRAM. *Optimus uses a RadixS(ort) to sort the workqueue and removes locks in big iterations.

amortised for the overall computation. However, during iterations where many vertices are added to the queue, there is more contention on the locks. To avoid this, for iterations with many active vertices, we do not add vertices into the queue, but rather use a bitmap to mark a vertex as active for the next iteration. When using a bitmap, we pay the cost of testing the bit for every vertex, and computing on the vertex if the bit is set. For this reason, the bitmap benefits only iterations where many vertices are active, as we would process them in any case. In addition to the bitmap marking vertices to be processed in the subsequent iteration, we need to keep track of vertices activated in the previous iteration, as we want to explore only their neighbours. By having two bitmaps, just like we have two work queues, we only explore neighbours from vertices that were activated in the previous iteration.
Once the number of vertices activated in an iteration drops, based on the bitmap, we fill the workqueue with vertices activated, and continue computing as before. We noticed an increase from 1400MB/s to 1980MB/s of sustained bandwidth during the two iterations of BFS for which we removed the locking.
This approach is similar to the push-pull model in Ligra [119], but without the overhead of creating and storing the list of incoming edges for every vertex.

Load balancing The in-memory approach described in [87] achieves good load balancing by giving threads a small chunk of vertices at a time. If, towards the end, there are idle threads, the chunk size is decreased. This has shown to have good performance in-memory, and for BFS, out-of-core as well.

However, for Pagerank, we observed the algorithm to be much more CPU bound than BFS, and the slight load imbalance caused by this simplistic scheme, had a significant performance impact on the algorithm when ran out-of-core. We improve on this scheme by smarter load balancing for global such as Pagerank. The work is assigned to threads before the iteration starts using a custom partitioning scheme. Vertices are assigned to threads by taking into account their degree, where threads end up processing a similar number of edges. In the case of Pagerank, this can be done at the very beginning only once, since the number of active vertices does not change between iterations.

Evaluation Table 5.2 shows the performance of BFS on RMAT-29, with 8 GB of DRAM, with all the previously mentioned optimisations. We show the performance when sorting the workqueue using quick sort and radix sort. While the I/O locality achieved with sorting, improves overall performance, quick sort still causes an overhead compared to radix sort. This is solely due to the speed of the NVMe, since on the SSD, the difference between the two sorting techniques is only 6%. The final running time of BFS is only 1.2× slower than that of Flashgraph.

The running time of Pagerank is 223s, which is only slightly faster than Mosaic, but outperforms GridGraph and Flashgraph by 1.2× and 1.6× respectively.

Summary We show that the optimisations Optimus applies to in-memory implementations are sufficient to match the performance of highly optimised out-of-core systems for traversal algorithms, while outperforming systems whose data layouts are more I/O friendly for Pagerank.
Note that Optimus is not able to match the performance of Flashgraph on slower devices, such as our SSD, while being fully on par with it when they both run in-memory. In memory, the work-queue sorting done by Optimus is not amortised by the gains in locality.

This demonstrates that the NVMe has a behaviour different from both memory and SSDs, and warrants a new design.

5.3 Grids in Optimus

While the adjacency list implementation already outperforms the state-of-the-art systems, in terms of efficiently utilising the storage, it is still behind I/O friendly layouts such as the grid, or its variation, tiles used in Mosaic. Both Mosaic and GridGraph reach up to 2.5GB/s of bandwidth, while the average sustained bandwidth of adjacency lists is 2.0GB/s.

To evaluate how big the gains of such a data layout can really be, we implemented both, a grid and an optimised compressed grid.

To create a compressed representation, for a graph up to 2^{32} vertices, we define the number of rows and columns (P) such that $\frac{NoVertices}{P}$ is 2^{16}. We then relabel the edge endpoints to their 16-bit counterparts since within a cell they can be ID-ed with at most 16 bits. This reduces the size of the input by two. For 64-bit vertex identifiers, we relabel them to 32-bit identifiers. Due to space constrains, since the compressed grid always outperformed the grid, we refer to this representation when talking about the grid.
Graph representation The graph is stored as a list of edges with compressed source and destination, contiguous on disk row by row. Optimus stores an index with the starting offset of each cell. The index can be loaded in DRAM or memory mapped. Since it is typically small in size (512MB for RMAT-29 which has 64GB), we found it was cached most of the time, and the performance degradation with a memory mapped index was even smaller than with adjacency lists. However, for very large graphs, where we need many cells to allow for compression, the index can grow up to 30GB. To avoid storing it in memory, we memory map the index. The index is read between iterations and not used during computation. While memory mapping it can introduce slow down between iteration, while assigning work to threads, it does not impact the actual compute time. The computation is the dominating part of the algorithm execution, and memory mapping the index has very little impact on the overall running time of the algorithm. We discuss this in more detail below.

Programming model and load balancing Mosaic [85], GridGraph [149], and the in-memory implementation of the grid [87], process the graph one cell at a time, iterating first over all the columns in one row, before proceeding to the next. This allows for lock-free computation when a thread is assigned one cell at a time, since all the destination vertices that belong to the cell are updated only by one thread. On the NVMe, we found it more beneficial to have a better load balance among threads by allowing them to move on to the next row without waiting for all the cells within a row to be processed. Mosaic achieves better load balance by creating tiles upfront. However, creating tiles takes much longer, and requires much more DRAM than the compressed grid in Optimus.
We define an ideal number of edges per thread by dividing the total number of edges with the number of threads. Optimus uses the index to compute the size of a cell. If a cell has more edges than the ideal number, we assign it to multiple threads for processing. Each thread is assigned a number of work items. A work item consists of 16-bit row and column IDs, and 64-bit starting and ending offset. Threads fetch edges at the given offsets in the edge file. We need to explicitly store the row and column IDs so that we can uncompress the vertex IDs at a particular offset. The ID is re-labeled based on the cell it is in, which cannot efficiently be obtained from the offset itself. Since the thread accesses edges at the offsets stored in the work-items, we do not require the index to be read during computation. This is especially beneficial for large graphs where the index size is not negligible.
For Pagerank, we do this once before the first iteration. For BFS, WCC and SSSP, this is done between every iteration. As GridGraph and Mosaic, we skip rows that have no active vertices, thus not propagating the value to any neighbour. This can significantly reduce the amount of I/O, especially at the tail iterations of BFS.
To keep track of vertices activated in an iteration, we provide two bitmaps. One with vertices activated in the current iteration, and one to remember vertices activated in the previous iteration.

5.4 Graph transformation

Ideally, we want to be able to switch between a graph representation depending on the algorithm. Before deciding on the feasibility of transforming the graph on the fly, we have to understand the cost of creating a data layout. Systems often attribute this to a once payed pre-processing cost, that is amortised during computation. The actual cost of the pre-processing has not been the subject of recent papers, and optimising it has not been the priority of state-of-the-art systems.

We implemented approaches used both in-memory [87] and out-of-core [45, 149] to create the different data layouts from an edge array, and found that none of these algorithms works optimally on the NVMe. The in-memory algorithms read and write data randomly at a fine granularity (less than a page size) and most of the data fetched from storage is not used. Disk algorithms were made for devices with low peak bandwidth and are CPU-bound on NVMes. We therefore propose three new algorithms to pre-process graphs on NVMes.

5.4.1 Adjacency lists

Adjacency list using a count sort The simplest way to create an adjacency list from an edge array is to perform a count sort. This approach has been used in memory [87]. The input edge array is mapped into memory and threads read memory mapped data to compute the degree of all vertices. The offset of the vertices in the output file is then computed. Then the edge array is read a second time and edges are written at the offset of their source vertex in the memory mapped output adjacency list file. This approach is in theory optimal in terms of IO: it only requires reading the input file twice and writing the output file once. Memory mapped data is read sequentially, but writes are performed to as many offsets as there are vertices in the graph.

Adjacency list using a radix sort Previous work [87] has reported that sorting an edge array using a parallel radix sort was the fastest way to create an adjacency list in memory. To evaluate the feasibility of this approach out-of-core, we reuse the code of [87, 119], and change the code so that the input file and temporary data is mapped into memory.

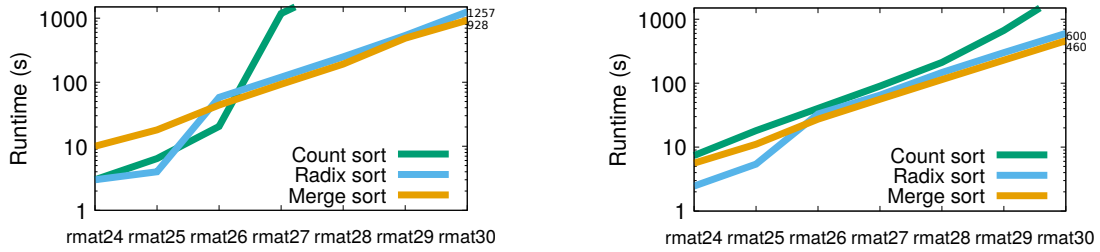
In phase 1, each one of N threads gets a 1/N section of the input file and counts the number of edges with the right prefix in its section. In phase 2, each thread rereads its section of the input file and gets 1/N of the output file to write its result. Each thread separates edges in 256 buckets that it writes in its section of the output file. In phase 3, the buckets of the different sections of the output file are merged in parallel. The merged buckets are then recursively sorted. It takes 4 iterations to sort a graph. Every iteration requires reading the graph 3 times and writing it twice (12 reads and 8 writes in total). Reads are sequential, and writes are performed at the offsets of buckets (256*number of threads offsets).

Adjacency list using a disk-optimised merge sort We use the external merge sort implemented in the STXXL library, a state-of-the-art external sort library that has optimal I/O volume [45]. The sorting is done in two phases. First, as 1GB of data is loaded into memory. The data is sorted and the 1GB chunk of sorted data is written on disk. The second phase sequentially merges these partially sorted chunks. STXXL is slower than the radix sort when the graph fits in DRAM but becomes faster as memory space is constrained. STXXL is the fastest approach but it is CPU bound: neither the sorting of chunks nor the merge is able to saturate the device bandwidth. Disk algorithms are thus inefficient on NVMes.

The radix sort is faster than the count sort despite reading and writing more because writes are less random. However, the radix sort used in [87] is still suboptimal: (i) it reads and writes the graph 20 times in total and (ii) the merge of per thread buffers still causes trashing in the page cache. For instance, to sort 64GB of data, the radix sort ends up reading over 2000GB from storage. In-memory algorithms are thus not efficient when applied out of the box on NVMes.

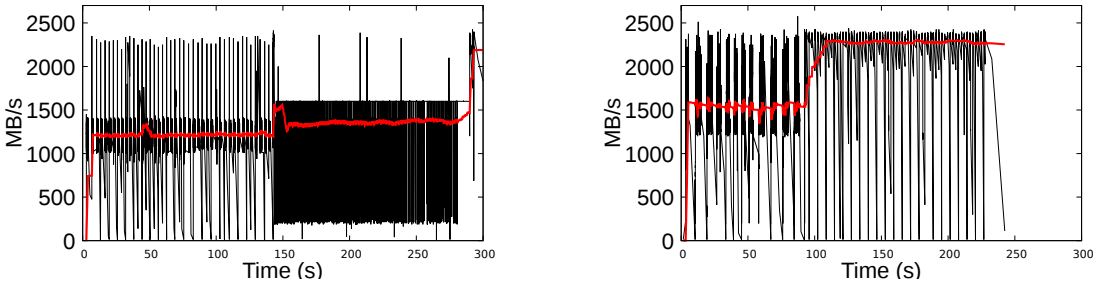
Algorithm	Existing approach (s)	Opt. method (s)
Count sort	49256	1260
Radix sort	1257	720
Merge sort	928	558

Table 5.3 – Time to sort an rmat30 graph with different sorting techniques.



(a) Existing approaches (b) Optimus

Figure 5.2 – The time to create the adjacency list representations using the existing and optimised approaches presented in previous works. We scale the graph size and run with 8GB of DRAM Rmat(X+1) is double the size of rmatX. Axes are in logscale.



(a) Merge sort (STXXL) bandwidth (b) Optimised merge sort bandwidth

Figure 5.3 – Read bandwidth of the optimised merge sort of the RMAT-29 graph. In red, the moving average over the entire running time.

Optimising algorithms for NVMe

In the previous section we have shown that techniques used to pre-process data on NVMe are either CPU-bound, trash data from the page cache, or are IOPS bound. In this section we present three optimisations of the previously presented sorts that improve performance on NVMes.

Adjacency list using an optimised count sort The main problem with the naive count sort is that edges are written at a fine granularity to random locations on the disk. To circumvent this issue it is possible to draw inspiration from disk approaches and perform the count sort in two steps by splitting the output file into 1GB chunks. First, edges are read from the memory mapped input file and appended sequentially in the correct chunk. All chunks are memory mapped, and the page cache flushes previously written data as more edges are appended to the chunks (this doesn't induce page trashing as chunks are written sequentially). Then each chunk is memory mapped, sorted, and sync'ed to disk. We split the data in parallel and sort every chunk in memory using the parallel radix sort presented in [87]. The optimised count

sort performs 2 reads and 2 writes of the graph. All accesses to disk are sequential.

Adjacency list using an optimised radix sort The main problem with the in memory radix sort used in [87] when data doesn't fit in DRAM is (i) extra copies and (ii) page trashing during the merge of buckets. We changed the algorithm to avoid the need of merging buckets. In memory, threads were given different sections of memory to write data to avoid false sharing. We removed this optimisation: in our optimised radix sort threads write edges at their right offset directly (i.e., the offset edges would have after the merge). We compute the correct offset of the per thread buffers during phase 1. Having edges written at their correct offset instead of different sections of the output file results in more interference during the sort: if a bucket is small, multiple threads may write to the same page. This false sharing at page level can cause inefficiencies in the page cache (the same page might be read and flushed multiple times). In practice we observe that the buckets are big enough and that the interferences introduced during the sort are minimal. Contrarily to the optimised count sort, the graph is not split into chunks, so every iteration of the sort reads and write the whole graph. We rely on the page cache to prefetch memory mapped data and to flush dirty pages to disk when memory is constrained. The optimised radix reads the file twice and writes it once per iteration (8 reads and 4 writes in total).

Adjacency list using a parallel merge sort STXXL (disk merge sort) is CPU-bound. To circumvent the issues of STXXL, we reimplemented a parallel external merge sort that uses efficient in-memory sorting and merging techniques. Just as for STXXL, data is loaded in memory in chunks of 1GB to be sorted. We changed the parallel quick sort used by STXXL to a parallel radix sort. Chunks are then written to disk. We then merge the chunks using a parallel algorithm. We start by allocating a 1GB buffer in memory, into which we copy the head of all chunks (with N chunks, we copy 1GB/N from each chunk in the buffer). We then sort the buffer using a parallel radix sort and partially dump it to disk. After the sort, part of the buffer may not be dumped to disk because its content is higher than the minimum value not yet copied from the chunks. We then repeat the process until all the data in the sorted chunks has been copied into the buffer and flushed to disk.

Evaluation Table 5.3 summarises the performance of the different approaches on an rmat30 graph with 8GB of DRAM. Our best optimised approach is 1.6× faster than existing approaches. Figure 5.2 present the evolution of the optimised approaches with 8GB of DRAM, varying the graph size. When the graph fits in memory our approaches are on par with existing approaches (only the optimised count sort is slower than the existing count sort because it reads and writes twice more data). All optimised sorts perform better than the approaches presented in previous work when the graph does not fit in memory. The best approach to sort large graphs is the optimised merge sort inspired by disk techniques.

The optimised merge sort is still CPU-bound. Figure 5.3 presents the bandwidth over time of the unoptimised and optimised merge sort. We can see that replacing the parallel quick sort used by STXXL by a parallel radix sort reduces the overall sorting time (first phase of the

graphs). The disk is however still idle while data is sorted in memory (only the first iteration of the radix sort is overlapped with loading data from disk). Indeed, on our machine it takes 0.4s to load 1GB in DRAM, but the fastest sort we are aware of takes 3s to sort data in memory. The disk is thus idle during most of the sort. On the other hand, the parallel merge phase maximises disk bandwidth.

Summary Three main observations can be made from these results:

- Despite their speed and high random IO bandwidth, NVMes cannot be used directly as memory. In memory algorithms such as the radix sort used in [87] perform poorly. However, simple modifications (removing cache optimisations) made the algorithm perform well on the NVMe.
- Disk approaches are not efficient on NVMes: they are CPU-bound. Parallel optimisations to these algorithms are key to have good performance.
- Despite parallel optimisations, the disk approaches are still CPU-bound. As a consequence, an increase in device speed would benefit more to the radix sort inspired by in-memory algorithms.

5.4.2 Grid

Grid using many files To create a grid, the simplest approach consists in opening 1 file per cell and copying the edges in the right cells. This is the approach used by Gridgraph. More precisely, threads read the input file and place edges into small in-memory buffers (one per cell). Once the buffers are full, they are flushed to the corresponding cell file. This method should be optimal to create a grid: edges are only written once to disk. In practice we found this method to be IOPS bound: even with a small number of cells, buffers in memory must be small and are thus frequently flushed to disk. This results in frequent writes to different files on disk, a scenario similar to randomly writing small amount of data on disk. It takes 300s seconds to create a grid with 512x512 cells on rmat29 (64GB) instead of the theoretical 61s (one read + one write of the graph).

Grid using a sort Another option to create the grid is to sort the edge array so that edges appear in cell order in a single file. Just as in the adjacency list case, existing sorts are IOPS bound or CPU bound on fast devices. Here we report numbers using the parallel merge sort described previously. This approach performs twice the amount of IO as the "one file per cell" approach, but all reads and writes are sequential. Figure 5.4 shows the grid creation time using "one file per cell" and the parallel merge sort. When the number of cells is larger than 64, even though sorting does more IO, it outperforms the "one file per cell" approach.

Summary Just as for the adjacency list, the best way to create the grid is the approach that uses the storage device efficiently. The choice depends on the number of cells: when the number of cells is low the Gridgraph approach works best because writes can be efficiently

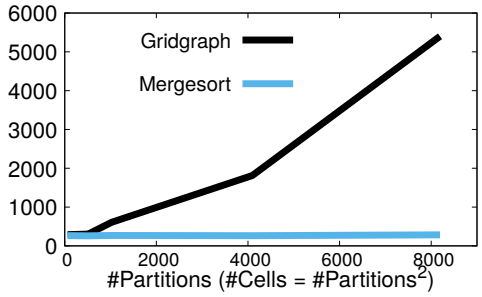


Figure 5.4 – The time to create a grid from an rmat29 (64GB) graph depending on the number of cells with different approaches on an NVMe.

buffered before being written to disk. Surprisingly, as the number of cells increases, sorting is faster even though it writes twice as much data to disk.

5.5 Evaluation

In this section we evaluate the performance of Optimus for different algorithms and graphs, on the two presented data layouts.

We first present the algorithms and datasets used. Then we evaluate the performance of Optimus on both layouts. We analyse the memory footprint of each layout. Finally, in section 5.5.3 we compare against the accelerated version of Mosaic.

All the experiments were done on the setup described in Section 5.1, on the NVMe. We vary the amount of DRAM given to applications. For each combination of algorithm, dataset, and graph layout, we allow applications to store the vertex state and metadata in DRAM with 2GB of page cache for buffering.

Algorithms We selected 4 algorithms with different characteristics in terms of functionality, number of vertices active during computation, type of graph required, and computation complexity. **Breadth-first search (BFS)**, **Single Source Shortest Path (SSSP)**, **Weakly Connected Components (WCC)**, and **Pagerank (PR)**.
Datasets. For simplicity, in this chapter we focus on the pre-processing and compute costs of RMAT graphs [33], synthetic graphs similar to social network graphs. Section 5.5.1 extends the findings to graphs with higher diameters or different vertex to edge ratios.

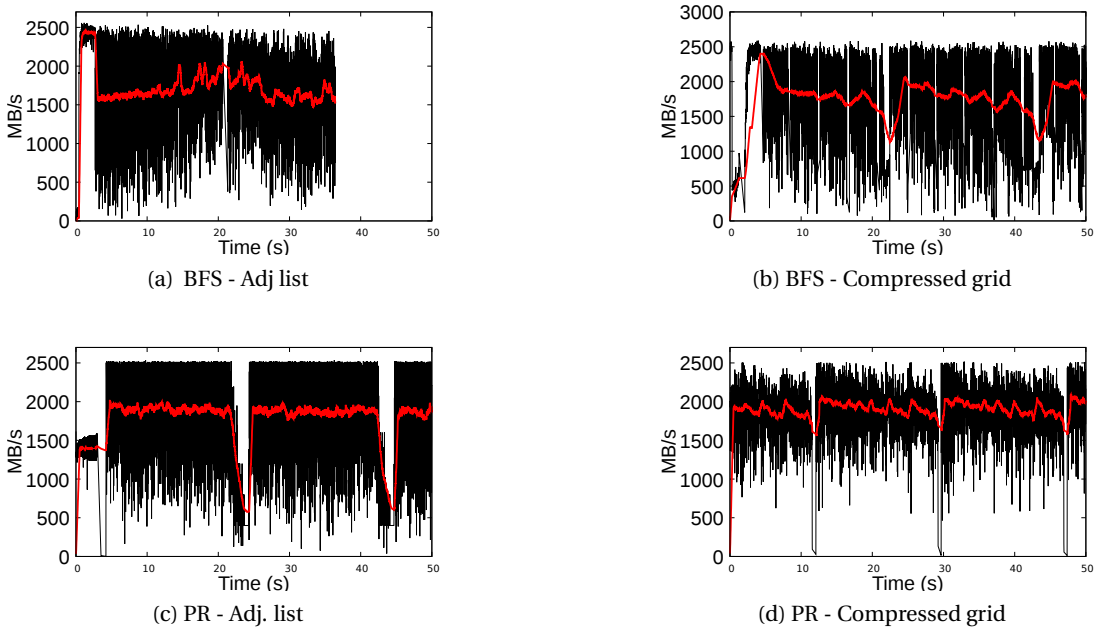
Graph	Edges	Size (GB)	sorted
rmatN	2^{N+4}	8 to 768GB	n
sk-2005	1.9B	14.5	n
twitter	1.5B	11	y
yahoo-webgraph	6.6B	50	y

Table 5.4 – Datasets used in the chapter. N is 26 until 32 for RMAT graphs.

Table 5.4 gives an overview of the chosen datasets and their characteristics. All algorithms except SSSP take an unweighted graph as input. The biggest graph we can accommodate on our NVMe is RMAT-32, whose weighted, unprocessed size is 768GB.

Layout	Transf.	BFS	PR	WCC	SSSP
Adj.list	196*	**37**	223	96	**165**
Comp.Grid	190	77	**189**	**56**	228

Table 5.5 – Execution time for RMAT-29 for various algorithm using the two data layouts. The shorter running time is highlighted.The second column provides the time to create the data layout from an edge array. ∗ For WCC, this time should be doubled for directed graphs.



(a) BFS - Adj list

(b) BFS - Compressed grid

(c) PR - Adj. list

(d) PR - Compressed grid

Figure 5.5 – The bandwidth sustained when using different data layouts, for BFS and PR. The experiments are run on RMAT-29. In red, the moving average over 50s of execution.

Table 5.5 shows the running time of all the algorithms when using both, the adjacency list and compressed grid layout. For BFS and SSSP the adjacency list is more suitable than the grid. For WCC, We see a gain of 1.7× when the graph is transformed into a compressed grid. The algorithm requires an undirected graph. In case the graph is directed, we need to create the adjacency list representation of incoming edges as well. Since the grid stores both endpoints of an edge, this is not needed, leading to adjacency lists doing two times more I/O.

Figure 5.5 shows the bandwidth sustained by the different data layouts for BFS and Pagerank. Both the adjacency list and the compressed grid have spikes during which the NVMe is idle, particularly visible at the end of every iteration.

Overall the compressed grid sustains a higher bandwidth. Especially for Pagerank, the NVMe is used more efficiently, with less idle time and fewer spikes.

5.5.1 Other algorithms and graphs

Table 5.6 presents running times for all the datasets and algorithms evaluated. The table shows that the benefit of a data layout is impacted not only by choice of algorithm, but by the shape of the graph. For BFS and SSSP, high diameter graphs like the Yahoo-webgraph, require many

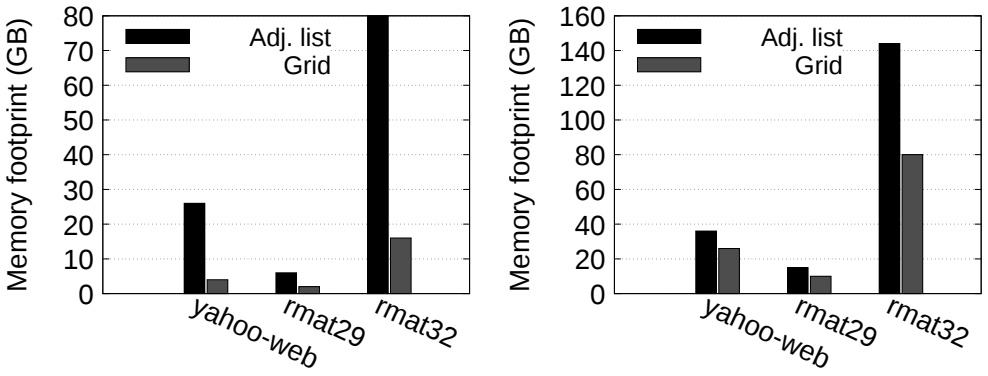
Algo	Layout	RMAT-32	Yahoo-Webgraph	SK-2005	Twitter
BFS	Adjacency list	588	45	71	15
	Compressed grid	844	359	97	19
PR	Adjacency list	3016	190	160	26
	Compressed grid	1603	140	46	22
SSSP	Adjacency list	1700	220	278	44
	Compressed grid	604	450	483	70
WCC	Adjacency list	2053	1461	95	37
	Compressed grid	1209	834	61	24

Table 5.6 – Algorithm execution times for all datasets.

iterations to complete(more than 4000). Each iteration has very few vertices active, resulting in a much bigger gap between the performance of adjacency lists and compressed grids.

For very big graphs, such as the RMAT-32 graph, the transformation time between data layouts is 1.5h. The improvement when using adjacency lists for BFS and SSSP, is smaller compared to the benefits of using the grid for PR and WCC. Therefore, we find it more beneficial to pay the transformation cost once and run everything on the compressed grid. An additional argument for this is the lower memory footprint of the data layout. We discuss the later in the following section.

5.5.2 The DRAM cost of out-of-core systems



(a) BFS (b) PR

Figure 5.6 – Memory footprint of BFS and PR for different graphs and data layouts.

These data structures differ in the amount of DRAM they need to store the metadata, for example the active vertex set in adjacency lists. Figure 5.6 shows the minimum DRAM required for different graphs depending on the data representation. Just as the choice of transformation method, the choice of data structure depends on the ratio of DRAM vs input size. Thus, while a data layout performs better, it might not be possible to use it if the amount of DRAM it requires is unavailable.

For all the data layouts, we include the vertex state in the amount of DRAM needed to execute.

Having the vertex state in DRAM is a requirement of many recent semi-external systems [57, 144]. We discuss below any additional metadata each of the data layouts stores in DRAM. The vertex state is, per vertex, 4B for BFS and 20B for Pagerank.

Adjacency lists store two work-queues whose size is a function of the number of vertices. The DRAM cost of the compressed grid are the work items. As the graphs grow, the size of the metadata grows significantly faster for adjacency lists than the grid. This is not surprising given that the cost for the grid is a function of the number of threads, orders of magnitude lower than the number of vertices.

The DRAM needed to create the layouts using the optimised merge-sort, for all the graphs is a function of the number of vertices in the graph for adjacency lists, and the number of cels for grids. The maximum amount of DRAM needed to create an adjacency list from RMAT-32 is 53GB, while the grid requires 32GB to store the offsets. They can also be memory mapped when the amount of DRAM is not satisfied.

Reducing the DRAM footprint. In practice, we see that even for very large graphs, such as RMAT-32, we can process the graph with 20GB of DRAM. The goal of this section to show how Optimus can support external mode if needed.

VPart [50] is a recent system that argues for storing the vertex state on NVMes as well. The work builds on Grafboost [67] designed to leverage FPGSs for graph computation and storing the vertex state out-of-core.

VPart represents the graph as an adjacency list, but partitions the graph in such a way that the vertex state of one partition fits in the available DRAM. Edges are carefully placed into partitions as to balance out the work per partition. When the source vertex of an edge does not belong to the same partition as the destination, the vertex state of the source is replicated and updated between iterations.

VPart is evaluated on RMAT-32 and a large real-world dataset. Both graphs have similar input size, while the RMAT graph has more vertices, thus needing more DRAM to store the state. Since the code is not open source, we could not run the system directly. However, we ran GrafBoost on our setup and note that the performance is lower than what is reported in the VPart paper. This is most likely due to the fact the VPart evaluates its system and Grafboost on two NVMes, running at much higher bandwidth. The applications evaluated were BFS and Pagerank. When using a compressed grid, Optimus processes RMAT-32 on both algorithms within the memory limit in their paper while keeping the state in DRAM. On their setup, the paper reports 334s per Pagerank iteration and 803 seconds for BFS on RMAT-32. As seen in Table 5.6, we complete 10 iterations of Pagerank on the compressed grid in 1600s with an average per-iteration time of 160s. BFS runs for 588s using an adjacency list, and 844s when using a compressed grid, on one NVMe.

We add support for external computation by memory mapping the state, and accessing

only $P\_RAM = \frac{STATE\_IN\_RAM}{P}$ cells at a time, where $STATE\_IN\_RAM$ is the number of vertices whose state can fit in the given amount of DRAM.

Before processing a batch of P_RAM cells, Optimus loads the state of the vertices belonging to the row currently being processed in a separate array. The memory mapped the state of the vertices who correspond to the range of columns in P_RAM is memory locked (using mlock) to avoid it being evicted by the edges. This leads to a slight load imbalance due to many cells in P_RAM being small or empty. We leave further optimisations on this for future work.

With the current design, running one iteration of Pagerank using 40GB takes 560s, 1.7× slower than the time reported by VPart (running on two NVMes and with 128GB of DRAM).

Conclusion While we see room for improvement in the external mode in Optimus, even the numbers reported by VPart show that there is a performance penalty when storing the state out-of-core. We thus encourage reducing the memory footprint by transforming the graph representation into a grid, rather than storing the state out-of-core.

5.5.3 Comparison against specialised hardware

Mosaic was designed to leverage Xeon Phis to extract the full bandwidth of one or more NVMe devices. In Table 5.7 we compare the numbers they report for the Twitter graph, using specialised hardware, with Optimus. Since the time to create the tiles using specialised hardware is not precisely reported in the paper, we only compare computation times. We show that, with a simpler data layout, 1 NVMe, and no specialised hardware we achieve comparable results.

Algo	**Mosaic 16 NVMe**	**Mosaic 1 NVMe**	**Optimus**
BFS	11	52	15
PR	1.8	5	2.5

Table 5.7 – Compute time of BFS and one iteration of Pagerank over Twitter for Mosaic and our code. We run BFS with adjacency lists and Pagerank over the compressed grid. (NVMe)

While the specialised hardware allows for more efficient background I/O, our results suggest that the benefits from improving transformation time, and using a data layout more suitable for a particular algorithm is more promising.

5.6 Summary

In this chapter we presented different ways to transform(pre-process) and compute on graphs stored on NVMes. Surprisingly we found that existing out-of-core algorithms are CPU bound on NVMes and that algorithms inspired by in-memory techniques are IOPS bound. We designed a system inspired by out-of-core and in-memory techniques and show that, given the right choice of data layout, it outperforms existing graph engines when processing graphs on NVMes.

6 Exploiting byte addressable NVMs in Large-scale Graph Analytics

This chapter explores how replacing DRAM with emerging non-volatile memories (NVM)such as PCM or RRAM impacts state-of the art graph processing frameworks. We study whether NVM can eventually become a cheaper and more scalable alternative to DRAM reducing the degree of scale-out and the cost of graph analytics. Table 6.1 shows how these NVM technologies relate to DRAM. Increased capacity and lower cost come with higher latencies and lower bandwidth as well as limited write endurance. These characteristics can potentially degrade the performance of systems designed assuming DRAM as the underlying technology.

However, if carefully designed to mitigate the limitations of NVM, graph processing systems could leverage the underlying memory hierarchy in order to scale at a lower cost. The first step towards this goal is understanding how existing state of the art frameworks operate with NVM. To that extent, we quantify the performance of four state-of-the-art frameworks on NVM using a hardware emulator described in Section 6.2.

Graph analytics algorithms and frameworks differ vastly in terms of access patterns, data structures and programming models. We chose a representative subset based on the work in [115]. Section 6.4 shows the impact that different bandwidth and latency points of NVM have on Galois[98], Graphlab[84], Graphmat[121] and X-Stream[114] running Pagerank, Bread-First Search (BFS), Triangle Counting and Collaborative Filtering. We did not find a publicly available implementation of Collaborative Filtering for Galois, hence we do not evaluate Galois for this algorithm.

NVM causes a degradation in performance for all test cases but the magnitude of degradation varies between algorithms and frameworks, ranging from 1.5× to 4×. In order to understand these differences, we perform a detailed characterisation of the frameworks using hardware performance counters. The analysis shows that, due to CPU memory-level parallelism and hardware prefetchers, the performance degradation is not necessarily proportional to the reduced bandwidth or the increased latency of NVM, but it is still substantial compared to the DRAM-only performance.

Parameter	3D-DRAM	DDR-DRAM	NVM
Capacity per CPU	10s of GBs	100s of GBs	Terabytes
Read Latency	$\frac{1}{2}\times$ to $1\times$	$1\times$	$2\times$ to $4\times$
Write bandwidth	$4\times$	$1\times$	$\frac{1}{8}\times$ to $\frac{1}{4}\times$
Estimated cost	-	$5\times$	$1\times$
Endurance	10^{16}	10^{16}	10^6 to 10^8

Table 6.1 – Comparison of memory technologies [7, 11, 83, 112]. NVM technologies include PCM and RRAM [7, 112]. Cost is derived from the estimates for PCM based SSDs in [74].

As an attempt to bridge the gap between the performance on DRAM and NVM, we modify Graphmat to explore the opportunities for fine-grained data tiering in *hybrid* memory systems with DRAM and NVM. We show that by placing only a fraction of data in DRAM (6.7% to 31.5% of the total memory footprint), GraphMat achieves 2.1×-4× better performance than the corresponding NVM-only implementations, and within 1.02×-1.2× of the DRAM-only performance. This chapter makes the following contributions:

- Characterisation of a hardware emulator that accurately models various bandwidth and latency points expected for emerging NVM technologies.
- Detailed analysis of the impact of NVM on different graph analytic frameworks and algorithms, quantifying the overheads of NVM-only solutions (1.5× to 4×) compared to their DRAM-only counterparts.
- A study of the benefits of application-driven tiering with Graphmat, demonstrating that Graphmat can achieve close to DRAM-only performance (within 1.2×) by utilising only a fraction of DRAM (as little as 6.7%) in a *hybrid* memory system with DRAM and NVM.

6.1 Background

As previously stated, graph algorithms suffer from irregular access patterns that may limit their performance even on DRAM. We provide a short discussion on the impact of memory access patterns on the average memory access latency, and therefore performance of an application.

Depending on the actual implementation and the memory access pattern, mitigating factors to the high memory latency are modern processors' (e.g., $IntelXeon$) extensive use of out-of-order execution and aggressive hardware prefetching [12]. These features can successfully reduce the average latency of memory reads, for certain access patterns, by reducing the number of cache misses and increasing memory-level parallelism (MLP) [39].

These effects are demonstrated in Figure 6.1, which shows the average latency of memory reads (for various access patterns) on an Intel Xeon E5-4620 system. In these experiments, one thread reads memory (in the specified pattern) and measures average latency while other threads consume memory bandwidth by accessing their private memory. For *dependent* accesses, the memory of the thread measuring latency is initialised for pointer chasing. The locations for *independent* accesses are generated on the fly without any dependencies. Dependent accesses

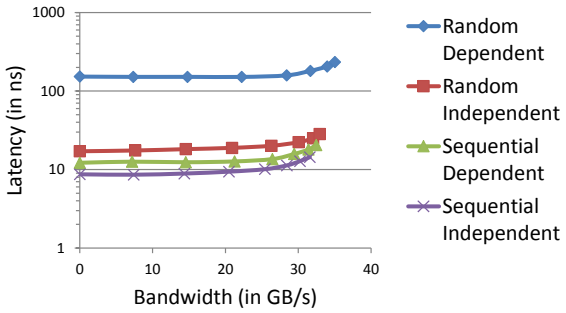


Figure 6.1 – Memory read latency for various access patterns.

can have only one memory load waiting for execution, and therefore does not benefit from out-of-order execution. In the case of independent accesses there can be many in-flight loads.

Random dependent represents the worst case scenario, with every load experiencing the entire memory latency. In comparison, *random independent* is an order of magnitude faster due only to MLP. Similarly, *sequential dependent* is an order of magnitude faster than random dependent, but entirely due to hardware prefetchers. *Sequential independent*, which benefits from both MLP and prefetchers, shows the best performance of all.

The key observation from this experiment is that the performance of memory-intensive applications depends heavily on the pattern of their memory accesses.

6.2 *Hybrid* Memory Emulator

The *hybrid memory emulation platform (HMEP)* enables the study of hybrid memory with real-world applications by implementing – (i) separate physical memory ranges for DRAM and emulated NVM, and (ii) fine-grained emulation of their relative latency and bandwidth characteristics. HMEP has been used in other research [21, 44, 47, 100, 101, 141], but it has not been described in the detail needed to explain the experimental results shown in this chapter.

HMEP is based on a dual-socket Intel Xeon E5-4620 platform, with each processor containing eight 2.6 GHz cores. Hyperthreading is disabled. Each CPU supports four DDR3 channels and memory is interleaved across the CPUs.

Separate DRAM and NVM physical ranges: Using custom BIOS firmware, HMEP partitions the four memory channels of a CPU equally between DRAM and emulated NVM. The NVM region is available to software either as a separate NUMA node (managed by the OS) or as a reserved memory region (managed by PMFS) [47]. The total amount of memory in the system is 320 GB of which 256 GB is reserved for emulated NVM.

Read latency emulation: HMEP emulates read latency on the NVM physical range using special CPU microcode, which uses debug hooks in the CPU to implement a performance model for latency emulation. The model monitors a set of hardware counters over very small intervals, and for each interval estimates (and applies) the additional cycles that the core would have stalled if the underlying memory was slower than DRAM. A naive method of

calculating *stall cycles* would be to count the number of actual memory accesses (i.e., last level cache misses) to NVM and multiply it by the desired extra latency. This method, however, is suited only for simple in-order processors and highly inaccurate for modern out-of-order CPUs (§6.1).

We implement a model based on the observation that the number of cycles that the core stalls waiting for the memory reads to complete is proportional to the actual memory latency. If Lp is the target latency to emulated NVM, then the additional (proportional) stalls that the model applies for the time interval is:

$\delta_{stall} = S \times \frac{L_p - L_d}{L_d}$, where S is the actual number of stall cycles due to accesses to the emulated NVM range, L_p is the desired NVM latency, and L_d is the actual latency to DRAM.

In calculating S, we are limited to the following available counters on our test processor:

- Core execution stall cycles due to second level cache (L2) misses (S_{L2}).
- Number of hits in LLC (H_{LLC}).
- Number of last level cache (LLC) misses to DRAM (M_{dram}) and NVM (M_{nvm}) ranges.

Using these counters, the model first computes the execution stalls due to LLC misses (S_{LLC}) as follows:

$S_{LLC} = S_{L2} - (H_{LLC} \times K)$, where K is the difference in latency of a LLC hit and a L2 hit.

Finally, the model computes S as:

$$S = S_{LLC} \times \frac{M_{nvm}}{M_{dram} + M_{nvm}}.$$

Validation: To validate the model, we emulate the latency of slower NUMA memories in multi-processor platforms and compare the performance of several application on emulated NVM vs. actual NUMA memory. Following this approach, we validated the latency emulation model for a large number of applications – including several microbenchmarks (e.g., various sort algorithms), benchmarks from SPEC CPU2006 and workloads in this chapter. Performance with NVM (emulating remote memory latency) is always within 7% of the performance with actual remote memory.

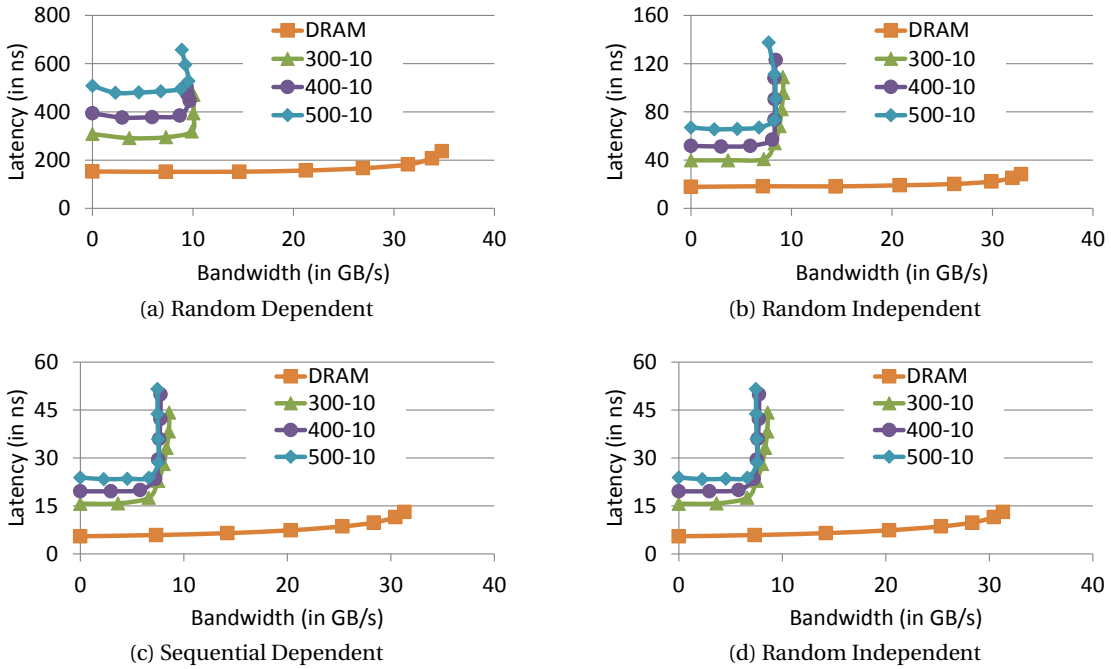
Limitations: NVM device characteristics are very different from that of DRAM. For instance, reads and writes to a PCM device have to wait for the preceding writes to the same memory line to complete [110]. The HMEP latency emulation model emulates only the average latencies and not NVM's device-specific characteristics. This restriction is primarily due to the limited internal CPU resources available for NVM latency emulation.

CPU hardware prefetchers can drastically improve the performance of sequential and strided memory accesses. HMEP assumes that the prefetchers will continue to be at least as effective

with NVM as they are today with slow remote memory (of comparable latency) on large NUMA platforms. This assumption is reasonable even if we ignore the fact that, if needed, CPU prefetchers could be assisted by some form of prefetching on the NVM modules as well.

Bandwidth emulation: NVM has lower sustained bandwidth than DRAM, particularly for writes (Table 6.1), though that could be improved using ample scale-out of NVM devices and buffers. [1] HMEP emulates read and write bandwidths by programming the memory controller to limit the maximum number of DDR transactions per μsec. This throttling feature can be programmed on a per-DIMM basis [8], and is applied only to the NVM range.

Limitations: The bandwidth throttling feature in the memory controller is a single knob that limits the rate of all DDR transactions. Therefore, HMEP is unable to vary the read and write bandwidths independently.



(a) Random Dependent

(b) Random Independent

(c) Sequential Dependent

(d) Random Independent

Figure 6.2 – Read latency-bandwidth plots for several HMEP configurations and all access patterns

Characterisation: Figure 6.2 shows latency and bandwidth characteristics for memory reads to the NVM range in various HMEP configurations. These configurations are denoted by *x*-*y*, where *x* is the emulated NVM read latency (in ns) and *y* represents the peak bandwidth (in GB/s) to the NVM range. Access patterns are as described earlier (§6.1). Read latency to NVM depends heavily on the access pattern (as with DRAM) – sequential and independent reads are much faster than random and dependent reads. Figure 6.3 shows the measured sustained bandwidth to NVM for various HMEP configurations and access patterns. As

[1] Since writes to write-back caches are posted, NVM's slower writes result in lower bandwidth and not higher latency on every write.

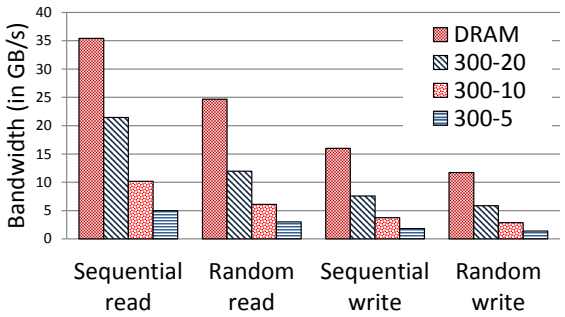


Figure 6.3 – Bandwidth of HMEP configurations

Algorithm	Graph type	Vertex property	Edge access pattern	Message size (B/Edge)	Vertices active	Input size (GB)- binary	Input size (GB) - text
Pagerank	Directed	Double	Sequential	8	All iter.	12	18
BFS	Undirected	Int	Random	4	Some iter.	24	40
Collab. filtering	Bipartite	Double[]	Sequential	8K	All iter.	24	34
Triangle counting	Directed	Long	Sequential	$0\text{-}10^{-6}$	Non-iterative	12	18

Table 6.2 – An overview of the main algorithm characteristics

expected, sequential accesses achieve higher bandwidth than random accesses, and read bandwidth is higher than write bandwidth.

To summarise, despite the stated limitations, HMEP adequately emulates the relative characteristics of DRAM and NVM in a hybrid memory system, and also the performance behaviour of various memory access patterns.

6.3 Algorithm Characteristics

Achieving sequential and independent access in graph analytics is not always possible due to the irregular structure of the graphs and additional dependency on the programming model of the framework itself. To evaluate how NVM would impact graph analytics, we chose four algorithms representative of the different analytics disciplines as discussed in [115]: Pagerank, BFS, Triangle counting and Collaborative filtering.
Pagerank is a communication intensive algorithm with updates propagated along all edges in each iteration. **Bread-first search (BFS)** is a traversal algorithm where in each iteration only the nodes adjacent to a newly discovered node are processed, thereby reducing intra-node communication. **Triangle Counting** requires each node to count the intersections among the neighbours of its immediate neighbours. Depending on the actual graph structure, the size of messages exchanged between the nodes in this algorithm could be very large. Depending on the framework, the bipartite graph for **Collaborative Filtering** is represented as a matrix or as a graph.

Table 6.2 provides an overview of algorithm characteristics along with the access pattern of each algorithm.

Framework	Programming Model	Execution Scheduling
Graphmat	Vertex-program + SpMV backend	Synchronous
Graphlab	Vertex-program	Async/Sync
Galois	Task-based	Async/Sync
X-Stream	Edge-centric Vertex program	Bulk-synchronous

Table 6.3 – Graph processing frameworks - characteristics

Algorithm Implementation The algorithms are commonly expressed as “vertex programs" where vertex state is propagated as a message along outgoing edges and updated based on messages along incoming edges. Table 6.3 shows the programming models of the frameworks evaluated.

Graphmat expresses computation as a vertex program but the operations are internally converted to sparse-vector matrix computations [121], resulting in a better compute time while maintaining a simple intuitive API.

Galois supports a slightly different computation model where each message activates a node which is thereafter put in a task-list. All items within the task-list are computed on in parallel by respective custom scheduling policies that account for locality and priorities [98].

Graphlab [84] follows the above described vertex-centric model whereas *X-Stream* [114] slightly modifies this model to optimise for access to secondary storage such that, instead of iterating through the vertex set, the program iterates through the edge-list sequentially.

6.4 Evaluation

6.4.1 Methodology

The frameworks are provided with synthetic graphs generated using the Graph500 RMAT generator[6]. The generator provides graphs that correspond to the structure of real-world graphs of interest and is widely used by the graph analytics community for system evaluation [84, 114, 115, 121].

For Pagerank and BFS, the input is a scale 26 graph with 2^{26} nodes and 2^{30} connections. Since BFS requires an undirected graph, we add reverse edges to the dataset. And, since the vertex state and intermediary data is larger for Triangle Counting, we use a smaller (scale 24) graph with 16M nodes with 268M edges. Finally, Collaborative Filtering requires a bipartite graph for which we generated a graph according to [115] with 8M nodes and an average of 256 connections per node. The input for all frameworks except for Graphlab is in binary format. The total size of the input depending on the format is shown in the last two columns of Table 2.

As a starting point for our analysis we ran the different algorithm implementations on DRAM as our baseline case. Using the emulator described in Section 6.2, we analyse the behaviour

	Pagerank	BFS	Triangle Counting	Collaborative Filtering
Graphmat	4.40	9.96	31.95	320.04
Graphlab	13.83	87.50	44.95	563.41
Galois	6.89	8.66	24.08	-
X-Stream	7.60	29.62	1058.00	79.68

Table 6.4 – Absolute runtimes in seconds. The differences between frameworks are explained in 6.4.2

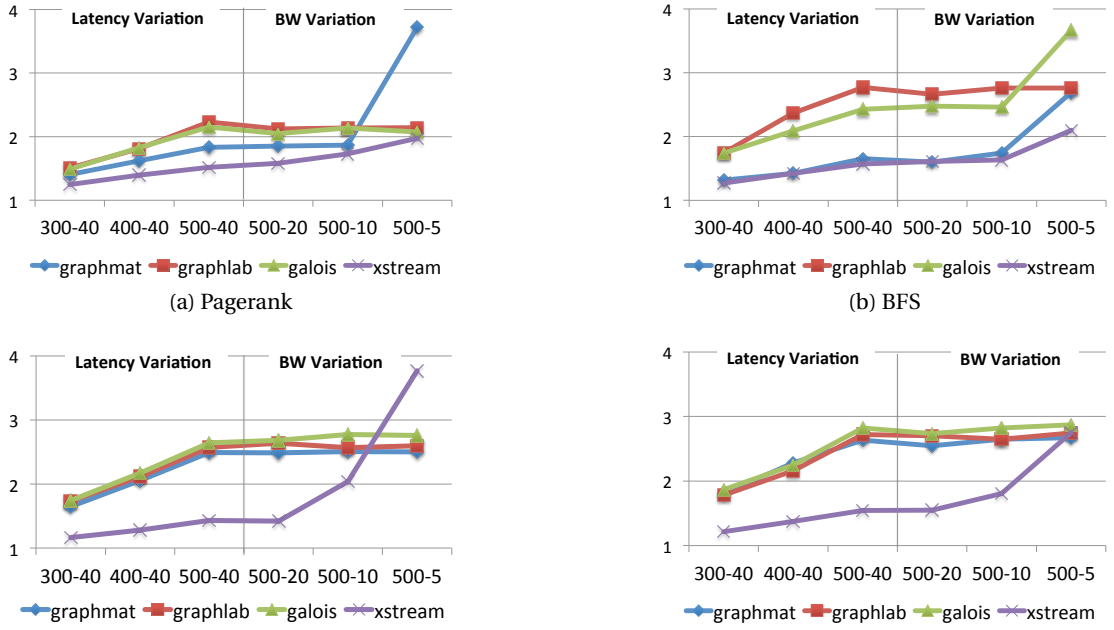
of these implementations with increasing memory access latency and decreasing memory bandwidth. The latency is varied from 300ns to 500ns and the bandwidth is varied from 40GB/s (equal to DRAM) to 5GB/s. The DRAM latency on the system is 150ns. Since the absolute runtimes differ among frameworks due to different implementations, we plot the ratio between the runtime of the framework at a particular latency/bandwidth point compared to the corresponding runtime on DRAM.

6.4.2 Analysis of performance in DRAM

The performance of the implementations depends on the programming model and data-structures of a particular framework, resulting in widely different runtimes as shown in Table 6.4.

The times reported are per iteration times for Pagerank and Collaborative filtering, whereas for BFS and Triangle counting we present the entire runtime, excluding the time taken to load the graph or for other setup.

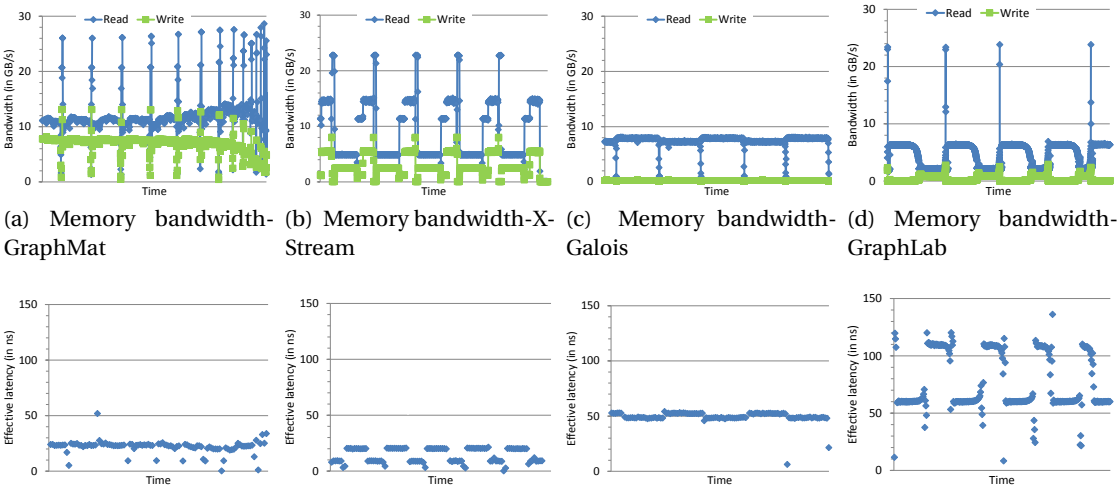
Graphmat and Galois have similar performance but for different reasons. The SpMV backend of Graphmat allows for quick computations and better expressibility of the data, especially for Collaborative filtering where the average degree of a vertex is 256 and SpMV operations lead to a 2× improvement over Graphlab. Unlike Graphmat and Graphlab which calculate the ratings for Collaborative Filtering using Stochastic Gradient Descent(SGD), X-Stream uses an optimised version of ALS[146]. The algorithm applies updates as they are generated computing the ratings faster than implementations of SGD.
Galois on the other hand supports asynchronous computation and its task-based programming model leads to quicker convergence. Asynchronous computation does not play a big role in algorithms such as Pagerank where we propagate updates along every edge in each iteration, but for traversal algorithms, where we pass an edge only once, it can significantly improve the time to converge. We can clearly observe this behaviour with X-Stream where all edges are streamed in every iteration, which for BFS leads to a large number of unnecessary reads.
Due to the need to support streaming, the implementation of Triangle counting in X-Stream is an approximate implementation [25], executed in a predefined number of iterations (100 in our case) which causes the significant difference in runtimes for this algorithm.
We note that Graphlab was designed as a distributed system and, therefore, some of their optimisations for distributed computation may have caused increased runtimes on a single node.



(a) Pagerank (b) BFS

(c) Triangle Counting (d) Collaborative filtering

Figure 6.4 – Performance variation on NVM. The X-axis shows HMEP configurations as **NVM latency(ns)-Bandwith(GB/s)**. The Y-axis shows the run time in NVM normalised to the run time in DRAM for a particular framework.



(a) Memory bandwidth-GraphMat (b) Memory bandwidth-X-Stream (c) Memory bandwidth-Galois (d) Memory bandwidth-GraphLab

(e) Effective latency-GraphMat (f) Effective latency-X-Stream (g) Effective latency-Galois (h) Effective latency-GraphLab

Figure 6.5 – Bandwidth (in GB/s) and Effective memory latency (in ns) for Pagerank. The X axis represents time.

6.4.3 Analysis of performance in NVM

Figure 6.4 shows results for several NVM latency and bandwidth points. First we vary the latency of NVM from 300ns to 500ns, while the bandwidth is fixed at 40GB/s (same as DRAM). Then we vary the bandwidth from 40GB/s to 5GB/s, to highlight the impact of both increased

latency and decreased bandwidth in a concise manner. While the performance degrades for all the frameworks, they are not equally sensitive to latency and bandwidth. Graphlab and Galois exhibit higher sensitivity to increased latency rather, while Graphmat and X-Stream are more sensitive to lower bandwidth.

To understand these results, we profiled the applications using hardware performance counters. Figure 6.5 shows the following key metrics from counter analysis of the Pagerank algorithm: (i) Read and write bandwidth, and (ii) Effective latency.

Effective latency approximates the average memory read latency in an application by measuring the core stalls due to pending reads per LLC miss. This metric, shown in nanoseconds, measures the effectiveness of MLP and hardware cache prefetchers. Higher effective latency means the workload is more sensitive to the higher latency of NVM. Similarly, applications with high bandwidth requirements are likely to perform worse with NVM.

Results in Figure 6.5 (shown in a timeline) correspond to the actual execution phase, excluding the loading and initialisation phases. GraphMat and X-Stream achieve significantly higher bandwidth than Galois and GraphLab, which explains the sensitivity of these frameworks to lower bandwidth. The observed effective latency for different frameworks confirms that the performance degradation at higher latencies is due to the stalls resulting from memory accesses in the framework. We attribute this result to the inability of certain frameworks (particularly Graphlab) to exploit the hardware prefetches and MLP. In fact, running these same experiments with prefetching disabled results in a performance drop of 12-25% for Graphmat and X-Stream, but has negligible impact on the performance of Galois and Graphlab.

X-Stream's sequential access pattern and GraphMat's efficient matrix representation of the data incur fewer random accesses than the indexing methods of Galois and Graphlab. The difference between Galois and Graphlab can be explained by the fact that Galois achieves better locality by placing the data as close to the execution threads as possible and then using a custom scheduler that efficiently schedules the active vertices for the next iteration.

The reduced bandwidth becomes a limiting factor for frameworks such as Graphmat and X-Stream, especially for communication intensive algorithms such as Pagerank where we observe a 30% drop performance when latency increases from 300ns to 500ns and a further 2× degradation when we reduce the bandwidth from 40 GB/s to 5 GB/s. In the case of Graphmat, even though the memory access latency is hidden well with prefetching, when the message size becomes large enough, such as for Triangle Counting and Collaborative Filtering, the framework becomes sensitive to higher latency instead of the lower bandwidth. The runtime increases by over 50% at 500ns latency (compared to 300ns) but does not change much as we reduce the bandwidth. The increased message size in Triangle Counting causes X-Stream to become more sensitive to reduced bandwidth than in the case of Pagerank and BFS where, compared to the performance at 40 GB/s, we see the performance drop by 1.5× at 20 GB/s and 3.7× at 5 GB/s.

6.5 Tiering

Our NVM-only analysis shows that, due to CPU's prefetch and MLP capabilities, the performance degradation of graph analytic applications with NVM is mitigated to an extent. However, depending on the implementation, the impact of NVM latencies and bandwidth can be further mitigated using only a modicum of DRAM in a hybrid memory system, and by intelligently tiering the data between DRAM and NVM.

In this setting, the system would place only the most performance-sensitive data(e.g., frequently accessed random data or critical write-only data) in high-performance DRAM and leverage the capacity (and cost) of NVM for all other data. Ideally, the data structures chosen for placement in DRAM would have to be small enough (compared to the overall graph size) for tiering to be effective from the cost perspective. To evaluate the potential of intelligent data

	Sparse Vectors	**Vertex data**	**Matrix**
Pagerank	1.53	1.06	18.84
BFS	1.02	1.56	35.71
Triangle Counting	0.63	2.64	7.10
Collaborative Filtering	3.89	1.28	31.38

Table 6.5 – Size in GB of Graphmat datastructures and the initial input size

placement in graph analytic platforms, we implemented a simplistic version of data tiering in Graphmat. The choice of Graphmat for this experiment is due to the fact that it is easy to identify the critical data structures in the Graphmat implementation. The SpMV backend of Graphmat defines three important data structures: sparse vectors (SV), vertex associated data (VxD) and (sparse) matrices (MTX) allocated to represent the data in memory. In this model, the 'messages' sent from one node to another are translated into sparse vectors. This vector is then applied to a vector containing the vertex data and the graph is represented as a matrix. The size of each of the data structures (per algorithm) is shown in Table 6.5. The size of SV ranges from 2.7% to 10% of the total memory footprint, while the vertex data ranges from 3.2% to 25.6%. Other data (e.g., for book-keeping) is negligible in size for all algorithms.

For the tiering experiments, we use HMEP in NUMA mode – i.e., software can access DRAM and NVM as separate memory nodes and use the NUMA API (e.g., libNUMA in Linux) to control the allocations from DRAM/NVM. We perform experiments where we allocate only SV or only SV+VxD in DRAM, while the rest of the application memory is allocated in NVM. Table 6.6 shows these tiering results with the two baselines – NVM-only and DRAM-only. We assume NVM latency of 500ns and bandwidth of 5GB/s in this case.

In these experiments, Graphmat's NVM-only performance is 2.5×-4× worse than its corresponding DRAM-only performance. By placing the sparse vectors alone in DRAM, Graphmat's performance improves to within 1.97× of DRAM-only for all algorithms other than Triangle Counting. The gains are particularly impressive for Pagerank and BFS (1.25× and 1.32×, respectively). Placing vertex data vectors (along sparse vectors) results in even better performance — 1.03×-1.2× of DRAM-only — but at a higher cost in terms of the amount of DRAM required

	PR	BFS	Triangle Counting	Collab. Filtering
NVM-only	17.58	26.78	79.87	854.51
SV in DRAM	5.49	13.62	72.71	628.31
SV+VD in DRAM	4.94	12.54	34.74	328.85
DRAM-only	4.40	9.96	31.95	320.04

Table 6.6 – Static tiering of data between DRAM and NVM. The table shows runtimes in seconds for various tiering options.

(6.7% to 31.5% of the total memory footprint). For the previously stated reasons, vertex data vectors in Triangle Counting are very large in size (25.6% of the total size) and latency-sensitive, and therefore result in the worst case scenario w.r.t. the amount of DRAM needed relative to NVM (31.5%).

To summarise, our initial experiments with tiering suggest that it has the potential to enable graph analytic frameworks to achieve close to DRAM-only performance, while requiring that only a fraction of the application memory footprint be present in DRAM in a hybrid memory system. As the next step, we plan to build more generalised analytics systems based on this observation.

6.6 Summary

Emerging NVM technologies are likely to bridge the gap between DRAM and block devices in terms of capacity and cost but they come with increased latencies and reduced bandwidth [5]. Our study shows that, despite optimised software implementations, NVM-only performance of these frameworks is 1.5×-4× worse than that of their DRAM-only counterparts, due to either higher latency or lower bandwidth of NVM.

Our subsequent experiments with data tiering suggest that, with optimal data placement in a well-suited implementation (such as Graphmat), it is possible to achieve close-to-DRAM (1.02× to 1.2×) performance in hybrid memory system with only a fraction of the costly DRAM (6.7% to 31.5% in Graphmat's case).

We believe that this conclusion can be generalised to other big-data applications that employ indices or cache a small portion of the data in order to achieve good performance. In addition to analysing the impact of NVM on other applications, we are exploring system software to automate the classification and optimal placement of data in hybrid memory systems with any number of different physical memories.

Finally, though the density of NVM compared to DRAM enables processing more data on a single machine than previously possible, we do not expect it to eliminate the need for scaling out. More likely NVM will reduce the degree of scale-out, resulting in interesting implications to the complexity of the overall system, particularly of the networking subsystem. We plan to explore these aspects in future.

Dynamic graph processing Part II

7 Graph analytics

This chapter presents Snowy, a new incremental graph processing system for graph analytics algorithms. These algorithms compute on the entire graph, i.e. one update can propagate through all the edges. In these applications, Snowy in most cases achieves very low latency in reflecting the updates to the graph in the result of the computation, even at very high update rates. The key idea in Snowy is to allow updates to the graph to occur in parallel with continually executing the graph computation. A programmer can adapt his program for incremental execution in Snowy with relatively little effort. The basic model is vertex-centric [86]: the state of the computation is kept in the vertices, vertex programs run in parallel, and may put new vertices on a work queue. To support incremental execution, callbacks need to be added for what action to take for a particular graph modification. In many cases, it suffices to put some vertices on the work queue, but in some cases it may also require un-doing some of the state rendered invalid by the deletion of an edge.

We have implemented Snowy in 3700 lines of C code. We have used the implementation to experiment with 6 common graph algorithms, including traversal algorithms (BFS, SSSP, WCC) and sparse vector-matrix multiply style algorithms (ALS, Belief Propagation, Pagerank). For traversal algorithms, the latency of individual updates is mostly in the microsecond range, with occasional outliers up to a second. In any case, the latency is much lower than batching could achieve. This result holds for different algorithms, graphs and graph sizes. For sparse vector-matrix multiply algorithms, there is no easy way to measure the latency of an individual update, as the result of the algorithm typically depends on many updates. For these algorithms we show that results derived from the execution of the algorithm, such as the top-10 pages in Pagerank, converge very quickly to their final value. We also evaluate some of the implementation choices we made, including using locking for synchronisation and storing edges in unsorted bucket lists.

The contributions of this chapter are:

- The observation that for graph algorithms graph updates can proceed efficiently in parallel with graph computation.

- A study of the latency benefits of this approach, and the tradeoffs involved in terms of consistency of the results.
- An efficient platform for incremental graph computation that incorporates these ideas, and its experimental evaluation on a variety of graphs, graph sizes and algorithms.

The outline of the rest of this chapter is as follows. In Section 7.1 we present the design of Snowy and the API it provides to the programmer. In Section 7.2 we discuss the possible inconsistencies that can result for processing updates in parallel with computation, and measures to cope with those inconsistencies. In Section 7.3 we detail some of our implementation decisions. In Section 7.4 we present the graphs and algorithms used in the evaluation and evaluate the performance of Snowy along a number of dimensions. We present the conclusions from this chapter in Section 7.5.

7.1 Design

In this section we present the high-level design of Snowy, including the system and programming model, and describe the type of updates we expect. In the following section, we will delve deeper into interfacing graph algorithms with Snowy.

7.1.1 Programming model

The programming model is vertex-centric: the state of the computation is stored in the vertex value, and the programmer writes a vertex program. This model is flexible enough to express a broad class of graph algorithms [86]. A vertex program can access the vertex value, the incoming and outgoing edges of the vertex, and the vertex values of its neighbours. The vertex program can update the vertex values, but the updates must be commutative and associative, a common assumption in graph processing systems. Algorithm 1 provides an example of Single Source Shortest Path (SSSP) written in this fashion.

Algorithm 1: Vertex-centric implementation of SSSP.

```
1  sssp(vertex v) {
2      foreach_outgoing_edges(v, dst, edge) {
3          lock(dst);
4          if(dst.dist > v.dist + edge.weight) {
5              dst.dist = v.dist + edge.weight;
6              dst.father = v;
7              push_in_next_wq(dst);
8          }
9          unlock(dst);
10     }
11 }
```

The overall computation consists of a number of iterations, executed one after the other. An iteration starts with a number of vertices on a work queue and ends when the work queue is empty. Snowy takes a vertex from the work queue, and executes the vertex program on

that vertex. Vertex programs may execute in parallel, and the programmer is responsible for proper synchronisation between the parallel executions of the vertex programs. In addition to reading and writing vertices, a vertex program may insert new vertices on the work queue for the next iteration. The computation is initialised by setting the vertex values to some initial value, and putting one or more vertices on the work queue.

Vertex values are modified in place, so they can change in the middle of an iteration. If the vertex program requires that this does not happen, then it is possible to split the iteration into multiple phases that are serialised. Algorithm 2 presents the pseudo-code of a vertex-centric implementation of Pagerank. The computation is performed in two separate phases. During the first phase vertices pull Pagerank values from their neighbours; Pagerank values are not updated during this phase. During the second phase vertices update their Pagerank and push their neighbours in the work queue.

Algorithm 2: Vertex-centric implementation of Pagerank. At every iteration, the `Pagerank` vertex program is called twice on vertices present in the work queue. The two distinct phases ensure that all vertices in the work queue pull the same Pagerank value from a given vertex in phase 0.

```
1  Pagerank(vertex v, int phase) {
2      if(phase == 0) {
3          v.sum = 0.;
4          foreach_incoming_edges(v, dst)
5              v.sum += dst.rank/dst.out_degree;
6      } else if(phase == 1) {
7          double new_rank = 1 - α + α * v.sum;
8          if(diff(n.rank, new_rank) > 5%) {
9              v.rank = new_rank;
10             foreach_outgoing_edges(v, dst)
11                 push_in_next_wq(dst);
12 }   }   }
```

7.1.2 Graph updates

Snowy supports vertex addition and deletion, edge addition and deletion, and modification of edge weights, if applicable. We focus in this discussion on edge addition and deletion. Edge weight modifications are handled in a similar fashion to edge additions and deletions. Vertex additions (resp. deletions) are modelled as addition (resp. deletion) of associated edges.

A graph update consists of an update to the data structures representing the graph (called structural updates in the rest of this chapter) and incremental execution of the graph algorithm to reflect the updates in the result. The key distinguishing feature of Snowy is that these two – structural updates and incremental algorithm execution – proceed in parallel.

We distinguish between two types of vertex programs: *monotonically-converging* programs and *always-converging* programs. We define *always-converging* programs as programs that will converge towards a correct final state of the graph even if vertices are initialised with random values. Pagerank is an example of a such a program: as long as all vertices have been

initialised with strictly positive values, the relative rank of vertices will be correct at the end of the computation. We define *monotonically-converging* programs as programs in which vertex values can only evolve in one direction (increase or decrease but not both). SSSP is an example of such a program: the vertex program can only decrease the value (distance) of a vertex. *Monotonically-converging* programs assume that the graph is initialised with "correct" values (e.g., an infinite distance for SSSP). If this assumption does not hold, then the final state of the graph might be incorrect. For short we refer to *monotonically-converging* programs as *monotonic* programs in the rest of the chapter.

Adding support for incremental execution in these two types of programs mainly consists in finding vertices that might be affected by a structural change. These can be determined by inspection of the vertex program. In fact, with proper compiler support, this set of vertices could be determined automatically, but for now it is left to the programmer. We describe how edges additions, removals and modifications are handled by *always-converging* programs, and *monotonic* programs.

Edge addition: Edge additions are simple to handle both for *monotonic* and for *always-converging* programs. In both cases it suffices to find the list of vertices that (i) could push or pull values through that edge, or that (ii) depend on the number of edges of the source or destination vertex. Adding these vertices to the workqueue is sufficient to make sure that the effect of the edge addition are correctly propagated in the graph.

With these simple additions, the program converges to the correct solution on the original graph plus the added edges. The invocations of the vertex programs on the original graph, followed by the invocations of the vertex program resulting from the additions, constitute a correct sequence of invocations on the resulting final graph, and therefore the final result is correct.

To support edge additions, programs have to implement an `on_edge_addition` function. Algorithm 3 presents this function for the SSSP and Pagerank programs. In SSSP the source vertex of the edge is placed in the work queue. In the next iteration of the vertex program, the source will attempt to propagate its distance via the newly added edge, which will eventually result in the correct SSSP tree being built. In Pagerank, vertex values depend on the number of edges of their incoming siblings so when an edge is added, all outgoing siblings of the source vertex (including the destination of the newly added edge) need to recompute their value. To that end, we place all the outgoing neighbours of the source vertex in the work queue.

With an understanding of the semantics of the vertex algorithm, a number of optimisations are possible. For instance, in SSSP it is possible to check if the edge addition will lead to a reduction of the distance of the destination vertex. If it does not, then the structural update can be safely ignored.

Edge removal: In *always-converging* programs, edge removals are handled exactly as edge additions. For *monotonic* programs treatment of deletions is more complex. The reason for

Algorithm 3: Function called when adding an edge for SSSP and Pagerank.

```
1 on_edge_addition_sssp(edge e) {
2     push_in_next_wq(e.src);
3 }
4 on_edge_addition_Pagerank(edge e) {
5     foreach_outgoing_edges(e.src, dst)
6         push_in_next_wq(dst);
7 }
```

this is that, unlike for edge addition, the current state of the computation as reflected by the current vertex values, no longer results from a correct sequence of invocations of the vertex programs since some of these invocations may reflect the presence of the edge just deleted.

Figure 7.1 shows an example of an SSSP computation where after the deletion of an edge, simply invoking the vertex programs of the affected vertices does not converge to the correct solution. In *monotonic* programs, we first need to undo the effects that the deleted edge may have had on the state of the computation, a process we call invalidation, before we can move forward and recompute towards the new solution.

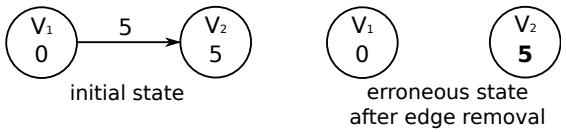


Figure 7.1 – Edge removal in SSSP. Even if the vertex program is called on both vertices, the 2nd vertex will keep its old (and now wrong) value (here: 5 instead of infinity)

The removal of an edge is done in two serialised steps in *monotonic* programs. The goal of the first step is to undo the effects of the deleted edge, i.e., invalidate all vertices which depend on that edge. This is done recursively by traversing the graph. We require *monotonic* programs to maintain a "father" field in vertices that tracks dependencies (see Line 6 of Algorithm 1). If vertex N1 is the father of vertex N2, it means that value of N2 depends on the edge between N1 and N2. Snowy uses this "father-son" relationship to recursively invalidate vertices in the graph.

Once all dependencies have been invalidated, the vertices in the graph either have a correct value or have been invalidated. The second step consists in bringing back the invalidated vertices to a correct state, i.e., give them a correct value. This step requires the application developer to push vertices that could give a correct value to the invalidated vertices in the work queue. Note that we wait until all vertices that depend on an edge are invalidated before trying to give them a new value. This is important to avoid "invalidation loops". Figure 7.2 exemplifies what may happen if vertices try to pull their new value before invalidation is complete. In this example 3 vertices (V_2, V_3, V_4) depend on the edge that is being removed. If V_2 pulls a new value before the invalidation phase is done, then it might pull a value from V_3. However, V_3 will be invalidated in the future, which means that V_2 will need to be invalidated again. This can result in infinite loops of invalidations occurring in the graph.

Invalidation is done automatically by Snowy. *Monotonic* programs have to implement the

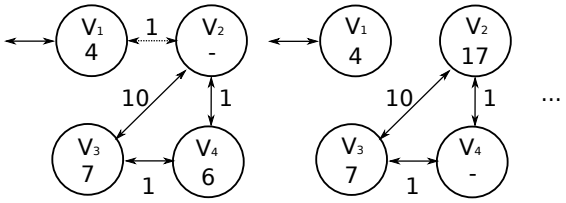


Figure 7.2 – Edge removal. Example of an infinite loop when vertices don't wait for the invalidation phase to be complete.

`after_invalidation` function to support edge removals. The goal of this function is to find vertices that could give a correct value to the invalidated vertices, and push them in the workqueue.

Algorithm 4 presents this function for SSSP. Figure 7.3 presents an example of invalidation. First the vertices that depended on the removed edges are recursively invalidated (marked with a "-" in the figure, step 1 and 2). Then the `after_invalidation` function is called and correct values start being pushed to the invalidated vertices (step 3 and 4).

Algorithm 4: Generic function to recursively invalidate the dependencies in the graph, and after_invalidation function for SSSP. In SSSP we need to push all the incoming vertices of invalidated vertices in the work queue; these vertices will then push their value to the invalidated vertex.

```
1  invalidate(vertex src, vertex dst) {
2      // Handled by Snowy automatically
3      lock(dst);
4      if(dst.father == src && dst.state == valid){
5          dst.state = invalid;
6          push_in_next_wq(dst);
7      }
8      unlock(dst);
9  }
10 after_invalidation(vertex v) {
11     v.distance = inf;
12     v.state = valid;
13     foreach_incoming_edges(v, dst)
14         push_in_next_wq(dst);
15 }
```

Edge weight modification: In *always-converging* programs, edge modifications are handled exactly as edge additions.

For *monotonic* programs, we distinguish between weight increase and weight decrease. If the *monotonic* program can only decrease the value of vertices (as it is the case with the distance in SSSP), and the weight of the edge is decreased, then the edge modification is treated as an edge addition. Intuitively this can be seen as "adding a better edge" between the two vertices. If the weight of the edge is increased, then intuitively the values propagated through the edge with the previous weight might now be wrong, so the weight modification is treated as a removal. The opposite is done for *monotonic* programs that can only increase the value of vertices.

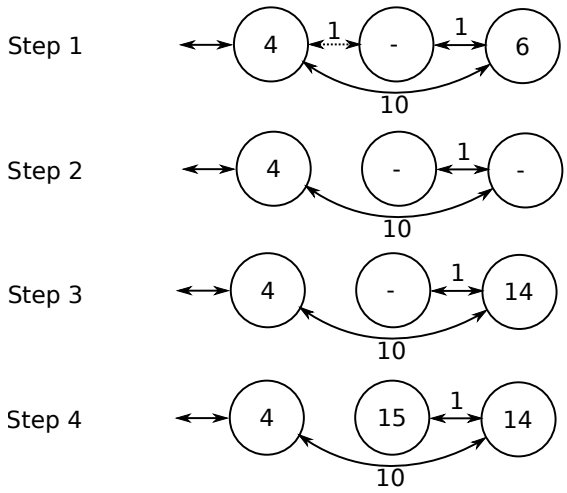


Figure 7.3 – Example of edge removal. Refer to the text for a detailed explanatio of the 4 steps.

Summary: To add support for incremental execution, developers mainly have to find the set of vertices that might be affected by an update, or that can push a correct value to a vertex. Update threads then have to push these vertices in the work queue, and re-computation of the correct values is triggered. For *monotonic* programs, the invalidation of values that depend on a removed edge is performed automatically by Snowy.

7.2 Interfacing to Snowy

Snowy continuously applies structural changes while running the vertex program. It is thus possible that, if updates are pushed with a high rate, Snowy constantly has pending work and the graph never reaches a stable state. This raises important questions that we answer in this section:

- What does it mean to perform requests on the graph when a monotonic program has not been run to convergence? Are results correct in the middle of an invalidation phase?
- How do always-converging programs behave? What does it mean to read the value of a vertex in the middle of a computation?

7.2.1 Monotonic programs

Monotonic programs maintain the following properties on vertices: if a vertex is "valid" then its value is the correct value or it reflects a correct state of the graph in the past. Otherwise the vertex is marked as invalid. When querying the graph, it is thus possible to know if the individual vertex values can be trusted or not. Our programming model on monotonic programs is as follows: any given request will read the values of some vertices; if any of these vertices is invalid, we restart the request from scratch.

We exemplify the use of this programming model with SSSP. In SSSP, a developer usually wants to know the shortest path between a vertex and the root vertex. The answer to that request is a

path that contains only valid vertices. To build this path, we use the father-son relationships. If a vertex N1 has a father N2, it means that a path leading to the root from N1 existed, and that this path was going through the edge N2-N1.

In the case of edge additions, all vertices of the graph are always in a valid state. It means that a path containing only valid vertices can always be found by following the father-son relationships. Snowy does not guarantee that this path is the current optimal one, but that the path is in between an optimal path of the (recent) past and the current optimal one. Note that existing graph analytic engines that batch updates or re-run the full analytics to get fresh results also return values from a past state of the graph. In these engines, updates that arrive when the batch is executed, or when the analytics are running are delayed until the next batch or the next re-computation. Snowy on the opposite integrates new updates in the computation on the fly, and is thus more likely to reflect their effect earlier in the computation.

The main issue with *monotonic* programs comes with edge removals and the invalidations they induce. When an edge is removed, it is possible that part of the graph becomes invalid, so it is possible to end up with a path with invalid vertices. Our solution in that case is to retry building the path as long as it has invalid vertices on it.

In practice we found that this solution was working very well, even with substantial update rates (millions of updates / s). This is due to the fact that most updates in monotonic programs are reflected within few microseconds and only impact none or few vertices. So even with substantial update rates, the vast majority of the graph is in a valid state. For instance, on an RMAT25 graph (530 million edges), when updating 10 million edges every second (5% of these updates being removals), we found out that, at any given time, only a maximum of 0.3% of the vertex had an invalid path to the root. Section 7.4 contains more details on latency measurements, and probabilities of having to retry a request.

7.2.2 Always-converging programs

Requests in always-converging programs consist in reading the value of a vertex, or finding an order between vertices. Always-converging programs do not have a clear notion of "valid" vs. "invalid" vertex values and work on approximations. Values of vertices may never be exactly equal to their theoretical correct value, but rather the more iterations of the program are run, the closer the values are to their theoretical correct value.

The programming model of Snowy is based on the following assumption: adding or removing a single edge mainly has local effects on the graph, i.e., adding or removing an edge does not drastically change values of all vertices in the graph, but only strongly impacts the values of vertices close to the edge. This assumption is verified in practice on most convergence based algorithms when run on large power-law graphs: vertices propagate values to their neighbouring vertices and lose "importance" as they get re-propagated towards more distant vertices. This has two consequences on the convergence behaviour of algorithms: (i) the important effects of an update are reflected during the few iterations that follow the update

(important propagation effects are local), and (ii) even with high update rates, most of the vertices in the graph do not change significantly (graphs are large and effects are local).

This behaviour of always-converging programs might seem counter-intuitive, so we exemplify it with an analysis of Pagerank. Figure 7.4 presents the value of the Pagerank of the top 10 vertices of an RMAT25 graph. Before iteration 28, the graph contains all edges but the edges of vertex 0, and the algorithm has been run to convergence. In between iteration 28 and 29 we add all edges of vertex 0 to the graph. Vertex 0 is the biggest vertex of the graph (it has 600K edges), so one might expect that suddenly adding such a vertex to the graph would have dramatic and non-local effects on most rankings in the graph. Instead we observe that (i) after just 1 iteration of Pagerank vertex 0 has a high Pagerank, and that its Pagerank value does not fluctuate much in the subsequent iterations, and (ii) the Pagerank of the other vertices in the top 10 is not strongly affected by the updates. This can easily be explained as follows: even though vertex 0 has the highest Pagerank of all vertices (value of 20K), its contribution to the Pagerank of its neighbours is small. According to line 5 of Algorithm 2, the contribution of vertex 0 is equal to its Pagerank divided by its degree, which is approximately 20K/600K = 0.03, a value negligible compared with the Pagerank of the top ten vertices (3K-20K).

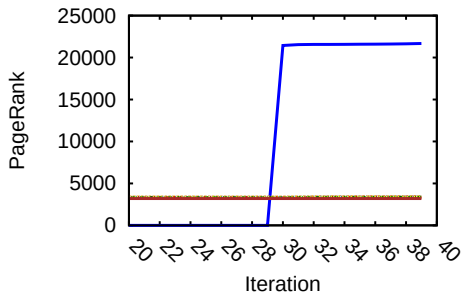


Figure 7.4 – Pagerank of the top 10 vertices of a graph (vertices 2-10 have a value of 3000). At iteration 28 all edges of vertex 0 are added in the graph. At iteration 29 changes have already been fully reflected.

As a consequence, Snowy always considers values of all vertices to be correct, because they either reflect a view from the (recent) past, or a value very close to the theoretical optimal value. In Section 7.4.4 we give more examples of the behaviour of always-converging programs and show that in all cases values in the graph are always close to their theoretical optimal or reflect a correct view of the (recent) past state of the graph, even when updates are added while computing on the graph.

7.3 Implementation

We focus on two aspects of the implementation. First, we discuss the data structures used to represent the graph. Second, we discuss concurrency issues and efficient synchronisation mechanisms to resolve these issues.

7.3.1 Data structures

Since the number of edges is much larger than the number of vertices, we focus on memory-efficient representation of the edges. Since we expect lookups of edges to dominate, followed by addition of edges, with relatively few deletions, we focus on time-efficient implementation of search and addition.

Vertices are represented by records in an array indexed by the vertex identifier. Each vertex record maintains a list of outgoing and incoming edges, identified by the vertex identifier of the other endpoint of the edge, including the edge weight, if applicable. The following describes the representation for out-edges. The representation of in-edges is similar.

One of the efficiency issues faced by the implementation is that the number of edges greatly varies between vertices, especially in power-law graphs with Zipfian degree distributions. A small number of vertices has a large number of edges, while most vertices have very few. For instance, in the RMAT-25 graph [33], most vertices have no more than a few tens of edges, while one vertex has 600,000 edges.

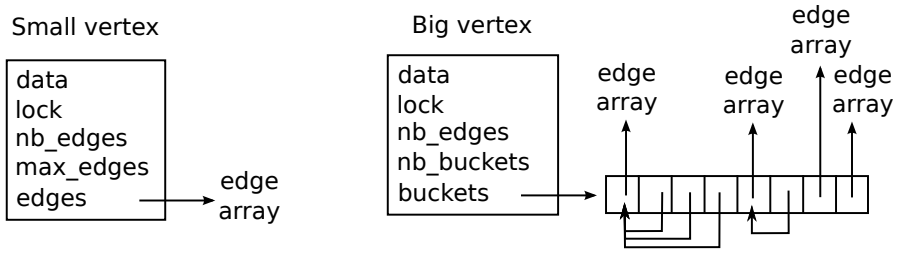


Figure 7.5 – In memory representation of a vertex.

For vertices with a small number of edges (up to a maximum of M), the edges are maintained in an unordered edge array, pointed to by the vertex record. The edge array is allocated on demand when the first edge is added. The initially allocated array can store K edges, and is doubled in size every time the array overflows. The vertex record stores the number of edges, making addition of an edge trivial. Lookup requires a linear scan of a (small) array.

For vertices with a large number of edges, the edges are maintained in a number of such edge arrays, pointed to by entries in a bucket list, which in turn is pointed to by the vertex record. Figure 7.5 represents the in-memory representation of a vertex with a small number of edges, and a vertex with a large number of edges. When the number of edges exceeds M, a bucket list with, in general, 2^N pointers is allocated, pointing to 2^N edge arrays. Edges go into a particular edge array, by hashing the vertex identifier using a Marsaglia XorShift [89], and taking the top N bits.

When the initial edge array overflows, a bucket list of size 2 is allocated, with two edge arrays, and the edges are distributed over these two arrays according to the higher-order bit of the hash function. If one of the edge arrays overflows again (let's say that edge array 0 overflows), then the bucket list is doubled in size, two new edge arrays are allocated, and the edges in

edge array 0 are split over the buckets 00 and 01, according to the two higher-order bits of the hash function. The edge array 1 is placed in entry 10, the entry 11 in the bucket list points to it. If, at some later point edge array 10 overflows, then no doubling of the bucket list is necessary. The edges of array 10 are split between edge arrays 10 and 11. In any case, when an edge array overflows, only the edges of that edge array are split between the two new edge arrays. The other edge arrays are left untouched.

The chosen data structures allow for reasonably fast lookup (one or two pointer dereferences plus a linear search through a small array). Insertion is also fast (insertion in an array), except in the relative rare case where a new edge array needs to be allocated. Memory utilisation is high, which is important with big graphs. The design is a compromise between, on the one hand, a single unsorted array, which has optimal memory utilisation, but makes for an expensive lookup for high-degree vertices, and, on the other hand, a single sorted array, which has log(n) rather than linear lookup, but implies expensive addition of edges. Other data structures can be considered, but we have found the chosen data structure to have good performance for our applications. We evaluate our choice in more detail in section 7.4.

7.3.2 Work queue

The presentation in Section 2 uses a work queue per iteration. Since a particular iteration only uses the current work queue and the work queue for the next iteration, we only need two work queues, and we switch atomically between the two at the end of an iteration. The size of the work queue is equal to the number of vertices. This requires us to guarantee that no vertex is inserted more than once in the queue.

7.3.3 Concurrency and synchronisation

We use a fixed number of update threads and a fixed number of computation threads. The update threads process incoming updates, and put affected vertices in the work queue. The computation threads read vertices from the work queue, modifies the vertex values, and possibly adds other vertices to the queue.

Access to edge arrays

When a thread iterates through the edges of a vertex, or when an edge is added/deleted/searched in the edge list, then the edge list is locked. We found the overhead of locking the edge lists to be negligible: in practice two threads never try to lock the same edge list at the same time. This can be intuitively explained by the following facts: (i) compute threads always iterate on different vertices (a vertex is present only once in a work-queue), so conflicts can only occur between update threads and compute threads or between an update thread and another update thread, (ii) the number of vertices is far superior to the number of threads (millions of vertices vs. dozen of threads), thus the probability of two threads working on the same vertex is extremely low. We study in detail the impact of locking in section 7.4.

Workqueue

Threads access the work queue, to remove a vertex or to add vertices. Since we need to guarantee that a vertex only appears once in a work queue, we lock the vertex record, check if the vertex is already in the queue (using a flag in the vertex record). If not, we set the flag and add the vertex to the end of the work queue, and unlock the vertex record.

In its simplest form, we would have to lock the queue before performing the insertion, but such a lock would experience a lot of contention, because all computation threads continually add vertices to the queue. Instead, we have each computation thread store the vertices in a local buffer, and when the buffer is full, flush the buffer to the queue, while holding the queue lock. This buffering causes a problem, when the queues are switched. At this point it must be guaranteed that the buffers are flushed. To achieve this, a lock is acquired on the buffer, and the buffers are flushed to the next queue.

7.4 Evaluation

We first describe our experimental setup including evaluated algorithms, datasets and hardware. We next present and discuss the results of our experiments and address the following questions:

- We report the maximum update ingestion rate that Snowy can sustain.
- We report and analyse the latency for monotonically converging and always converging applications.
- We compare Snowy's performance to that of state of the art (dynamic and static) systems.
- We evaluate the implementation decisions.

For simplicity, we only show latency numbers in the case of edge additions and removals. Vertex additions and removals are a special case of edge modifications. We load the first half of the graph and apply the remaining half as updates. Removals try to remove the first half of the graph first.

7.4.1 Experimental environment, algorithms, and datasets

Experimental environment: We evaluate Snowy on two machines. Machine A has 4 AMD Opteron 6272 processors, each with 8 cores (32 cores in total). The machine has 256GB of RAM. Machine B has 4 Intel Xeon E5-4650 processors, each with 8 cores (32 cores in total) and 1.5TB of RAM. Unless stated otherwise, all experiments are run on machine B, since its large memory allows us to experiment with larger graphs.

Algorithms: We select six algorithms with different characteristics in terms of functionality (traversal, machine learning, ranking), vertex metadata as well as number of vertices active in

the computation: **Breadth-first search (BFS)**,**Single source shortest path (SSSP)**, **(Weakly) connected components (WCC)**, **Pagerank (PR)** [29], **Alternating Least Squares (ALS)**, and **Belief Propagation (BP)** [70] .

Datasets: We use both synthetic and real-world datasets to evaluate Snowy. The synthetic datasets are power-law graphs generated by the RMAT graph generator. This generator can generate graphs of different sizes, allowing us to evaluate Snowy's scalability in terms of graph size. In particular, RMAT31 (256GB, 34 billion edges) is the largest graphs that fits in memory on machine B with the large amount of main memory. The real-world datasets are the Twitter follower graph [77], the UK-2002 web graph, and the Netflix graph. The Netflix graph is only used in the ALS algorithm.

Pagerank and SSSP take directed graphs as input, while BFS and WCC operate on undirected graphs. The undirected graphs are generated by adding the opposite edges of the corresponding directed graphs. Table 7.1 gives an overview of graphs used along with the number of vertices and edges in millions. The undirected versions of the graphs have twice as many edges and the same number of vertices.

Graph	Vertices	Edges
RMAT-N	2^N	2^{N+4}
Twitter	62M	14,684M
UK-2002	2M	298M
Netflix	0.5M	100M

Table 7.1 – Graphs used in the evaluation.

7.4.2 Maximum update ingestion rate

The maximum update ingestion rate that Snowy can sustain is equal to the maximum rate at which the updates can be applied to the data structures representing the graph. Table 7.2 summarises the maximum update ingestion rate on machines A and B, for additions, removals and weight changes. On machine A and B Snowy can sustain a constant rate of 11 million edges additions per second, and 6 million modifications or removals per second. Machine B has a lower memory bandwidth, which explains the inferior results. Removals and weight modifications are less efficient than additions, because the latter do not require a scan of the edge array. The ingestion rate is only modestly impacted by the size of the graph: Snowy is only 10% slower on RMAT31 (with 34 billion edges) than on RMAT27 (with 2.1 billion edges). This good scalability is the result of our bucket list implementation of edge arrays, which bounds the cost of most operations to the computation of a hash, a pointer dereference, and, in the case of removal and weight modification, the streaming of a small array.

7.4.3 Monotonically-converging programs

In this section we present the performance of Snowy on monotonic programs. We show the latency between the time an update arrives in the system and the time when its effects have been fully reflected. We further measure the latencies when varying different parameters and

Machine	Graph	Add./s	Removals/s	Mod./s
A	RMAT25	12.5m	6.1m	6.1m
	RMAT27	11.4m	5.6m	5.6m
B	RMAT27	11m	4.7m	4.7m
	RMAT31	10m	4.2m	4.2m

Table 7.2 – Maximum update ingestion rates for various graph sizes, for additions, removals and edge modifications.

show the success rate of queries performed on the graph.

Latency: In Snowy multiple updates are being processed concurrently, and one update may cancel out the effect of another one. It is therefore not entirely straightforward to measure the latency of a particular update. We use the following strategy for performing these measurements. We keep a table with one entry for each update that stores the start and end time when this update is processed. When an update thread starts processing an update, it gives it a unique update identifier, used to index the table, and stores the current time as the start time for that update. When the value of a vertex is changed, that vertex is tagged with the update identifier, and the current time is written into the end time of that update. Furthermore, the vertices pushed on to the work queue as a result of the update are tagged as well. In the end we obtain the latencies of all updates, and we can derive a distribution for them.

Table 7.3 reports the average, 99 percentile and maximum latency value of the update latency distribution for BFS, SSSP and WCC on RMAT25 for an update rate of 1,000,000 updates per second. The updates consist of 99% edge additions and 1% edge removals, a division between additions and removals also assumed in other work on incremental graph processing [116].

Algorithm	Average	99th percentile	Maximum
BFS	6μs	3μs	518ms
SSSP	2.5μs	10μs	385ms
WCC	2.6μs	5μs	380ms

Table 7.3 – Characteristics of the update latency distribution for various algorithms on RMAT25.

The update latency of the overwhelming number of updates is close to zero, with a few outliers having a high update latency. This is due to the fact that the update either has no effect whatsoever on the result, or its effect can be determined immediately. This corresponds to our intuition for these algorithms. For instance, for BFS or SSSP, the removal of an edge not part of the tree has no effect, which can be determined immediately. However, the removal of an edge near the root of the tree causes lengthy invalidation and re-computation. In practice, there are very few edges near the root and thus this happens rarely. A more detailed analysis of the SSSP experiment reported in Table 7.3 shows that 97% of edge modifications have no impact at all, and that 99.9% of edges impact less than 5 vertices. The maximum impact of any single edge is 340,000 vertices, which in RMAT25 amounts to 0.9% of all vertices.

In Tables 7.5 to 7.4 we report results for some variations in the parameters of the above experiment. In Table 7.5, we vary the ingestion rate from 1,000,000 updates per second to the maximum sustainable rate of 10,000,000 per second. In Table 7.6 we use RMAT29-und, the

largest graph for which we can measure the latency, as well as two real world datasets, the Twitter and the UK-2002 web graph. These experiments use 1% removals and 99% additions. In Table 7.4, we vary the % of removals included in the updates. The results are reported only for BFS on Machine B. Results for SSSP and WCC are similar.

Table 7.5 shows that the latency distributions stay the same under higher input rates, which means that compute threads are not the bottleneck and are able to keep up with update threads up to the maximum structural update rate. Table 7.6 furthermore shows that for a graph 16× larger than the one used in Table 2, the maximum latency increases by only 17 percent. As expected, the latency increases considerably for removals, because of the need for invalidations.

% of removals	Average	99th perc.	Maximum
0	5μs	3μs	500ms
10	6μs	10μs	390ms
30	0.3ms	12μs	4.5s
50	0.3ms	13μs	2.9s
100	7ms	98ms	10.3s

Table 7.4 – Characteristics of the update latency distribution when varying the percentage of removals for BFS on RMAT25.

Update rate	Average	99th perc.	Maximum
1,000,000 U/s	6μs	5μs	510ms
3,000,000 U/s	6μs	5μs	510ms
5,000,000 U/s	6μs	5μs	520ms
10,000,000 U/s	6μs	5μs	502ms

Table 7.5 – Characteristics of the update latency distribution when varying the input rate for BFS on RMAT25.

	Input graph	Average	99th perc.	Maximum
BFS	RMAT-29-und	6μs	16μs	600ms
	Twitter	1μs	3μs	92ms
	UK-2002	1.5μs	2μs	131ms

Table 7.6 – Characteristics of the update latency distribution for RMAT29 - undirected, Twitter and the UK-2002 Webgraph.

Request success rate: Figure 7.6 presents, for SSSP on RMAT25, with an input rate of 10 million updates per second (5% edge removals, 95% edge additions), the percentage of vertices that have at least one invalid vertex on their shortest path from the source vertex. The number is 0 most of the time, and spikes for a short amount of time to a value below 0.3%. Therefore, the number of requests for the shortest path from a vertex to the source that have to be retried because of an invalid vertex in the path is extremely small, and one retry most often suffices. The success rate is even higher for algorithms like WCC, where a request accesses only a single vertex rather than a path.

Intuitively this can be explained as follows: (i) most updates have no impact on the result, (ii) most edge removals only impact a very small number of vertices, so the probability of these

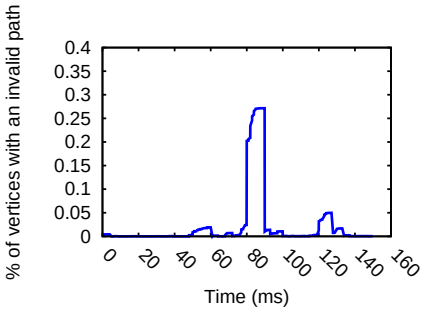


Figure 7.6 – Percentage of vertices that have an invalid path at any given time for SSSP on RMAT25.

vertices belonging to a path is extremely small, (iii) iterations of these algorithms are very short (lower than 1 millisecond most of the time), so the effects of updates are propagated very quickly.

7.4.4 Always-converging programs

Unlike monotonic algorithms, changes in always-converging algorithms such as Pagerank, ALS and BP do not have a single vertex responsible for their change. As long as updates are pushed into the system, the values will be updated. In this section, we aim to provide an idea of how quickly structural updates are reflected in the computation and how the vertex values change in the presence of updates. We provide empirical evidence to demonstrate that always-converging algorithms tend to behave predictably and converge quickly towards the fix-point solution in the presence of random updates on power-law degree graphs.

We run the experiments on Pagerank, ALS and BP and present the results in Figure 7.7. For Pagerank, we add random updates to the graph at a rate of 10m updates / second. For ALS we report the predicted score of the top 10 movies of a user. Before iteration 5, ratings of the user have not been added in the graph, so the predictions for top 10 movies are based on the average movie ratings of all users. At iteration 4 we add all rankings of the user by first adding them into the data structure and then compute. We also perform the same experiment but ratings of the user are continuously added to the graph while it runs between iterations 4 and 13. For BP we add a very connected vertex with all its dependencies and random updates at a rate of 10 million updates per second.

As can be observed in all cases, each algorithm converges rapidly and usually a single iteration is sufficient to reflect the changes in vertex values. We can further observe that even if the values do not converge immediately, the rankings (ratings) reach a stable state. Even if computation and updates overlap, the vertices reach and remain close to a new stable value. In the case of ALS, the predictions are reflected within one iteration but evolve (slightly) over time to to the extra added ratings. As for Pagerank, even though the values keep slightly changing, the absolute impact of adding edges, even at a very high rate is negligible.

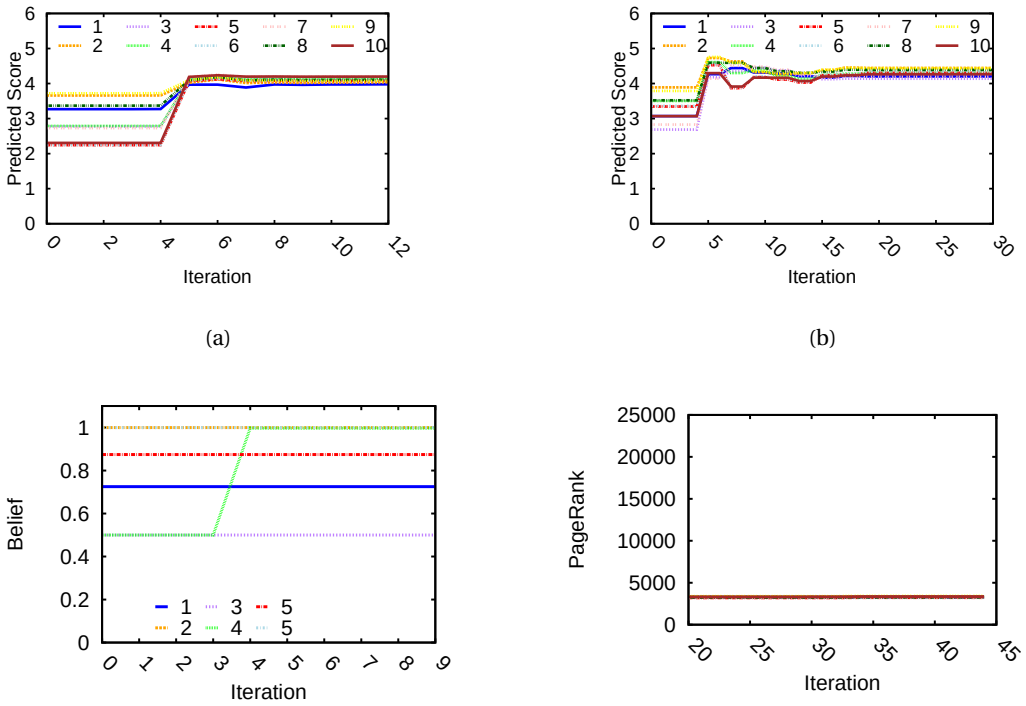


Figure 7.7 – Evolution of metrics through time. (a and b) ALS: predicted rating of the top 10 movies of a user on the Netflix dataset. Before iteration 4 the user has not been added to the graph and the recommendations are not based on his ratings. (a) All ratings of the user are added between iteration 4 and 5 (b) Same as (a) but ratings of the user are continuously added to the graph while it runs between iterations 4 and 13. In both cases the predicted ratings converge quickly towards their final value. (c) Belief of 5 random vertices on the graph. At iteration 3 the edges of the vertex 4 are added to the graph. On top of that, 10 million random updates are also pushed per second. (d) PR: evolution of 10 vertices with the highest Pagerank, performing random updates at the rate of 10 millions updates/s. The rankings do not change much.

7.4.5 Comparison to other systems

In this section, we compare Snowy against recent incremental graph processing systems and demonstrate that the performance of Snowy on a static graph is on par with that of the best single-machine static graph processing systems.

Incremental graph processing systems: Most incremental graph processing systems process graphs in batches of updates executed sequentially. Snowy does not batch updates, but rather applies them immediately to the graph while computation is running. Table 7.7 compares the runtimes of weakly-connected components (WCC) on Naiad and Snowy. Naiad is a general-purpose dataflow system which supports, among others, incremental graph algorithms. Naiad has to make a tradeoff between latency (small batches) and throughput (large batches). Snowy does not do batching and achieves both a higher throughput and a lower latency than Naiad.

GraphIn is a recent system designed to compute on time-evolving graphs in batches of updates. The authors report high throughput numbers on small graphs (9 million updates / second on RMAT20 running WCC), but these numbers drop quickly as the graph size increases (2 million updates / second on RMAT22).

System	Machines	Input graph	Algorithm	Batch size	Throughput	Avg Latency
Naiad	1	RMAT-20	WCC	1K	2.5KU/s	0.3s
Naiad	1	RMAT-20	WCC	1M	125KU/s	7.9s
Snowy	1	RMAT-20	WCC	-	11MU/s	1μs

Table 7.7 – Throughput and latency comparison between Naiad and Snowy

Next, we consider the impact of batch sizes on the performance of batching-based incremental systems. We measure the time it takes Naiad to process 8.8 million updates for various batch sizes. The results for this experiment are shown in Table 7.8. As we can see, Naiad's performance decreases with smaller batch sizes as a result of each batch being marked with an individual epoch which is then scheduled by the Naiad scheduler, to different dataflow vertices. At a high update rate (small batches), this causes the scheduler to become the bottleneck. Furthermore, all changes to the graph are remembered by the system and the memory footprint increases quickly.

Batch size	Total time	Average per-batch time
100	16239.256s	0.194s
1000	3327.171s	0.397s
10000	171.833s	0.205s
100000	86.730s	1.032s
1.000.000	71.629 s	7.959s

Table 7.8 – Naiad: Time to incrementally compute WCC when adding 8.8 million edges in varying batch sizes to RMAT20

Static graph processing systems: We compare the runtime of Snowy on BFS and PR using a static RMAT-27 graph against other well-known static graph processing systems in order to evaluate the impact of supporting incremental graph processing. Table 7.9 shows the preprocessing times and runtimes of Polymer [138], Galois [98], and X-Stream [114], which are three single machine state-of-the-art static graph processing systems.

System	Preprocessing	BFS Compute	PR Compute	Characteristics
Polymer	480s	10.3s	92.2s	NUMA aware graph processing
Galois	8556s	17s	37.2s	Work-list general purpose system
X-Stream (in-memory)	182s	186s	219s	Edge centric streaming
Snowy	121s	12.8s	266.3s	Incremental graph processing

Table 7.9 – Preprocessing and compute times for state-of-the art graph processing systems. BFS and PR on RMAT27.

As can be observed, Snowy's runtime remains in the same order of magnitude as that of static general-purpose graph processing systems, although it is generally a bit slower. Static graph

processing systems can take advantage of the immutable nature of the input graph to perform optimisations, which explains the difference. These optimisations (usually partitioning) come at the cost of increased pre-processing time, which can be amortised over multiple algorithms. However, these optimisations are not as easily achievable in the context of a mutable graph and the modification of a single edge would require re-running both partitioning and the entire algorithm for static systems. In contrast, Snowy can incorporate modifications at comparatively small runtime costs.

7.4.6 Design evaluation

This section evaluates some of our design decisions in the context of dynamic graphs.

Synchronisation overhead: As stated in Section 7.1 mutual exclusion within the algorithm has to be defined by the user. To ensure this we use locks. We also use locks to protect the edge lists from concurrent modifications. This approach goes against the common belief that locking vertices is expensive. On tested graphs and algorithms, we measured that Snowy was spending at most 4.7% of its time in locks (and an average of 3.6%), so locking overhead is small. In practice locks are not contented; we measured that compare and swap operations fail less than once every million operation. This is explained by the size and nature of graph workloads. The graphs of interest have millions of vertices, and only 32 threads competing to process on them. The probability of two threads wanting to lock the same vertex at the same time is thus very low.

CPU utilisation: The power-law nature of real world graphs leads to work imbalance between threads: some vertices have more edges and are thus more costly to process. To address this issue, we divide the work in the current work queue into small chunks (currently 1024 vertices). Compute threads compete to process these chunks. While this strategy might not be optimal for batch processing of large graphs, we found that it worked sufficiently well for graph updates. In practice we measured that the machine is idle only 8% of the time in the worst case when updates are pushed at a high rate (10 million updates / second on Pagerank on Twitter).

Locality: Good locality has shown to improve performance for state of the art graph processing systems [138], but requires a more complex partitioning of the graph. In order to keep Snowy simple, the only memory-related optimisation we perform is to interleave vertices and edge lists on all NUMA nodes of the system to avoid contention issues. Snowy performance might be improved by applying more complex NUMA optimisations.

Edge buckets: The memory overhead of the bucket list is small. We only use of 16B of memory per bucket. On all tested graphs, we measured the bucket occupation to be more than 60%. On all tested graphs and algorithms we measure a memory overhead of less that 1%. The computation overhead is also small. We measured an average insertion time of less than 1μs, and an average search time of 2μs on the tested graphs.

7.5 Summary

We have presented Snowy, a new multi-core system for incremental graph processing. The main novelties in Snowy are that it handles each graph update individually, rather then batching them, and that execution of the graph algorithm proceeds in parallel with new updates being made to the graph. The benefits of doing so are that Snowy can handle updates with very low latency, even at very high update rates. The tradeoff is that the result of the execution of the algorithm may fluctuate, and may occasionally display old or inconsistent results. We have shown that these anomalies are very rare, essentially because the effect of most updates can be incorporated in the result very quickly. We have also described relatively straightforward techniques to deal with these anomalies, should they occur.

8 Graph mining

In this chapter, we present Tesseract, a system designed for general-purpose pattern mining on large evolving graphs. Tesseract supports high-throughput, continuous mining with three key ideas. First, it performs localised graph exploration by efficiently searching for all new or changed pattern instances involving individual updates in the graph. This *update-driven* search is feasible for many GPM problems, such as motif counting and keyword search, because they are localisable or bounded [51], enabling efficient incremental computation.

Our graph exploration algorithm uses backtracking to search for pattern instances, fully expanding one embedding at a time. Unlike most existing static, general-purpose GPM systems that expand all embeddings in the graph by one vertex at each iteration and must therefore store all intermediate embeddings, Tesseract transparently (re)generates and caches embeddings, which reduces memory pressure and is especially effective when updates have locality.

Second, our localised exploration algorithm relies on a *novel canonicalisation scheme* for filtering duplicate patterns in the presence of updates, thereby reducing the exploration space significantly, and ensuring that we only explore embeddings affected by an update. We extend this scheme to support batching updates for higher performance, and parallel coordination-free exploration for scale out.

Finally, Tesseract provides a pattern pruner API that allows developers to leverage domain- and pattern-specific optimisations for improving performance. The pattern pruner works by creating vertex or edge properties in the graph that are used to prune the search space.

Our localised exploration strategy, together with our canonicalisation scheme and pattern pruner, offers several benefits. It only computes the *pattern instances impacted* (created or removed) resulting from graph updates. It also makes it easier to scale to large graphs, because intermediate embeddings need not be materialised and kept in memory beyond their useful life cycle, thus requiring much less memory than existing systems. As updates arrive, it allows new pattern instances to be output with low delay and streamed for further processing. Our

execution model processes updates independently, enabling parallel exploration with minimal co-ordination. Our approach is also expected to be beneficial when applications are only interested in discovering and monitoring patterns involving a subset of a large graph.

We evaluate the performance of Tesseract and compare against BigJoin Dataflow [20], a recent subgraph query system supporting dynamic graphs. We show that Tesseract can ingest updates at a rate of up to 20 million per second on a single machine, and scales linearly with the number of nodes in a distributed setting. Tesseract is between 1.5X and 5X faster than BigJoin on dynamic graphs, all the while offering a more general computation model and being less sensitive to input ordering. When ingesting an entire static graph, Tesseract offers comparable performance to Arabesque [122] and Fractal [46], and outperforms RStream [129], three state-of-the art GPM systems for static graphs. Finally, we demonstrate how Tesseract can monitor changing pattern instances in large graphs.

We make the following contributions in this chapter:

- We present Tesseract, the first system to support continuous *general* mining on evolving graphs (8.2).
- We propose a novel, update-driven, localised search strategy that transparently computes and caches pattern instances affected by an update (8.2.1).
- We describe a canonicalisation technique that enables co-ordination-free, pattern exploration on multiple workers in the presence of graph updates, and show that this technique makes it possible to only consider pattern instances resulting from an update (8.2.2).
- We propose a pattern pruner API that allows developers to optimise our exploration algorithm by aggressively pruning the search space 8.2.3).
- We show that Tesseract can handle millions of updates per second on a single machine, and it offers significantly better performance than competing systems (8.4).

The rest of the chapter describes our approach in detail. Section 8.1 motivates the need for a new approach to dynamic GPM. Section 8.2 describes core techniques used by Tesseract to support dynamic graphs. Section 8.3 describes its implementation. Section 8.4 evaluates the performance of Tesseract on typical datasets, and compares it with other systems. Finally Section 8.5 provides our conclusions.

8.1 Background and Motivation

In this section, we provide background on GPM, and motivate the need for efficiently mining evolving graphs.

GPM problems aim to discover instances of interesting patterns in a graph dataset. The graph can be either directed or undirected, with labels attached to vertices and edges. Labels include identifiers (usually integers) as well as user-defined properties. Patterns are arbitrary connected subgraphs.

GPM is done via subgraph matching, i.e. enumerating all subgraphs, commonly referred to as *embeddings*, that match some criteria of interest, such as a specific pattern or certain graph properties (e.g., frequent occurrences in the graph).

Motivating example Consider, for example, the popular problem of *graph keyword search* illustrated in Figure 8.1. Given a set of labels, graph keyword search finds all subgraphs whose vertices contain all the labels of interest. These subgraphs must be minimal, i.e., not contain any unnecessary vertices.

This problem has many practical applications in social networks, recommender systems, and semantic web [128]. Furthermore, many applications require continuous, low-latency output from graph keyword search algorithms as the graph is updated, for instance, to perform ad targeting or to provide live training data to a deep learning model.

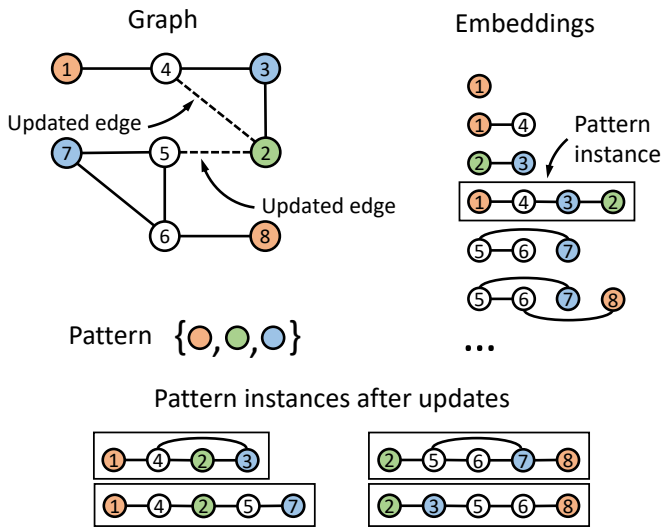


Figure 8.1 – Graph keyword search example

In the example in Figure 8.1, the pattern consists of three labels that are represented as colours (orange, green, or blue). A set of graph embeddings is shown on the top right of the figure. The pattern matches any embedding that contains exactly one vertex of each label. Embeddings may contain other vertices (represented in white). Initially, we ignore the (2, 4) and (2, 5) edges, and the graph has one pattern instance: (1, 4, 3, 2).

Now, suppose the graph were to be updated with edges (2, 4) and (2, 5), and we would like to compute the new pattern instances. Visually, it is easy to see that the addition of these edges results in 3 more pattern instances: (1, 4, 2, 5, 7), (2, 5, 6, 7, 8), and (2, 3, 5, 6, 8). Continuously computing pattern instances as the graph evolves is more challenging.

Current state of the art Current GPM systems can be broadly classified in two categories: general-purpose GPM systems and subgraph query systems.

General-purpose GPM systems [46, 122, 129] expand embeddings iteratively, starting from each vertex, and then adding one vertex (or edge) at a time to enumerate matching embeddings in the graph. At each iteration, they eliminate duplicate embeddings and filter out non-matching embeddings, before outputting all matched pattern instances in a batch. This approach is efficient for a static graph since each iteration reuses the embeddings enumerated in the previous iteration.

However, this dependence on the previous steps makes it difficult to perform incremental computation on an evolving graph, in which pattern instances may be created or deleted anywhere in the graph, and can invalidate an entire tree of expansion steps. Computing these instances incrementally would, in principle, require storing all the intermediate embeddings and being able to index them by all their vertices. For example, with the addition of edge (2, 5) in Figure 8.1, we would need to look for all embeddings containing vertices 2 or 5 and join them together to find new pattern instances. Many existing pattern instances could also be regenerated in the process, and would need to be discarded. Even with a graph dataset of moderate size, this process would have prohibitive storage and computational requirements.

As a result, to the best of our knowledge, no general-purpose GPM system supports incremental computation on evolving graphs. In addition, simply recomputing pattern instances from scratch on the whole graph is not practical at current scales. For instance, the state-of-the-art Arabesque system requires almost 3 hours to compute cliques of 5 vertices on a graph with 4 million vertices using 8 servers.

Subgraph query systems [20, 75, 79, 130] target a subset of GPM problems, matching exact subgraph patterns or patterns expressible as relational queries, enabling optimisations specific to these patterns. They build pattern instances by performing joins on edges in the dataset as directed by the query. There has been much recent work on continuous subgraph query matching in evolving graphs [20, 75, 120]. However, these systems are not designed for mining approximate patterns, such as clique mining [30], or discovering patterns, such as graph keyword search [128] or frequent subgraph mining [63], as these problems cannot be expressed as a single query.

For example, expressing the graph keyword search problem in a subgraph query system would require writing a separate query for each possible combination of labeled vertices. Consider the pattern instances found in Figure 8.1 after updates. A query for (1, 4, 2, 3) would be `q1 := e(a,b), e(b,c), e(c,d), e(b,d), o(a), w(b), g(c), b(d)`, whereas a query for (1, 4, 2, 5, 7) would be `q2 := e(a,b), e(b,c), e(c,d), e(d,e), o(a), w(b), g(c), w(d), b(e)`, assuming `e(x,y)` defines an edge relation, and `o`, `g`, `b`, `w` match label colours orange, green, blue, and white, respectively. Despite the potential for factoring out several subqueries from each individual query, it is obvious that this approach will require significant work.

8.2 Design

In this section, we present the design of Tesseract and describe how its different components work in synergy to support general pattern mining on dynamic graphs.

8.2.1 Update-driven Graph Exploration

Tesseract stores the graph structure (vertices and edges) in an adjacency list format. Vertex and edge labels are stored alongside the graph structure and indexed by a vertex or edge identifier. In addition, edges have an associated timestamp indicating the last time they were updated. Tesseract supports the following updates: 1) addition and deletion of edges, 2) addition and deletion of vertices, 3) addition, deletion and modification of labels. All updates are converted to edge updates: vertex updates add or delete associated edges, label updates delete the corresponding vertex or edge and then add them with the new label value.

Tesseract enumerates vertex embeddings. Vertex embeddings contain *all* edges connecting the vertices in the embedding. Each update may alter the graph so new pattern instances appear or existing pattern instances are deleted. Pattern instances are embeddings that match the pattern. For example, when edge (2, 5) is added in Figure 8.1, three new pattern instances are created: (1, 4, 2, 5, 7), (2, 5, 6, 7, 8), and (2, 3, 5, 6, 8). Pattern instances can also be removed, if the corresponding embedding becomes disconnected or invalid, or modified, if the edges or labels in the embedding change while still remaining a valid pattern instance. Tesseract outputs embeddings with the *new*, *removed* or *modified* status.

Tesseract provides the following *correctness guarantee* for evolving graphs. If a graph is updated by only adding edges, then at any point in the lifetime of the graph, the set of all embeddings that have been output correspond exactly to the set of embeddings produced by executing an equivalent static GPM system on the graph with all updates applied up to that point. In the presence of deletes, Tesseract will also output corresponding pairs of added and deleted embeddings, and modified embeddings.

Localised Exploration Algorithm

The key idea behind Tesseract's design is to compute pattern instances resulting from an update using the localised graph exploration strategy shown in Algorithm 5. In following algorithms and text, Tesseract's API functions (implemented by developers) are represented in `typewriter` font, and Tesseract's own functions (implemented in the system) are represented in SMALL CAPS font.

Upon receiving an edge update Δu, Tesseract applies the update to the graph and updates pattern-specific state in the pattern pruner (8.2.3). Then it invokes EXPLORE with this edge and an initial embedding e, containing this edge and its two vertex endpoints. A deleted edge is marked as invalid, but not removed, and thus treated similar to an added edge.

Algorithm 5: The EXPLORE Algorithm

```
1  upon update Δu containing edge (v, u) do
2      APPLY(G, (v, u))
3      pattern_update(G, (v, u))
4      EXPLORE(Δu, [v, u])
5  end

   input :Δu update
   input :e embedding

6  function EXPLORE(Δu, e) is
7      foreach neighbor v of e do
8          if pattern_filter(e, v) then
9              if CAN_EXPAND(e, v) then
10                 e' ← EXPAND(e, v)
11                 if filter(e') then
12                     if match(e') then
13                         OUTPUT(e' ⊖ Δu, e' ⊕ Δu)
14                     EXPLORE(Δu, e')
```

The EXPLORE algorithm employs *backtracking* to search for embeddings matching the pattern of interest. At each step of the algorithm, we expand the embedding with a neighbour to build embeddings *in depth* fully. To do so, the function loops through all neighbouring vertices of the embedding to look for possible extensions of the embedding. It first performs two types of filtering on the existing embedding e, and a candidate neighbour v. The `pattern_filter` function performs fast pattern-specific filtering using the pattern pruner (8.2.3). The CAN_-EXPAND function filters duplicate embeddings (8.2.2). If both these filters pass, the expanded embedding is created by adding the neighbouring vertex and all its edges connecting to the existing embedding. Next, this expanded embedding is filtered using the user-defined `filter` function, which returns true if expanding this embedding further *may* lead to a match. In this case, we finally `match` the expanded embedding against the pattern to decide whether to OUTPUT the embedding, and then invoke EXPLORE recursively on the new embedding.

Our filter-match computation model is inspired by the filter-process model introduced by Arabesque, and is expressive enough to implement a wide variety of GPM algorithms that satisfy the anti-monotonic property [122]. The EXPLORE algorithm completes once the neighbours of the embedding have been explored, which guarantees that all embeddings starting from the initial edge that match the pattern have been explored and passed to OUTPUT. We assume that the algorithms are localisable or bounded [51], and hence `filter` will return false and EXPLORE will stop after exploring a bounded set of neighbours around the update.

Examples Algorithm 6 shows two example GPM applications implemented using the filter-match model: a clique mining algorithm and the graph keyword search with 3 labels algorithm shown in Figure 8.1.

Algorithm 6: Examples of GPM Algorithms

```
1  def algorithm clique_mining as
2      def filter(e)
3          return num_edges(e) == len(e)*(len(e)-1)/2
4      def match(e)
5          return true

6  def algorithm graph_keyword_search as
7      def filter(e)
8          return num_orange(e) <= 1 and num_green(e) <= 1 and num_blue(e) <= 1
9      def match(e)
10         if num_green(e) !=1 or num_orange(e) != 1 or num_blue(e) != 1 then
             return false
11         foreach vertex v in e if color(v) == white do
12             if is_connected(e \ v) then return false
13         return true
```

A clique, also known as a complete subgraph, is a subset of vertices in a graph such that each vertex is connected to all other vertices in the subset. In this example application, the `filter` function checks that the number of edges in the embedding is equal to the number of edges that should be present in a clique of the same size (a clique with n vertices must have exactly $\frac{n(n-1)}{2}$ edges). The `len(e)` function returns the number of vertices in the embedding. Note that the `filter` function checks for cliques of any size, thus allowing mining patterns of varying sizes. However, we would need to limit the maximum size of the clique in this function for localised execution. The `match` function returns true since every filtered embedding is a valid pattern instance. A similar subgraph query system would require enumerating each edge in a clique and need separate queries for each size.

In the graph keyword search example, the `filter` function prunes embeddings if they have more than one vertex of a given colour, since these can never match. The `match` function initially checks that an embedding has exactly one vertex of each colour, and then ensures that the embedding is minimal by checking that the embedding does not contain any unnecessary vertices with other labels (represented as white here). It does so by checking for each white vertex whether the embedding remains a connected graph if that vertex is removed.

Let us now consider how Tesseract's exploration will proceed if the edge (2, 5) is added after (2, 4) to the graph in Figure 8.1. Tesseract will start by running EXPLORE with the initial embedding (2, 5). It will then expand by adding, say, vertex 6, which passes the `filter` check, but `match` will decline to output since the embedding does not contain all 3 colours. Next, the recursive call to EXPLORE will expand the embedding with, say, vertex 7, producing (2, 5, 6, 7), which passes `filter` but not `match`. Finally, the next call to EXPLORE will add, say, vertex 8, which `match` will output, since (2, 5, 6, 7, 8) matches the pattern. Similarly, (2, 3, 5, 6, 8), and (1, 4, 2,

5, 7) will be output as well.

Differential Output Processing

The OUTPUT function outputs the set of all embeddings affected by a graph update. To determine whether an embedding is new, removed or modified, we generate both the pre-update version of the embedding, $e_{pre} = e' \ominus \Delta u$, which removes an added edge from the embedding, and the post-update version of the embedding, $e_{post} = e' \oplus \Delta u$, which removes a deleted edge from the embedding (with a single edge update, only one of these is generated). Then we rerun `filter` and `match` on both the e_{pre} and e_{post} embeddings to decide the status:

1. e_{pre} doesn't match, but e_{post} matches $\implies$ *new*.
2. e_{pre} matches, but e_{post} doesn't match $\implies$ *removed*.
3. Both e_{pre} and e_{post} match $\implies$ *modified*.

This differential processing helps ensure our correctness guarantee across updates.

Caching Embeddings

As the graph evolves over time, Tesseract computes embeddings to discover affected pattern instances. This computation can cause Tesseract to perform a non-trivial amount of re-computation as compared to a static system executing with the benefit of hindsight, i.e. knowing in advance the entire structure of the graph.

Motivation Consider the graph keyword search scenario shown in Figure 8.1, where two edges are added back-to-back in the same neighbourhood of the graph: (2, 4) followed by (2, 5). After (2, 4) is added, the graph has one full pattern instance, (1, 4, 3, 2), and several partial pattern instances, such as (5, 6, 7, 8) and (1, 4, 2). When adding (2, 5), if the partial pattern instance (5, 6, 7, 8) was already cached, we could immediately form (2, 5, 6, 7, 8).

Embeddings Cache Tesseract uses a searchable embeddings cache to store intermediate embeddings to avoid unnecessary re-computation and speed up exploration. Our cache has a fixed size, which helps limit memory requirements, and, unlike in most static systems where all intermediate embeddings are required for correct operation, Tesseract's cache can be dropped without affecting correctness.

Tesseract caches embeddings by storing any embedding of length 3 or more that passes `filter` in a separate trie structure. The trie is indexed by the vertex ids of an embedding, in their sorted order. A node of the trie, both internal and leaf, may contain one embedding, i.e., the full subgraph, including the corresponding vertices, edges and associated labels, ready to be used by `filter` and `match`.

The EXPLORE function caches and looks up embeddings transparently using the vertices in the embedding. When processing an edge update, it first looks up the cache using the lower endpoint vertex to find all embeddings rooted at this vertex (cache hits). For each embedding, the updated edge is expanded using this embedding, and then `filter` and `match` are directly invoked on the expanded embedding. Caching improves performance because we apply these functions on the entire embedding, rather than on every edge of the embedding progressively. The same cache lookup process is performed for the higher endpoint vertex. Next, each embedding expansion that does not appear in the cache (cache miss) is processed using the regular exploration algorithm, considering one neighbour at a time. This process is repeated at every step of the exploration, with a cache lookup using the last vertex in the current embedding, since all previous vertices in the embedding have already been looked up in the cache.

Example Consider again the example in Figure 8.1. Assume for now that edge (2, 5) has not yet been added to the graph. The cache at vertex 1 contains (1, 2, 3, 4), (1, 2, 4), and (1, 3, 4), the cache at vertex 5 contains (5, 6, 7, 8) and the cache at vertex 6 contains (6, 7, 8). When edge (2, 5) is added, Tesseract looks up the cache at vertex 2 for embeddings and finds that it is empty. It then performs a lookup at vertex 5 and finds (5, 6, 7, 8). It then materialises (2, 5, 6, 7, 8) by adding vertex 2 and associated edges, checks against `filter` to find a valid pattern instance. Tesseract then explores prefixes not in the cache at both vertices, i.e., it expands (2, 5) with 3 and recursively explores (2, 5, 3), Note that (2, 5, 6) is not explored since (5, 6) is contained in the cache at vertex 5.

Cache Consistency Maintaining the consistency of the embeddings cache in the presence of edge deletions is essential, since pattern instances may have been deleted, and should therefore be invalidated in the cache. This task is challenging because the embeddings cache stores partial embeddings.

For example, suppose in Figure 8.1 that the updated edges have been added, and thus the cache at vertex 1 contains (1, 2, 3, 4) and (1, 3, 4). Now, suppose edge (3, 4) is deleted. We need to delete all cached embeddings that have the (3, 4) edge. Since (1, 2, 3) is indexed at 1, we cannot look it up efficiently. A similar delete problem would occur in static GPM systems.

Our approach simplifies this task, since it guarantees that we will lookup every cached embedding containing the deleted (3, 4) edge. To see why, recall from Section 8.2.1 that a deleted edge is treated similar to an added edge and the EXPLORE algorithm guarantees that all embeddings starting from an initial edge that match the pattern will be explored. In the process, every partial embedding is explored, and only these embeddings can be cached. As a result, each of these embeddings would be looked up in the cache when exploring the deleted (3, 4) edge. Thus for this deleted edge, for each embedding that is found, we delete node 3, and its descendants (since the trie is sorted by vertex id). In the example above, when computing embeddings from (3, 4), we will find (1, 2, 3, 4) and (1, 3, 4) and trim them to (1, 2) and (1).

Locality When updates have locality, the cache is expected to be particularly effective at reducing the amount of re-computation for the same embeddings or their subsets. For most GPM algorithms, it is likely that previous neighbouring updates will have already discovered pattern instances or subgraphs matching partial pattern instances as part of their processing.

8.2.2 Duplicate Elimination

Definition For vertex-based embeddings, two embeddings with the same vertices and edges, but in different permutation orders are called *automorphic* or duplicate pattern instances.

Challenges Duplicate elimination is highly desirable for GPM systems to guarantee correctness, since duplicate instances should not be exposed to the user, as well as for improving performance, since there is no benefit in exploring the same embedding multiple times. For example, a 4-clique has 24 duplicate embeddings. The Arabesque static mining system [122] performs duplicate elimination using a canonicalisation method that we adapt for evolving graphs.

Static Canonicality Arabesque [122] defines canonicality using two rules:

1. The first vertex in an embedding has the smallest vertex value among all the vertices in the embedding.
2. A vertex in the neighbourhood of the current embedding is added if it has the smallest id and has not been visited yet.

In Figure 8.1, for example, the pattern instance (1, 2, 3, 4) is stored in its canonical form as (1, 4, 3, 2), since it is generated by expanding from (1, 4) and then (1, 4, 3).

The introduction of updates to the graph can break canonicality of existing embeddings, thus requiring recomputation of all embeddings affected by an update. For example, after applying the edge update (2, 4), pattern instance (1, 4, 3, 2) is no longer canonical and must be represented and stored instead as (1, 4, 2, 3).

Strawman Approach A possible solution to eliminate duplicates in dynamic graphs would be to reuse the above static canonicality rule and maintain embeddings in canonical order while performing exploration. In the presence of updates, we cannot expand embedding simply by *appending* vertices, as in static systems, but must perform *inserts* to look for all valid canonical embeddings with the new vertex.

Although correct, this approach is computationally expensive. Given an embedding of length k to extend, it requires checking canonicality for k embeddings. For each of these embeddings, all vertices need to be rechecked in the presence of inserts, and since checking canonicality for a single vertex is linear in the size of the embedding, the complexity of a single expansion step is $O(k^3)$.

Update Canonicality Tesseract performs duplicate filtering in a more efficient way by reformulating canonicality for updates. We observe that by starting exploration with an updated edge as our embedding (instead of a single vertex), the only position where canonicality can break is for this edge. However, canonicality can be enforced for all subsequent expansions. Interestingly, by enforcing the updated edge to be in the first position in the embedding, we can ensure there are no duplicates in the presence of updates.

The CAN_EXPAND check in Algorithm 5 implements update canonicality as follows. Given an embedding $e = (v_1 \ldots v_k)$, and an expansion vertex v:

1. (v_1, v_2) is the updated edge, with $(v_1 < v_2)$.
2. The vertex v is added if, ignoring v_1 and v_2, it has the smallest id and has not been visited yet.

The second rule can be checked in $O(k)$, much faster than the insert-based approach described above.

Example We consider the example of the two updated edge in Figure 8.1 to demonstrate how Tesseract uses update canonicality. Given updated edge (2, 5) as the starting embedding, we can expand using vertices 3, 4, 6 or 7 as all four are valid expansion candidates according to update canonicality. If we expand using 3, then we can expand using 4, 6, or 7. If we then expand with 4, then adding 1 creates (2, 5, 3, 4, 1), an invalid pattern instance. If instead after expanding with 3, we expand with 6, we can no longer expand by 4, but we can expand further using 7, forming (2, 5, 3, 6, 7), an invalid pattern instance or 8, forming (2, 5, 3, 6, 8), a valid pattern instance. If we expand using 4, we cannot expand using 3 as this breaks update canonicality, but we can expand using 1, 6, or 7. Expanding by 1 and 7 forms (2, 5, 4, 1, 7), a pattern instance. Other expansions do not form any pattern instances. If we expand using 6, then we can only expand using 7, or 8. Adding both 7 and 8 forms (2, 5, 6, 7, 8), a pattern instance. Finally, if we expand using 7, then we cannot expand any further according to update canonicality.

Correctness We make an informal argument that update canonicality is correct, i.e. it does not prune any embeddings that should be explored and does not lead to duplicates.

Our update canonicality rules ignore Rule 1 and relax Rule 2 (by ignoring v_1) of static canonicality. As a result, they should not prune embeddings more aggressively than a static system in which the updated embedding is present (our Rule 1). Then the question is whether update canonicality can lead to duplicate embeddings. Such embeddings can be found in two ways: by executing exploration from different starting points or by choosing expansion vertices in different orders from the same exploration. Clearly, the former does not apply since we are considering a single exploration rooted at the updated edge. The latter also cannot happen since we are enforcing canonicality on any expansion vertices, guaranteeing that two vertices in the neighbourhood can only both be added to the embedding in the same order.

8.2.3 Pattern Pruner

Motivation Since subgraph enumeration and matching are costly, GPM systems aim to prune unnecessary exploration, for example through duplicate elimination or by filtering embeddings. However, checking for canonicality is expensive, and `filter` functions can be of arbitrary complexity.

While static mining systems are able to pre-process the graph, such as for breaking symmetry by sorting the vertices by their identifier [20], such pre-processing-based pruning is not feasible for evolving graphs. Fortunately, many GPM algorithms are amenable to pattern-specific optimisations. Tesseract allows programmers to leverage domain expertise to further prune the search space.

Pattern Pruner API This pattern pruner provides two user-defined functions: `pattern_filter`, and `pattern_update`. As shown in Algorithm 5, the `pattern_filter` function takes the original embedding and a neighbouring vertex as input, and uses pattern-specific properties or state, to quickly prune embeddings. This is beneficial since it is run early during exploration, before heavy-weight functions, such as `expand` that expands embeddings with a vertex and all edges connect it to the embedding. Moreover, certain types of filtering can be performed more easily before an embedding is expanded.

The `pattern_update` function is run before exploration is started to update pattern-specific state. It is used to traverse the vertices and edges in the neighbourhood of an update, and mark vertices or edges based on certain graph properties (e.g., degree, labels) so they can be pruned during exploration by `pattern_filter`.

Example In the graph keyword search example of Figure 8.1, we use the pattern pruner as follows:

- Pruning based on vertex labels: Since a pattern instance cannot contain more than one vertex with a given label, we can immediately prune an expansion for a vertex with a label matching one of the labels already in the embedding being explored. We implement this optimisation by having `pattern_filter` return false when this configuration occurs. In Figure 8.1, this could happen if we are currently exploring vertex 7 from embedding (2, 5, 3). This pruning saves the cost of running CAN_EXPAND, `filter`, and `match` to process an embedding that cannot possibly be a pattern instance.

- Pruning based on distance to specific label: If we were given the domain-specific constraint that two labels of interest cannot be separated by more than k vertices, we could prune large subgraphs where there are no labels matching the pattern keywords. This is implemented as follows. Upon receiving a graph update, `pattern_update` runs a traversal up to a distance of k edges around the updated edge and marks vertices with their distance to the closest vertex matching each label, or ∞ if none is found. When performing expansions, `pattern_filter` checks the new vertex for the distance to

labels still missing from the embedding, and return false if any vertex with a missing label is at a distance of ∞, thereby backtracking immediately. In our running example (see Figure 8.1), this would allow a search from (2, 5) to prune (5, 6, 7, 8) if k was 1, since orange vertex 8 is not marked by the `pattern_update` at (2, 5).

Tesseract's pattern pruner offers powerful capabilities to bridge the performance gap between general-purpose and specialised GPM systems. For instance, using the pattern pruner, programmers can implement most optimisations found in subgraph query systems, while at the same time leveraging the expressiveness of a general-purpose GPM system.

8.2.4 Scaling Tesseract

Batching

Motivation Certain types of updates can be problematic for our localised search strategy. For instance, adversarially ordered updates are particularly insidious. Consider, a k-clique mining algorithm where edge additions forming a k-clique are applied as follows: first connect the lowest vertex identifier to all other $k-1$ vertices, repeat the process by adding all edges connecting the second lowest vertex identifier to the other vertices, and repeat until the k-clique is fully formed. In this case, our basic exploration algorithm will enumerate the subgraph $O(k)$ times before finding a clique. As a result, it will perform significantly more work compared to the static case where all edges are available and a single exploration will find the clique. Although caching can mitigate this issue (13), it is beneficial to limit such repeated unsuccessful explorations.

Batching Updates Tesseract limits repeated localised explorations by processing updates in batches, essentially mimicking static mining for the updates within a batch. Recall from Section 8.2.1 that we timestamp edges indicating when the update was received. All updates within a batch are assigned the same timestamp. Upon receiving these updates, Tesseract first applies all updates to the graph to reflect the new structure and labels, updating timestamps as applicable, thereby ensuring that the graph is in a consistent state before proceeding with exploration.

While batching updates mitigates exploring partially updated pattern instances, it does not by itself reduce exploration. For instance, if updates are adjacent to one another in the graph, they are likely to be part of the same pattern instance. This overlap may lead to exploring the same instance from each of these updates. This problem occurs because while we ensure update canonicality for each single update, we may still explore the same instance multiple times from the different updates in the batch.

Batched Update Canonicality Tesseract supports update canonicality for batched updates by imposing a total order on the edges of the graph, thus ensuring that one starting edge in a pattern instance takes precedence over the others. This ordering ensures that pattern instances overlapping more than one update in a batch are only found during exploration

from one of these updates. The total order can be assigned in different ways, a simple one being based on the vertex ids of the edge.

Tesseract enforces batched update canonicality by using the edge timestamps stored in the graph to detect if the expansion of an embedding involves other edges with the same timestamps. When exploring an edge update, if the expansion uses an edge with an identifier lower than the starting updated edge, Tesseract ignores the expansion. Notice the similarity between this rule and canonicality Rule 1 for static graphs: both rules prevent the same pattern instances from being explored starting from a different vertex or edge.

Example Assume edge (2, 4) and (2, 5) in Figure 8.1 were added as part of the same batch and thus had the same timestamp in the graph (all other edges have lower timestamps), and assume that the total order on edges is such that $(2,4) < (2,5)$. The exploration process starting from (2, 5) will not expand using vertex 4, since the resulting embedding will include edge (2, 4), which has a lower edge identifier.

Similarly, in the case of the adversarial k-clique example, batched update canonicality will ensure that when a batch of updates creates an entire clique, Tesseract will perform exploration comparable to a static GPM system.

Batch Differential Output Processing

Section 8.2.1 describes how Tesseract uses differential output processing to output embeddings correctly across updates. In the context of batching, the pre-update version of an embedding removes *all* the added edges in this batch, and similarly the post-update version of the embedding removes *all* the deleted edges.

Concurrent Execution

Supporting GPM applications for high-throughput evolving graphs requires scaling to multiple cores. We do so by taking advantage of batched update canonicality and using a staged execution model.

Batched update canonicality enables processing updates in batches efficiently by ensuring that the same pattern instance will not be found starting from different updates. This guarantee makes it possible to process different edge updates within the same batch concurrently without generating any duplicate instances, while requiring no co-ordination.

A multi-threaded worker executes Algorithm 5 to process updates. With staged execution, the worker first applies all updates to the graph, then invokes `pattern_update` on these updates, which may update the graph, and then runs EXPLORE for each of these updates. This approach avoids data races.

Distributed Mining

To support mining on multiple nodes in a cluster, Tesseract executes the staged execution model described above in parallel on multiple workers, one per node in a cluster. To do so, it replicates the graph structure by broadcasting all graph updates to every worker.

Work Partitioning Tesseract partitions work via graph partitioning, or assigning subsets of the graph to workers. Note that unlike in a static graph where partitioning is performed upfront, we need to partition arriving updates. Tesseract is agnostic to the graph partitioning scheme. By default, we use the partitioning scheme based on PowerGraph's balanced vertex cuts [55], which assigns edges to workers evenly, allowing vertices to span multiple workers. This approach distributes edges across workers uniformly, while also limiting the number of workers spanned by a vertex, and therefore provides good locality and work balance.

Batching and Synchronisation The updates in a batch (typically, 100K) are distributed across workers. Batch update canonicality ensures that workers do not generate the same pattern instances. Each worker operates on a single batch at a time, and thus the graph at each worker is single versioned.

Tesseract requires minimal synchronisation or data communication across workers. The embeddings cache is maintained locally at each worker. As a result, workers across the cluster perform caching based on the updates they process locally. If updates at the same worker have locality, which the vertex-cut partitioning aims at, this approach will be effective. The pattern pruner is distributed and merged across workers to further minimise computation, before EXPLORE is invoked in Algorithm 5.

8.3 Implementation

This section describes Tesseract's implementation and deployment. Tesseract is implemented in about 8k lines of C++ code for the GPM engine and 1k lines of Scala code for distributed execution and interfacing to Apache Spark. We interface Scala with C++ using Java Abstracted Foreign Function Layer [13].

Tesseract leverages Spark Structured Streaming [137] to provision nodes and provide an execution environment for running our GPM engine. On the ingest side, we sanitise and deduplicate graph updates and route the updates using streaming primitives such as `map`, `filter`, and `keyBy`.

Fault tolerance is essential for high-throughput dynamic GPM systems that may execute for long periods of time. Tesseract has state in the graph store, the embeddings cache, the pattern pruner, as well as soft state in the workers. The graph store and pattern pruner is replicated across all workers, and thus can be recovered by failed workers. The embeddings cache can be lost without affecting correctness. We rely on the streaming engine to handle worker failures and work redistribution, and use Apache Kafka [69] to provide exactly-once semantics for

graph updates and outgoing pattern instances.

8.4 Evaluation

We evaluate Tesseract and compare it to existing GPM systems on various input graphs and algorithms. Our evaluation sets out to answer the following questions:

1. How efficient is Tesseract compared to existing GPM systems that support dynamic graphs? (8.4.2)
2. How does Tesseract compare to static GPM systems? In particular, is there a cost to supporting updates? (8.4.3)
3. Does the pattern pruner improve performance? (8.4.4)
4. Can Tesseract process large graphs? (8.4.5)
5. How well does Tesseract scale on multiple machines, and are there factors in the design that limit scalability? (8.4.6)

8.4.1 Experimental Setup

We use the following three common mining applications described in 2.2.2: k-clique enumeration (k-C), Graph keyword search (k-GKS) and Motif Coundting (MC).

k-C is implemented as per Algorithm 6, but we fix the number of vertices in the clique to k. k-GKS is also implemented as per Algorithm 6, and an example of 3-GKS is shown in Figure 8.1. k-MC is implemented by matching every subgraph produced during exploration (i.e. `match` return true and `filter` accepts any pattern of size $\leq k$). We process the output pattern instances to check for isomorphism to a motif and increment the count in OUTPUT.

Note that GKS and MC are *general* GPM algorithms since we are not looking for a specific pattern, but considering all possible patterns of a given size. As a result, neither GKS nor MC can be encoded as a single subgraph query because each isomorphic pattern requires a different query.

Datasets Table 8.1 lists the graph datasets we used in the experiments. We use graphs with various sizes and characteristics representing different real-world use cases. While the size of some datasets is small, they contain a large number of pattern instances. For example, LiveJournal has ~246 billion 5-cliques, and MiCo has ~109 billion 4-motifs.

For k-GKS, we assign labels to nodes randomly, uniformly across all k so that 1/8th of the nodes are labeled. We simulate a dynamic graph by loading and applying a shuffled subset of the edges (and associated vertices) of a static graph iteratively until the entire graph is constructed. We simulate deletions in a similar way by deleting a shuffled subset of edges already present

Dataset	Vertices	Edges
MiCo [49]	100K	1M
LiveJournal (LJ) [1]	4.8M	68.9M
Twitter (TW) [77]	41.5M	1.7B
UK-2007 (UK) [27, 28]	106M	3.7B

Table 8.1 – Datasets.

in the graph. Unless otherwise specified, we report the time it takes to construct the entire graph from scratch.

Hardware and Configuration We evaluate Tesseract on a 28-core machine (2 Xeon E5-2690), equipped with 1TB of main memory (M1). For scale-out experiments, we use 8 16-core machines (2 Xeon E5-2630), each equipped with 128 GB of DDR3 ECC main memory (M2). The M1 and M2 machines have two 500GB SSDs. GPM applications run alone on the machines and so have access to the full resources. On each machine, we use as many threads as cores. Unless otherwise specified, we run with a batch size of 100k graph updates.

8.4.2 Performance on Evolving Graphs

Comparison with BigJoin

Since Tesseract is the first general-purpose GPM system for evolving graphs, there is a lack of candidates for comparison. We therefore compare with BigJoin [20] a recent state-of-the-art dynamic subgraph query system based on Timely Dataflow.

Table 8.2 compares the runtime of various algorithms for BigJoin and Tesseract on the LJ dataset. Since BigJoin does not support undirected graphs, we compare both systems on LJ directed. We use BigJoin's existing implementation of k-C, and implement optimised versions of k-CL and 3-GKS. Since BigJoin cannot express a general query like 3-GKS, we have to issue all possible combinations as separate queries, running the algorithm multiple times. We report the runtime of one such query for BigJoin. We do not report results for MC as the algorithm would require one query per motif.

Sys.	4-C	5-C	3-CL	4-CL	5-CL	3-GKS
BJ	275s	4h50m	19s	303s	4h43m	†2h14m
TS	316s	4h51m	10s	53s	12m	1h50m

Table 8.2 – BigJoin-Delta(BJ) and Tesseract(TS) runtime for different algorithms on the LJ dataset using a single M1 machine. For BigJoin-Delta performance for 3-GKS (†), we show the running time of only one possible query.

Tesseract's performance is at par with BigJoin for 4-C and 5-C. For k-CL, we find that the larger the pattern, the better Tesseract fares in comparison to BigJoin. Tesseract leverages its pattern pruner to filter intermediate embeddings that do not have distinct labels, significantly reducing the search space. In contrast, BigJoin must materialize all cliques before it can check the validity of attached labels, and so its performance remains similar to k-C. Finally, for 3-GKS, Tesseract performs faster than BigJoin, even though it mines all matching pattern instances while BigJoin executes one possible query, demonstrating the effectiveness of our system.

Overall Performance

Table 8.3 shows Tesseract's runtime on the MiCo and LJ datasets for different algorithm configurations on a single M1 machine.

Dataset	3-C	4-C	5-C	3-CL	4-CL	5-CL	3-MC	4-MC	3-GKS
MiCo	0.8s	15.0s	554.0s	0.2s	2.5s	50.4s	1.2s	332.7s	104.1s
LJ	15.4s	316.2s	2h45m	9.5s	53.1s	768.0s	59.6s	3h39m	1h54m

Table 8.3 – Tesseract runtime for different algorithms on MiCo and LJ dynamic datasets (100k batches) using a single M1 machine.

We observe that as the size of the pattern we are looking for increases, so does Tesseract's runtime, since the complexity of the algorithm is exponential in the size of the pattern. Different algorithms have different runtimes based on their characteristics. MC is slower than C and CL as expected, since the algorithm finds a superset of their patterns and performs additional work to identify motifs.

3-CL is approximately twice as fast as 3-C due to the pattern pruner. This filtering improves for 4- and 5-CL since the labels prune exploration more effectively by reducing the set of possible matches. This shows that Tesseract can significantly improve performance for patterns with higher selectivity.

Table 8.4 shows the ingest (number of graph updates processed per second) and output rate (number of pattern instances found per second) of Tesseract running 3-C on a single M1 machine for MiCo, LJ, and UK, a large graph.

Dataset	Ingest rate	Output rate
MiCo	7.2M/s	21.2M/s
LJ	4.0M/s	5.7M/s
UK	1.0M/s	891k/s

Table 8.4 – Ingest and Output rate for Tesseract on 3-C using M1.

These numbers show that Tesseract's graph store can ingest millions of updates per second. The output rate depends on the dataset. For instance, UK has a power-law graph structure with many edges that will result in significant exploration in the presence of updates across batches.

8.4.3 Performance Comparison with Static Systems

We now compare Tesseract with three state-of-the-art general purpose static GPM systems: Fractal [46], Arabesque [122], and RStream [129]. All three systems support undirected graphs, so we perform these experiments on undirected versions of our datasets, which we obtain by adding edges in the other direction.

The main goal of this experiment is to evaluate the overhead of supporting evolving graphs. To do so, we run Tesseract on a single batch, consisting of edge additions for the entire graph.

Note that this experiment involves more work for Tesseract, since it must perform canonicality checks and differential output processing for each update.

Table 8.5 compares the performance of Tesseract with Fractal, Arabesque, and RStream for 4-C and 4-MC for the LJ dataset with a static input graph.

Algorithm	Fractal	Arabesque	RStream	Tesseract
4-MC	130s	190s	>1h†	253s
4-C	9s	8s	294s	11s

Table 8.5 – Fractal, Arabesque, RStream, and Tesseract runtime for 4-MC and 4-C on the MiCo dataset using a single M1 machine. Tesseract runs on a single batch containing all graph edges. RStream did not finish 4-MC in less than an hour, so we killed the run (†).

Overall, these results shows that while Tesseract is designed for evolving graphs, its runtime for static exploration is comparable with that of existing static systems. For all runs, we observe that Tesseract's memory footprint remained capped at 40 GB (with an embeddings cache of 32 GB). Fractal's memory footprint remained very low since it does not perform any caching. Finally, both Arabesque and RStream used 100's of GB. RStream is a single machine out of core system, thus storing intermediary data on the SSD.

8.4.4 Domain- and Application-specific Pruning

We evaluate the performance benefits of Tesseract's pattern pruner. Table 8.6 shows the runtimes for 4-CL and 3-GKS on MiCo and LJ with the pattern pruner enabled and disabled.

Input	4-CL NoOpt	4-CL	3-GKS NoOpt	3-GKS
MiCo	10.8s	2.5s	111.9s	104.1s
LJ	166.4s	53.1s	2h21m	1h54m

Table 8.6 – 4-CL and 3-GKS performance on MiCo and LJ using a single M1 machine with and without optimisations.

The pattern pruner reduces runtime between 1.2X and 3X, depending on graph size and algorithm. The longer the algorithm, the more benefits the pattern pruner offers. For example, in 3-GKS, the runtime is cut by 25 minutes.

8.4.5 Mining Large Graphs

We evaluate Tesseract's performance when continuously mining larger graphs using TW and UK. Enumerating all pattern instances in such large graphs can take hours. In this case, as updates are received, we simply want to efficiently compute affected pattern instances. In this experiment, we first preload all but 10M edges of the graph, and then apply the remaining edges in batches of 100k.

Table 8.7 shows the average batch execution time, and the average number of pattern instances found per batch. This table shows that the update time per batch is significantly lower than the time required for recomputing all patterns on these large graphs, showing the benefits of Tesseract's update-driven approach. 4-CL is faster on the UK graph than on TW, even though

UK is a larger graph, because UK is less connected and the pattern pruner can prune many more edges that are not part of a clique. The average degree of separation in the UK graph is 15.4 vs 4.4 in Twitter [27, 28].

	4-CL		3-GKS	
Input	**Batch time**	**#instances**	**Batch time**	**#instances**
TW	229s	680K	771s	506M
UK	50s	13M	1200s	341M

Table 8.7 – Average batch processing time and average number of pattern instances found per batch on TW and UK for 4-CL and 3-GKS when applying 10M updates in batches of 100k on one M1 machine.

8.4.6 Scalability & Bottlenecks

Updates at Scale

We run Tesseract in our cluster of M2 nodes using the LJ data and two different algorithms to determine how well the system scales as we increase the number of nodes from 1 to 2, 4, and 8. Figure 8.2 shows the result of these experiments. Tesseract processes 4-C 5.1X faster for 3-MC, 5.5X faster for 4-C, and 5.6X faster for 3-GKS on 8 machines than on a single one.

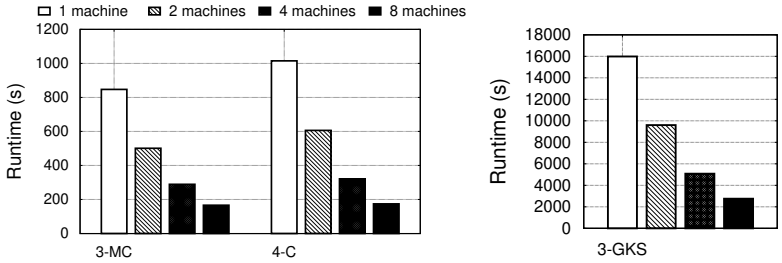


Figure 8.2 – Tesseract runtime for 3-MC and 4-C on LJ dataset with increasing number of M2 machines.

We do not achieve perfect scaling in the distributed case due to overheads introduced by full graph replication, since every worker applies all updates to its graph before processing the subset of its assigned updates. Also our current master-worker communication is simplistic, imposing latencies that can be addressed with a pipelined implementation. Finally, the interface between Spark and native binaries introduces a small slowdown. We plan to address this further in future work.

Batching

Figure 8.3 shows the sensitivity to batch size in our system for 4-C and 3-MC on the LJ dataset with an M1 machine. We vary the size of a batch between 1k and 10M, and each time report the runtime of the algorithm. Low batch sizes impact performance due to the overhead of applying them. Larger batch sizes of 100k are preferable with Tesseract, and very large sizes have negligible performance impact because we process updates in parallel without co-ordination.

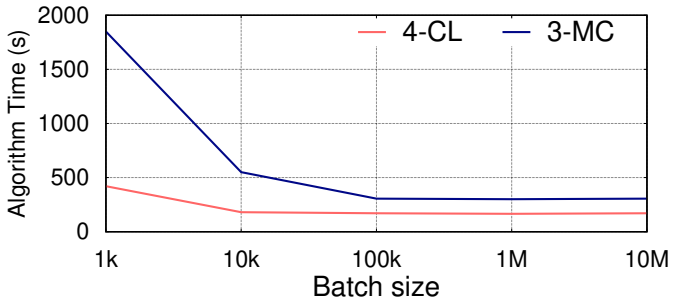


Figure 8.3 – Sensitivity to batch size for 4-CL and 3-MC running on LJ.

Work Balancing

A possible concern with using backtracking and depth-first expansion of embeddings is performance degradation as a result of load imbalance across worker threads or nodes due to different explorations taking different amounts of time to complete.

In our experiments, the highest load imbalance was for 4-C on LJ across 8 machines. In this experiment, the imbalance in completion time between threads was on average 2%, with one batch having a 4% imbalance, and the imbalance across machines due to imperfect partitioning was 14%.

8.5 Summary

We presented Tesseract, a fast, distributed system that supports interactive general GPM on dynamic graphs. Tesseract works well in the presence of updates because it uses a localized graph exploration algorithm that computes only changed pattern instances, and uses a novel canonicalization scheme to filter duplicates. Applications using Tesseract benefit from high throughput, scalability, and low memory overhead.

9 Related work

In this chapter we present papers and systems related to the work presented in this thesis, and were not presented in the previous chapters. We provide a brief analysis on how they compare to our work.

9.1 In-memory graph analytics

A large number of graph processing systems have been proposed [35, 36, 55, 59, 61, 64, 66, 78, 85, 95, 98, 106, 113, 114, 119, 132, 134, 138, 147, 149].

In Section 3.8 we presented the systems that inspired the work described in Chapter 3. In this section, we provide a brief overview of state of the art single machine and distributed in-memory systems.

Pregel [86] introduced the *think like a vertex* model where the programmer specifies the desired algorithm as a function to be applied on each vertex. The system then lifts computation to the whole graph using a succession of local operations. Pregel optimises network traffic by aggregating updates to the same vertex.**Giraph [36, 37]**, its open-source follow-on, demonstrated that this model can scale to very large capacity graphs.

Powergraph [55] and **PowerLyra [34]** propose a new approach to distributed graph placement that exploits the structure of real world graphs by defining operators on incident edges of vertices. PowerLyra improves on the initial partitioning of Powergraph by differentiating between high and low degree vertices. CUBE [139] presents a 3-D partitioning scheme beneficial to applications such as ALS, Gradient Descent or machine learning applications. The insight is that for algorithms where the vertex state is represented as a vector, this vector can also be partitioned in order to increase the parallelism.

Polymer [138] was the first single machine system designed to optimise for NUMA aware data placement. The graph is represented as adjacency lists that are divided among available NUMA-nodes. Polymer partitions the nodes of the graph so that each node is collocated with its out(or in) edges. To minimise compute imbalance due to the varying out (in) degree of

nodes, Polymer tries to partition the graph so that the number of edges per numa node is as close as possible.

Gemini [148] is a recent distributed graph processing system that supports single machine graph computation and, like Polymer, exploits NUMA-locality when placing data. The paper shows that even in a distributed setting it is important to optimise the computation on a single machine first. This minimises the need of scaling out but also improves overall performance. Gemini stores the graph as an adjacency list but adds work-stealing to account for load imbalance that might occur during computation.

The NUMA related experiments and partitioning schemes in Chapter 3 were inspired by work from these two systems.

Galois [98], **Ligra [119]**, and **GraphMat [121]** are single-machine systems focused on optimising the placement of graphs in memory to improve cache efficiency or memory locality on NUMA systems. These systems usually require complex pre-processing of the graph before processing.

Mizan [73] addresses the problem of graph partitioning and load balancing by migrating vertices between iterations in the hope of obtaining better load balance in the next iteration.

Gram [132] has shown how a graph with a trillion edges can be handled in the main memory of machines in a cluster.

Survey of in-memory graph processing techniques. Very few papers compare the benefits of different graph processing systems. **Satish et al. [115]** evaluate various single-machine and distributed systems and compare them to a hand-optimised baseline. The paper looks at complete systems rather than individual techniques. **Graphalytics. [32]** is a benchmark for graph processing platforms. We believe this to be a very good step in the right direction, due to the lack of a standardised benchmark for graph processing systems. The paper presents a set of benchmarks emphasising the robustness and scalability of different systems. The system offers developers to tune different parameters, but the user is the one providing the algorithm implementation and datasets to the system. Our work is orthogonal to Graphalytics, as it tests high level techniques and optimisations, regardless of how they are implemented in different systems.

9.2 Out of core graph processing

Graphene [57] uses a grid representation but changes cell sizes in order to balance the number of edges per cell. The system is designed for SSDs, and their main contribution is fine grained I/O management. The blocks on the storage device used by the application are bitmapped. I/O requests are handled by a background I/O thread that merges requests and sets appropriate bits in the bitmap. This way blocks are fetched only once. As with Flashgraph, this causes overheads. To improve the performance of traversal algorithms, **Lee et al. [80]** present a static

caching layer, on top of Flashgraph and Graphene.
Clip [18] uses asynchronous computation and single-threaded algorithms to minimise the I/O and the number of iterations of the algorithm. Our study has not evaluated asynchrony.
Lin et al. [82] show that for some workloads, memory mapping the graph inputs on SSDs can lead to a better performance than using traditional approaches. They do not include the pre-processing overheads, and contrary to their findings, our analysis in Chapter 5 shows that, it is not sufficient to just mmap the input without performing I/O related optimisations (sorting the active work queue for BFS, compressing the grid etc).
We are not aware of an analysis related to out of core graph processing which benchmarks the transformation cost. **Zhang et al. [140]** compare different systems, including two out of core systems, while varying architecture configuration. The main goal of the analysis is to determine the point when it is feasible to scale out rather than scale up. However, it is not clear what techniques, independent of architecture configuration, make a system outperform others.

Nilakant et al. [99] optimise the running time of graph processing on SSDs by prefetching data in the background to hide I/O latency. **Xu et al. [135]** analyse the impact NVMes have on database workloads. The paper points towards the fact that being I/O bound on NVMes is hard. **Huang et al. [62]** identify the redundancy of operation across multiple layers in the kernel I/O subsystem as a bottleneck in achieving better NVMe bandwidths. They implement a new layer between CPU and the NVMe, removing this redundancy.

9.3 Dynamic graph analytics

For completeness, it should be pointed out that some of the above systems support graph structure mutations. For instance, this is the case for Pregel and GraphChi. We opted to list them in the previous sections as their main target is not incremental computation. When run with graph updates, these systems behave similarly to the other incremental graph processing systems discussed below.

Many general-purpose incremental processing systems have been proposed over the past decade. These systems often introduce new primitives to existing batching or streaming frameworks in order to reuse previous output data and prior operator state. For example, **Haloop [31]** is a modified version of Hadoop which efficiently executes iterative MapReduce programs. **DryadInc [108]** adds incremental computation facilities on top of Dryad. **Mahout [102]** is a machine learning framework built on top of Hadoop which implements iterative model fitting. **Percolator [105]** uses observers to track data changes and trigger modification of other related data. Graph algorithms tend to be harder to express in these frameworks and they may miss on optimisation opportunities that a graph-only system can leverage. A recent line of research is to provide graph-specific abstractions on top of general-purpose data-parallel frameworks. **GraphX [56, 134]** and **iGraph [66]** are built on top of Apache Spark [137] and provide the programmer with high-level abstractions to perform interactive and incremental computation.

Naiad [91, 95] implements *timely dataflow* and, among others, supports incremental computation on large graphs. The key idea behind Naiad is to track dependencies between vertices. Naiad recomputes values only on the portion of the graph that depends on updated vertices. We reuse that idea to track dependencies for monotonic algorithms. Naiad separates computation in different epochs (batches of updates), which are serialised. As a consequence a trade-off has to be made between latency (small batches) and throughput (large batches).

Other graph engines that support dynamic graphs rely on batching updates and periodically re-running an incremental graph algorithm on the new version of the graph. **Kineograph [35]** was one of the first works to tackle dynamic graphs. Kineograph's key idea is to rely on atomic snapshots of the graphs. Every 10 seconds, Kineograph generates a snapshot of the graph, applies new structural updates, and runs an incremental analytics algorithm on it. The authors show that they can reflect the effect of updates in a few minutes on large graphs (2.5 minutes of latency to compute a page rank on the Twitter graph). **Chronos [59]** focuses on time-evolving graph snapshots and optimises their in-memory layout for locality and performance.

Stinger [48] is a data structure designed for processing evolving graphs on super-computers. Edges of a vertex are divided into blocks, and each edge maintains a timestamp for each update on it. Vertices point to the first bucket containing their edges.

GraphIn [116] is designed to compute on time- evolving graphs in batches of updates. The authors report high throughput numbers on small graphs (9 million updates / second on RMAT20 running WCC), but these numbers drop quickly as the graph size increases (2 million updates / second on RMAT22).

GraphOne [76] is also a batching system, but provides persistency by storing a log of changes on external storage. The system stores the graph in both, adjacency and edge list representations, in-memory. If the machine fails, the graph is reconstructed from the log. Updates are buffered and appended to the edge array immediately.The adjacency list view is periodically updated to reflect the updates, and the system maintains multiple snapshots of the graph.

Kickstrater [126] and **Graphbolt [88]** appeared after the work presented in Chapter 7.

Kickstarter only supports updates for monotonically converging algorithms, such as BFS and SSSP. It is an in-memory distributed system, that relies on dependency tracking for correctness. The system tracks the neighbour that is responsible for the value of a vertex, an equivalent of the father-son relationship in Snowy. As in Snowy, in the presence of deletions, the state of vertices needs to be corrected. Instead of cleaning, they reset the values based on a dependency tree. The tree is maintained during algorithm execution with the goal to avoid resetting the entire graph, a worst case scenario in Snowy. However, maintaining this tree creates a significant storage overhead. Kickstarter processes orders of magnitude smaller graphs than Snowy on a cluster of 16 machines. Since they batch updates, their latencies are also significantly higher than those in Snowy. For comparison, the worst case latency for Snowy when running BFS on the Twitter graph with 10% of deletions is 180ms, compared to

1.5s in Kickstarter. With 30% deletions, an unlikely scenario, Snowy has a worst-case latency of 8s, compared to the 5s reported by Kickstarter.
Snowy can easily be extended to implement the dependency tree, and using fast NVMe devices can be one way of decreasing the memory footprint.

Graphbolt is a single machine in-memory batching system for running always converging algorithms on evolving graphs. Graphbolt offers stronger consistency guarantees than Snowy, and batches updates before applying them. They report sub-second latencies to reflect the changes within one batch. This approach is feasible if correctness is a strict requirement, albeit at a higher memory footprint and without the support for traversal, monotonic algorithms.

9.4 Graph pattern mining

State-of-the-art general, GPM systems such as **Arabesque [122]** and **RStream [129]** use graph-wide exploration and enumerate *all* the pattern instances in the graph at the same time. In Arabesque, this approach enables parallelism via BSP-style phased execution, with embeddings being built incrementally in each phase, by adding one vertex or one edge at a time. In RStream, it enables storing and streaming embeddings from disk in a sequential manner. While, these systems work well for static graphs, they are not designed for graph updates, as discussed in Section 8.1.

Fractal [46] is a static GPM system which uses a depth-first search approach to enumerate embeddings. Fractal supports a graph reduction mechanism where developers can use domain-specific knowledge to produce a reduced graph, by pruning vertices and edges, therefore reducing memory footprint and enumeration costs. Tesseract's pattern pruner works in a similar fashion, but is designed to operate in a dynamic context, where the graph cannot necessarily be simplified without missing pattern instances formed as a result of graph updates.

Several **GPM systems [20, 75, 79]** use relational methods for supporting subgraph queries by expressing patterns as a relation query over the graph edges, and generate pattern instances by joining the edge table.

BigJoin [20] performs subgraph queries over static graphs using the GenericJoin algorithm [97] to provide worst-case optimal performance guarantees. It is implemented using the Timely Dataflow system [95], and is especially effective for purely structural queries that just involve joins, since the joins are run in parallel. However, this parallel execution makes it harder to filter embeddings efficiently, requiring the joined tuples to be generated before they can be filtered. In contrast, we can pre-filter a vertex by checking its neighbours' colors, reducing the search space significantly. **Delta-BigJoin [20]** performs subgraph queries on evolving graphs by combining incremental view maintenance methods with BigJoin. Tesseract filter-match model is more general and our canonicalisation and pattern pruner can reduce exploration greatly.

The complexity of GPM has led several researchers to develop specialised, domain-specific GPM systems, some of which can accommodate continuous queries [16, 72, 117, 133]. These systems do not generalise to other GPM problems. For example, **Wukong [130]** presents many optimisation techniques that can be used for RDF queries. Tesseract supports general-purpose GPM, and introduces a pattern pruner allowing developers to implement cutting-edge techniques developed for specific problems or application domains.

There has been much work on improving the performance of subgraph queries over evolving graphs by storing information about query vertices in the vertices or edges of the graph. Song et al [120] propose colouring edges to reduce the need for subgraph matching. **TurboFlux [75]** is a fast subgraph query system that employs a more sophisticated, graph-based representation for storing partial pattern instances in the graph. Our pattern pruner can be used to store intermediate query results, thus enabling such optimisations. Moreover, these algorithms use a single-threaded implementation, while Tesseract's pattern pruner enables a scale out implementation.

ASAP [65] is a fast, approximate subgraph query system that estimates the number of pattern matches in a graph, and it provides an error profile that allows trading accuracy for query runtime. It has good performance due to sampling, but it cannot be used to enumerate the pattern instances, and it has limited support for labels.

Our work is motivated by theoretical results on localisable and bounded GPM algorithms [51]. An algorithm is localisable if its cost is decided by the neighbours of the updated nodes instead of the entire graph. An algorithm is bounded relative to a batch algorithm, if its cost is determined by the number of updated nodes and changes to the affected area of the graph that is checked by the batch algorithm.

9.5 Graph analytics on specialised hardware

Qureshi et al. [112] discuss the use of NVM as main memory and evaluate several main memory organisations with DRAM and PCM, including *NVM-only* and *multi-level memory*. Their evaluation is based on simulation of a simple in-order processor model and memory that models only higher latency of PCM and not lower bandwidth. Further, their evaluation is limited to simple medium-sized application kernels. Our goal is to quantify the performance of NVM on modern CPUs with out-of-order execution and prefetch capabilities (§6.1), and with large-scale applications that are both latency-sensitive and bandwidth-intensive.

Lim et al. [81] study the use of slow memory in the context of shared, network-based (disaggregated) memory and conclude that a fast paging-based approach performs better than directly accessing slow memory. While their choice of target applications is key to their findings, their work also relies on a simple processor model and does not account for CPU's MLP and prefetch features (unlike our work).

Ferdman et al. [52] conduct a thorough study of many scale-out workloads using hardware

performance counters and conclude that these workloads are unable to exploit the CPU's MLP, leading to poor power efficiency. While similar in the use of counters, our work is different from theirs in several ways – (i) since our goal is to study the use of NVM, our workloads are all large in-memory applications, (ii) depending on the implementation, our workloads are able to achieve high MLP, and (iii) we conclude that, for future heterogeneous memory architectures with NVM, it is imperative (and hugely beneficial) for the application's performance to exploit MLP and hardware prefetching when accessing NVM, even if it requires re-designing these applications.

Qureshi et al. [111] study the impact of *MLP* on the effective cost of LLC misses in an application, and categorise those misses as costly isolated/dependent misses and cheaper parallel/independent misses. Their proposal to expose this information to cache replacement algorithms to reduce the number of isolated misses is even more relevant to the NVM architectures in this paper, owing to NVM's higher latencies.

NVM in the *hybrid* architecture has been explored in several contexts. Prior work has examined the use of NVM for both capacity and persistence, with emphasis on the necessary system software and libraries to provide applications with efficient access to NVM [40, 47, 71, 124, 125]. Lessons learned from our analysis are applicable to all of them.

Researchers have previously explored the use of data classification and intelligent data placement in hybrid memory systems, particularly in the context of HPC applications [17, 104]. We have applied this well-studied concept to large scale graph analytics applications and present our initial results that demonstrate the benefits of tiering with Graphmat in Chapter 6.

Not explored in this thesis, the use of **GPUs for graph processing** has been the subject of some recent works [43, 54, 92, 131, 145]. This approach could affect the relative magnitude of pre-processing vs. algorithm execution time, and thereby impact the conclusions from Chapter 3 for certain algorithms.

10 Conclusions and future work

In this thesis, we presented different systems to mitigate the bottlenecks of graph processing applications. The bottlenecks differ depending on the underlying hardware, and graph characteristics. The presented systems address the differences in system design depending on these characteristics.

We have shown that DRAM is still the fastest medium to process graphs from. However, many optimisations that improve in-memory graph computation rely on extensive pre-processing. Without optimised and properly targeted pre-processing, the system can end up improving only some algorithms while heavily penalising a different subset of algorithms. It is thus important to understand the bottlenecks of different algorithms and adjust the optimisations accordingly.

We explored the opportunities for bridging the gap between in-memory and out-of-core graph processing, offered by emerging non-volatile technologies. Fast PCIe NVMe devices support efficient execution of algorithms designed for in-memory computation, when minor optimisations, to increase I/O locality, are applied. This provides the potential for a system to dynamically adapt to the available DRAM, scaling-up automatically when the memory pressure is too high.

Optimus, presented in Chapter 5, already supports in-memory computation by memory mapping the input, which stays cached if there is enough DRAM. However, the optimisations to achieve I/O locality are an unnecessary slowdown when everything fits in DRAM. We plan to extend the work to automatically disable these optimisations when the ratio of available DRAM and input size grows beyond a certain threshold.
Integrating Optimus and the in-memory optimisations presented in Chapter 3 into one system would lead to a system that seamlessly scales across the storage stack. In fact, the work in the thesis can be a motivation to build a system that automatically selects optimisations, depending on the workload and execution environment. Such a system could also feed back to the end user the trade-offs between different runtime configurations.

In the Cloud such a system can take into account the cost of different configurations. For example, one of the many benefits achieved by computing from storage is the opportunity to leverage more storage devices at once by scaling up into the Cloud. The cost benefits are significant when using state-of-the-art out-of-core engines in the cloud, providing a much cheaper option for processing larger graphs compared to expensive instances with a lot of DRAM. The Cloud is a unique environment, and many optimisations that improve the processing time in the Cloud did not have the same effect within a local rack. This is in line with our conclusion that optimisations have to be carefully adjusted to the workload and underlying resources.
As NVMe devices are now offered in the Cloud, there is a new opportunity to further improve the performance of graph processing in the Cloud, by combining the optimisations we implemented in Optimus, with the Cloud-specific optimisations presented in Chapter 4. Potentially, using NVMes in the cloud would rely more on network provisioning than compression, as the network was already a bottleneck with commodity SSDs.

Optimising static graph processing is beneficial for many offline analytics. But we believe that fast processing of evolving graphs is more aligned with the dynamic nature of todays data.
While the two systems presented in Chapters 7 and 8 support updates with low-latency, both systems are in-memory systems, limited by the available memory. Inspired by GraphOne, a recent single-machine, persistent, evolving graph store [76], we see the opportunity to leverage fast NVMe devices in this design.
More specifically, we would like to design a distributed persistent graph store, that is application oblivious. Such a store would allow application developers to focus on the algorithm and application, without the need to worry about updating the underlying graph data structure.

In conclusion, this thesis fundamentally argues for a wholistic approach to system design. It is important to understand the workload and use-cases before optimising the system. Many state-of-the-art devices are under-utilised because the focus of the system is not on the *efficient* use of the underlying hardware. We have already seen a few papers [94, 142] motivated by the work presented in Chapter 3. The works present optimisations that adapt to algorithms leveraging input from developers, and suggest online optimisations as an alternative to an expensive pre-processing step that dominates the end-to-end time. It is our hope that the work done in the thesis will be beneficial for both, developers and users, of large-scale graph processing systems.

Bibliography

[1] http://snap.stanford.edu/data/soc-LiveJournal1.html.

[2] http://dimacs.rutgers.edu/Challenges/.

[3] http://zlib.net/.

[4] https://code.google.com/p/snappy/.

[5] Intel and micron produce breakthrough memory technology.

[6] Introducing the Graph 500 - Cray User Group. https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf, 2010.

[7] Crossbar Resistive Memory: The Future Technology for NAND Flash. http://www.crossbar-inc.com/assets/img/media/Crossbar-RRAM-Technology-Whitepaper-080413.pdf, 2013.

[8] Intel Xeon Processor E5 v2 Product Family (Vol 2). http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-2.pdf, 2013.

[9] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html, 2014.

[10] https://www.windowsazure.com, 2014.

[11] Intel Xeon Phi (Knights Landing) Architectural Overview. http://www8.hp.com/hpnext/posts/discover-day-two-future-now-machine-hp#.U9MZNPldWSo, 2014.

[12] Intel64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[13] https://github.com/jnr/jnr-ffi, 2019.

[14] FIO , 2018. http://freecode.com/projects/fio.

[15] IDC . Executive summary data growth, business opportunities, and the it imperatives. https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[16] ABDELHAMID, E., ABDELAZIZ, I., KALNIS, P., KHAYYAT, Z., AND JAMOUR, F. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE Press, p. 61.

[17] AGARWAL, N., NELLANS, D., STEPHENSON, M., O'CONNOR, M., AND KECKLER, S. W. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15.

[18] AI, Z., ZHANG, M., WU, Y., QIAN, X., CHEN, K., AND ZHENG, W. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 125–137.

[19] ALON, N., DAO, P., HAJIRASOULIHA, I., HORMOZDIARI, F., AND SAHINALP, S. C. Biomolecular network motif counting and discovery by color coding. *Bioinformatics 24*, 13 (2008), i241–i249.

[20] AMMAR, K., MCSHERRY, F., SALIHOGLU, S., AND JOGLEKAR, M. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment 11*, 6 (2018), 691–704.

[21] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15.

[22] BACKSTROM, L., BOLDI, P., ROSA, M., UGANDER, J., AND VIGNA, S. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference* (New York, NY, USA, 2012), WebSci '12, ACM, pp. 33–42.

[23] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 12:1–12:10.

[24] BEAMER, S., ASANOVIC, K., PATTERSON, D. A., BEAMER, S., AND PATTERSON, D. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Tech. rep., 2006.

[25] BECCHETTI, L., BOLDI, P., CASTILLO, C., AND GIONIS, A. Efficient algorithms for large-scale local triangle counting. *ACM Trans. Knowl. Discov. Data 4*, 3 (Oct. 2010), 13:1–13:28.

[26] BI, F., CHANG, L., LIN, X., QIN, L., AND ZHANG, W. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 1199–1214.

[27] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), ACM Press.

[28] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[29] BRIN, S., AND PAGE, L. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7* (1998), WWW7.

[30] BRON, C., AND KERBOSCH, J. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM 16*, 9 (1973), 575–577.

[31] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 285–296.

[32] CAPOTĂ, M., HEGEMAN, T., IOSUP, A., PRAT-PÉREZ, A., ERLING, O., AND BONCZ, P. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15* (New York, NY, USA, 2015), GRADES'15, ACM, pp. 7:1–7:6.

[33] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining* (2004), SIAM.

[34] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 15.

[35] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the ACM European conference on Computer Systems* (2012), ACM, pp. 85–98.

[36] CHING, A. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).

[37] CHING, A., EDUNOV, S., KABILJO, M., LOGOTHETIS, D., AND MUTHUKRISHNAN, S. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1804–1815.

[38] CHO, Y.-R., AND ZHANG, A. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on information technology in biomedicine 14*, 1 (2010), 30–36.

[39] CHOU, Y., FAHS, B., AND ABRAHAM, S. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (2004), ISCA '04.

[40] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[41] COOK, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (1971), ACM, pp. 151–158.

[42] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 381–394.

[43] DAVIDSON, A., BAXTER, S., GARLAND, M., AND OWENS, J. D. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 349–359.

[44] DEBRABANT, J., ARULRAJ, J., PAVLO, A., STONEBRAKER, M., ZDONIK, S., AND DULLOOR, S. A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS@VLDB* (2014).

[45] DEMENTIEV, R., KETTNER, L., AND SANDERS, P. Stxxl: Standard template library for xxl data sets. In *Algorithms – ESA 2005* (Berlin, Heidelberg, 2005), G. S. Brodal and S. Leonardi, Eds., Springer Berlin Heidelberg, pp. 640–651.

[46] DIAS, V., TEIXEIRA, C. H., GUEDES, D., MEIRA, W., AND PARTHASARATHY, S. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data* (2019), ACM, pp. 1357–1374.

[47] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14.

[48] EDIGER, D., MCCOLL, R., RIEDY, J., AND BADER, D. Stinger: High performance data structure for streaming graphs. pp. 1–5.

[49] ELSEIDY, M., ABDELHAMID, E., SKIADOPOULOS, S., AND KALNIS, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment 7*, 7 (2014), 517–528.

[50] ELYASI, N., CHOI, C., AND SIVASUBRAMANIAM, A. Large-scale graph processing on emerging storage devices. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 309–316.

[51] FAN, W., HU, C., AND TIAN, C. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 155–169.

[52] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII.

[53] FLAKE, G. W., LAWRENCE, S., GILES, C. L., AND COETZEE, F. M. Self-organization and identification of web communities. *Computer*, 3 (2002), 66–71.

[54] FU, Z., PERSONICK, M., AND THOMPSON, B. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2014), GRADES'14, ACM, pp. 2:1–2:6.

[55] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[56] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)* (2014), pp. 599–613.

[57] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 81–97.

[58] GUPTA, P., SATULURI, V., GREWAL, A., GURUMURTHY, S., ZHABIUK, V., LI, Q., AND LIN, J. Real-time twitter recommendation: online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment 7*, 13 (2014), 1379–1380.

[59] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[60] HAN, W.-S., LEE, J., AND LEE, J.-H. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 337–348.

[61] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 349–362.

[62] HUANG, J., BADAM, A., QURESHI, M. K., AND SCHWAN, K. Unified address translation for memory-mapped ssds with flashmap. *SIGARCH Comput. Archit. News 43*, 3 (June 2015), 580–591.

[63] INOKUCHI, A., WASHIO, T., AND MOTODA, H. An apriori-based algorithm for mining frequent substructures from graph data. In *European conference on principles of data mining and knowledge discovery* (2000), Springer, pp. 13–23.

[64] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 59–72.

[65] IYER, A. P., LIU, Z., JIN, X., VENKATARAMAN, S., BRAVERMAN, V., AND STOICA, I. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 745–761.

[66] JU, W., LI, J., YU, W., AND ZHANG, R. igraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* (2016), 1–15.

[67] JUN, S.-W., WRIGHT, A., ZHANG, S., XU, S., AND ARVIND. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2018), ISCA '18, IEEE Press, pp. 411–424.

[68] KACHOLIA, V., PANDIT, S., CHAKRABARTI, S., SUDARSHAN, S., DESAI, R., AND KARAMBELKAR, H. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 505–516.

[69] KAFKA, A. A high-throughput distributed messaging system. *URL: kafka. apache. org as of 5*, 1 (2014).

[70] KANG, U., CHAU, D., AND FALOUTSOS, C. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications* (2010).

[71] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[72] KARGAR, M., GOLAB, L., AND SZLICHTA, J. Effective keyword search in graphs. *arXiv preprint arXiv:1512.06395* (2015).

[73] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 169–182.

[74] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14.

[75] KIM, K., SEO, I., HAN, W.-S., LEE, J.-H., HONG, S., CHAFI, H., SHIN, H., AND JEONG, G. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data* (2018), ACM, pp. 411–426.

[76] KUMAR, P., AND HUANG, H. H. Graphone: A data store for real-time analytics on evolving graphs. In *USENIX Conference on File and Storage Technologies, (FAST)* (2019).

[77] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *Proceedings of the International conference on World Wide Web* (2010), ACM, pp. 591–600.

[78] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12.

[79] LAI, L., QIN, L., LIN, X., ZHANG, Y., CHANG, L., AND YANG, S. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment 10*, 3 (2016), 217–228.

[80] LEE, E., KIM, J., LIM, K., NOH, S. H., AND SEO, J. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 459–474.

[81] LIM, K., CHANG, J., MUDGE, T., RANGANATHAN, P., REINHARDT, S. K., AND WENISCH, T. F. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09.

[82] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., AND KANG, U. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)* (Oct 2014), pp. 159–164.

[83] LOH, G. H. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08.

[84] LOW, Y., GONZALEZ, J. E., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *Proceedings of Very Large Data Bases (PVLDB)* (8 2012).

[85] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 527–543.

[86] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data* (2010), ACM, pp. 135–146.

[87] MALICEVIC, J., LEPERS, B., AND ZWAENEPOEL, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 631–643.

[88] MARIAPPAN, M., AND VORA, K. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, ACM, pp. 25:1–25:16.

[89] MARSAGLIA, G., ET AL. Xorshift rngs. *Journal of Statistical Software 8*, 14 (2003), 1–6.

[90] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[91] MCSHERRY, F., MURRAY, D. G., ISAACS, R., AND ISARD, M. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[92] MERRILL, D., GARLAND, M., AND GRIMSHAW, A. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 117–128.

[93] MILO, R., SHEN-ORR, S., ITZKOVITZ, S., KASHTAN, N., CHKLOVSKII, D., AND ALON, U. Network motifs: simple building blocks of complex networks. *Science 298*, 5594 (2002), 824–827.

[94] MUKKARA, A., BECKMANN, N., ABEYDEERA, M., MA, X., AND SANCHEZ, D. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), IEEE, pp. 1–14.

[95] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[96] NEUMANN, T., AND WEIKUM, G. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal—The International Journal on Very Large Data Bases 19*, 1 (2010), 91–113.

[97] NGO, H. Q., RÉ, C., AND RUDRA, A. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record 42*, 4 (2014), 5–16.

[98] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[99] NILAKANT, K., DALIBARD, V., ROY, A., AND YONEKI, E. Prefedge: Ssd prefetcher for large-scale graph traversal. In *Proceedings of International Conference on Systems and Storage* (New York, NY, USA, 2014), SYSTOR 2014, ACM, pp. 4:1–4:12.

[100] OUKID, I., BOOSS, D., LEHNER, W., BUMBULIS, P., AND WILLHALM, T. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (2014), DaMoN '14.

[101] OUKID, I., LEHNER, W., THOMAS, K., WILLHALM, T., AND BUMBULIS, P. Instant Recovery for Main-Memory Databases. In *Proceedings of the Seventh Biennial Conference on Innovative Data Systems Research* (2015), CIDR '15.

[102] OWEN, S., ANIL, R., DUNNING, T., AND FRIEDMAN, E. Mahout in action.

[103] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[104] PAVLOVIC, M., PUZOVIC, N., AND ADRIAN, R. Data placement in hpc architectures with heterogeneous off-chip memory. In *Proceedings of the 31st IEEE International Conference on Computer Design* (2013), ICCD '13.

[105] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010), vol. 10, pp. 1–15.

[106] PEREZ, Y., SOSIČ, R., BANERJEE, A., PUTTAGUNTA, R., RAISON, M., SHAH, P., AND LESKOVEC, J. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1105–1110.

[107] PHUA, C., LEE, V., SMITH, K., AND GAYLER, R. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119* (2010).

[108] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. Dryadinc: Reusing work in large-scale computations. In *HotCloud* (2009).

[109] PRŽULJ, N., CORNEIL, D. G., AND JURISICA, I. Modeling interactome: scale-free or geometric? *Bioinformatics 20*, 18 (2004), 3508–3515.

[110] QURESHI, M. K., FRANCESCHINI, M. M., JAGMOHAN, A., AND LASTRAS, L. A. PreSET: Improving Performance of Phase Change Memories by Exploiting Asymmetry in Write Times. *SIGARCH Comput. Archit. News 40*, 3 (June 2012).

[111] QURESHI, M. K., LYNCH, D. N., MUTLU, O., AND PATT, Y. N. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (2006), ISCA '06.

[112] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09.

[113] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 410–424.

[114] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[115] SATISH, N., SUNDARAM, N., PATWARY, M. M. A., SEO, J., PARK, J., HASSAAN, M. A., SENGUPTA, S., YIN, Z., AND DUBEY, P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 979–990.

[116] SENGUPTA, D., SUNDARAM, N., ZHU, X., WILLKE, T. L., YOUNG, J., WOLF, M., AND SCHWAN, K. Graphin: An online high performance incremental graph processing framework.

[117] SHAHRIVARI, S., AND JALILI, S. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing 97*, 11 (2015), 1101–1120.

[118] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), ACM, pp. 625–636.

[119] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[120] SONG, C., GE, T., CHEN, C., AND WANG, J. Event pattern matching over graph streams. *Proceedings of the VLDB Endowment 8*, 4 (2014), 413–424.

[121] SUNDARAM, N., SATISH, N., PATWARY, M. M. A., DULLOOR, S. R., ANDERSON, M. J., VADLAMUDI, S. G., DAS, D., AND DUBEY, P. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1214–1225.

[122] TEIXEIRA, C. H., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 425–440.

[123] UGANDER, J., KARRER, B., BACKSTROM, L., AND MARLOW, C. The anatomy of the facebook social graph. *CoRR abs/1111.4503* (2011).

[124] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (2011), FAST'11.

[125] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[126] VORA, K., GUPTA, R., AND XU, G. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 237–251.

[127] WANG, C., AND PARTHASARATHY, S. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the 18th annual international conference on Supercomputing* (2004), ACM, pp. 31–40.

[128] WANG, H., AND AGGARWAL, C. C. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*. Springer, 2010, pp. 249–273.

[129] WANG, K., ZUO, Z., THORPE, J., NGUYEN, T. Q., AND XU, G. H. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 763–782.

[130] WANG, S., LOU, C., CHEN, R., AND CHEN, H. Fast and concurrent {RDF} queries using rdma-assisted {GPU} graph exploration. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 651–664.

[131] WANG, Y., DAVIDSON, A., PAN, Y., WU, Y., RIFFEL, A., AND OWENS, J. D. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2016), PPoPP '16, ACM, pp. 11:1–11:12.

[132] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[133] XIANG, J., GUO, C., AND ABOULNAGA, A. Scalable maximum clique computation using mapreduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013), IEEE, pp. 74–85.

[134] XIN, R. S., GONZALEZ, J. E., FRANKLIN, M. J., AND STOICA, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[135] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (New York, NY, USA, 2015), SYSTOR '15, ACM, pp. 6:1–6:11.

[136] ZAGHA, M., AND BLELLOCH, G. E. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), ACM, pp. 712–721.

[137] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12.

[138] ZHANG, K., CHEN, R., AND CHEN, H. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 183–193.

[139] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 285–300.

[140] ZHANG, Q., CHEN, H., YAN, D., CHENG, J., LOO, B. T., AND BANGALORE, P. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 40–51.

[141] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15.

[142] ZHANG, Y., YANG, M., BAGHDADI, R., KAMIL, S., SHUN, J., AND AMARASINGHE, S. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang. 2*, OOPSLA (Oct. 2018), 121:1–121:30.

[143] ZHENG, D., BURNS, R., AND SZALAY, A. S. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 69:1–69:12.

[144] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: Processing billion-node graphs on an array of commodity ssds. In

13th USENIX Conference on File and Storage Technologies (FAST 15) (Santa Clara, CA, 2015), USENIX Association, pp. 45–58.

[145] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst. 25*, 6 (June 2014), 1543–1552.

[146] ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. Large-scale parallel collaborative filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management* (Berlin, Heidelberg, 2008), AAIM '08, Springer-Verlag, pp. 337–348.

[147] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA* (2016).

[148] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 301–316.

[149] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 375–386.

Jasmina Malicevic

PhD Candidate in Computer Science
Operating systems laboratory, EPFL

- Date of birth: 16.06.1987
- E-Mail: jasmina.dustinac@gmail.com
- Phone: +41 78 640 11 68

REASEARCH INTERESTS

My research is focused on the impact irregular data structures, such as graphs, have on system design and choice of algorithms. I am also interested in storage implications to large-scale graph analytics. My most recent work is on designing an efficient out of core graph processing engine that leverages PCIe NVMe devices, and a graph pattern mining system to mine evolving graphs.

Education

2013 – 2019 **PHD in Computer science,** *EPFL, Switzerland*
Advisor: Prof. Willy Zwaenepoel
Relevant courses: Principles of Computer Systems, Big Data, Concurrent algorithms, Topics on Approximate Computing, Introduction to NLP

2010 – 2012 **Master in Computer Science**
Faculty of Electrical Engineering, University of Belgrade, Serbia

2005 – 2010 **Bachelor in Computer Science**
Faculty of Electrical Engineering, University of Belgrade, Serbia

2008 – 2009 **University Exchange stay**
Minnesota State University, Mankato, United States
Honors: Dean's list

Industry experience

Summer 2014 **Intel Labs,** Hillsboro, OR, US,*Graduate Research Intern, Systems and Storage Group*
Mentor: Dulloor Subramanya
Research on systems software for emerging non-volatile memories. By analyzing four different algorithms, we quantified the impacts of latency and bandwidth variations of NVM on four state of the art graph processing systems. We demonstrated that by static tiering of data among NVM and a small portion of DRAM, the performance was within 1.2x of DRAM-only performance

2009 – 2010 **Serbian Object Laboratories,** Belgrade, Serbia
Java and web based software development using executable UML principles

Honors and awards

- **Best Paper Award, USENIX ATC'17** for "Everything you always wanted to know about multicore graph processing but were afraid to ask"
- **Invitation to the MSR Cambridge summer school,** June 2015
- **Best student presentation,** EcoCloud Annual Event 2014, Lausanne, Switzerland
- **EPFL 1st year PhD Student Fellowship**, 2013
- **Student travel grants for** EuroSys'14, EuroSys'15, SOSP'15
- **Faculty of Electrical Engineering, Belgrade**: 4 year tuition waver for the top 10 on the entering exam
- **Dean's list,** Minnesota State University, Mankato for extraordinary achievements during the scholar year of 2009/2009

Publications

Rock you like a Hurricane: taming skew in large scale anaylitcs , Eurosys 2018
L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, W. Zwaenepoel

Everything you always wanted to know about multicore graph processing but were afraid to ask
USENIX Annual Technical Conference 2017
Jasmina Malicevic, Baptiste Lepers, Willy Zwaenepoel

Chaos: scale-out graph processing from secondary storage , SOSP 2015
Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel

Exploiting NVM in large-scale graph analytics, INFLOW 2015
J.Malicevic, S. Dulloor, N. Sundaram, N.Satish, J.Jackson, and W. Zwaenepoel.

Scale-up graph processing in the cloud: challenges and solutions. CloudDP 2014
Jasmina Malicevic, Amitabha Roy, Willy Zwaenepoel

Talks

- November 2018: Invited talk at KAUST, Saudi Arabia
- November 2017: Invited talk at the 7th INRIA- Technicolor workshop, Rennes, France
- USENIX ATC 2017: "Everything you always wanted to know about multicore graph processing but were afraid to ask"
- INFLOW 2015: "Exploiting NVM in large-scale graph analytics"
- 9th EuroSys Doctoral Workshop: "In-memory analytics of large scale evolving graphs"
- EcoCloud Annual Event 2014: "X-Scale: A Storage-Agnostic Graph Processing System"
- CloudDP 2014: "Scale-up graph processing in the cloud: challenges and solutions

Teaching

- Operating systems, EPFL (2014 – 2018)
  Undergraduate course ~80 -100 students. Designing practical exercises and projects on Linux Kernel internals.
- Real time systems, EPFL (2015 – 2018)
  Design of a project using Protothreads in ContikiOS.
- Programming 1 and Programming 2, Faculty of Electrical Engineering, Belgrade (2006 – 2010)
  Practical exercises in Java and C++

Languages

Serbian – native ; English – fluent; German – good; French – basic

Skills

Programming languages: C++/C, Java Operating systems: Linux

Personal

I am married and the mother of an energetic 22-month old. I enjoy pilates as often as I can. I like to hike and try to squeeze in time for a good book.