Performance, memory efficiency and programmability: the ambitious triptych of combining vertex-centricity with HPC

Ludovic A. R. Capelli

A thesis presented for the degree of
Doctor of Philosophy

THE UNIVERSITY
of EDINBURGH

School of Informatics
The University of Edinburgh
United Kingdom
2022

To my parents...

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisors Dr Brown and Dr Bull for their time, academic guidance and technical insight, which have proved valuable in helping focus efforts, considering potential new research directions and exploring lower-level aspects more thoroughly.

Also, it should be noted that the research reported in this thesis primarily exploits results collected from experiments run on *Cirrus*[1], a United Kingdom (UK) National Tier-2 High-Performance Computing (HPC) Service at the Edinburgh Parallel Computing Centre (EPCC) funded by the University of Edinburgh and the Engineering and Physical Sciences Research Council (EPSRC) under grant agreement EP/P020267/1, and *NEXTGenIO*[2], a system funded by the European Union's Horizon 2020 Research and Innovation Program under grant agreement number 671951, and supported by EPCC, at the University of Edinburgh.

I would like to highlight the importance of my internship at the National Institute of Informatics (NII) of Tokyo, Japan, in which I discovered the vertex-centric programming model, under the supervision of Prof Hu. This internship was supported by the International Internship Program of the NII of Tokyo, and the Japan Society for the Promotion of Science under grant agreement number 17H06099.

Finally, this research would not have been possible without EPSRC, which funded the first three years of this PhD, as part of the Centre for Doctoral Training (CDT) in Pervasive Parallelism award under grant agreement EP/L01503X/1, or as comprehensive without Huawei, as well as my mentor Mr Ye, which helped to support this research, via the Huawei Fellowship Program, enabling me to benefit from an additional year of funding and industrial insight.

[1] http://www.cirrus.ac.uk
[2] http://www.NEXTGenIO.eu

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- [1] Ludovic A. R. Capelli, Z. Hu, T. A. K. Zakian, "iPregel: A combiner-based in-memory shared-memory vertex-centric framework" in *Proceedings of the 47th International Conference on Parallel Processing Companion* (2018).
  DOI: 10.1145/3229710.3229719

- [2] Ludovic A. R. Capelli, Z. Hu, T. A. K. Zakian, N. Brown, J. M. Bull, "iPregel: Vertex-centric programmability vs memory efficiency and performance, why choose?" in *Journal of Parallel Computing 86* (2019) 45 – 56.
  DOI: 10.1016/j.parco.2019.04.005

- [3] Ludovic A. R. Capelli, N. Brown, J. M. Bull, "iPregel: Strategies to deal with an extreme form of irregularity in vertex-centric graph processing" in *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis* (2019).
  DOI: 10.1109/IA349570.2019.00013

- [4] Ludovic A. R. Capelli, N. Brown, J. M. Bull, "NVRAM as an enabler to new horizons in graph processing" in *Springer Nature Computer Science*, Volume 3, Article 385 (2022) 1–13.
  DOI: 10.1007/s42979-022-01317-4

Abstract

The field of graph processing has grown significantly due to the flexibility and wide applicability of the graph data structure. In the meantime, so has interest from the community in developing new approaches to graph processing applications. In 2010, Google introduced the vertex-centric programming model through their framework *Pregel*. This consists of expressing computation from the perspective of a vertex, whilst inter-vertex communications are achieved via data exchanges along incoming and outgoing edges, using the message-passing abstraction provided. *Pregel*'s high-level programming interface, designed around a set of simple functions, provides ease of programmability to the user. The aim is to enable the development of graph processing applications without requiring expertise in optimisation or parallel programming. Such challenges are instead abstracted from the user and offloaded to the underlying framework. However, fine-grained synchronisation, unpredictable memory access patterns and multiple sources of load imbalance make it difficult to implement the vertex centric model efficiently on high performance computing platforms without sacrificing programmability.

This research focuses on combining vertex-centric and High-Performance Computing (HPC), resulting in the development of a shared-memory framework, *iPregel*, which demonstrates that a performance and memory efficiency similar to that of non-vertex-centric approaches can be achieved while preserving the programmability benefits of vertex-centric. Non-volatile memory is then explored to extend single-node capabilities, during which multiple versions of *iPregel* are implemented to experiment with the various data movement strategies. Then, distributed memory parallelism is investigated to overcome the resource limitations of single node processing. A second framework named *DiP*, which ports applicable *iPregel*'s optimisations to distributed memory, prioritises performance to high scalability.

This research has resulted in a set of techniques and optimisations illustrated through a shared-memory framework *iPregel* and a distributed-memory framework *DiP*. The former closes a gap of several orders of magnitude in both performance and memory efficiency, even able to process a graph of 750 billion edges using non-volatile memory. The latter has proved that this competitiveness can also be scaled beyond a single node, enabling the processing of the largest graph generated in this research, comprising 1.6 trillion edges. Most importantly, both frameworks achieved these performance and capability gains whilst also preserving programmability, which is the cornerstone of the vertex-centric programming model. This research therefore demonstrates that by combining vertex-centricity and High-Performance Computing (HPC), it is possible to maintain performance, memory efficiency and programmability.

Acronyms

AMD	Advanced Micro Devices. 10
API	Application Programming Interface. 12
BSP	Bulk-Synchronous Parallel. 15
CAS	Compare-And-Swap. 157
CPU	Central Processing Unit. 1
DBLP	Database and Logic Programming Bibliography. 74
DCPMM	Data Centre Persistent Memory Module. 108
DIMM	Dual In-line Memory Module. 61
DOI	Digital Object Identifier. 5
DRAM	Dynamic Random Access Memory. 2
FPGA	Field Programmable Gate Arrays. 1
GPGPU	General Purpose Graphics Processing Unit. 11
GPU	Graphics Processing Unit. 1
GTEPS	Giga-Traversed Edges Per Second. 120
HDD	Hard Disk Drive. 11
HPC	High-Performance Computing. 2
IBMP	Interval-Based Message Processing. 4, 241
INBS	Intra-Node Buffer Sharing. 4, 241
LTS	Long-Term Support. 61
MPI	Message-Passing Interface. 12

NIC Network Interface Card. 1
NUMA Non-Uniform Memory Access. 9, 10
NVRAM Non-Volatile Random Access Memory. 4

OpenMP Open Multi-Processing. 12

PMDK Persistent Memory Development Kit. 111

RDMA Remote Direct Memory Accesses. 2
ReRAM Resistive Random Access Memory. 2
RMA Remote Memory Access. 195

SDK Software Development Kit. 192
SIMD Single Instruction Multiple Data. 11
SIMT Single Instruction Multiple Threads. 11
SM Streaming Multiprocessor. 11
SMT Simultaneous MultiThreading. 9
SNAP Stanford Network Analysis Project. 74
SPFC Slow Passing Fast Consuming. 37
SSD Solid State Drive. 11
SSSP Single-Source Shortest Paths. 20
STL Standard Template Library. 129

USA United States of America. 53

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are a uniquely flexible and generic data structure. From social networks to molecular structures, through road maps and neural networks, graphs are used to represent a wide range of datasets. Due to this flexibility, there has been a growing interest in graph processing applications: including clustering [6, 7] and traversal algorithms [8], pattern matching [9] and data analytics [10] to name a few. This growing interest has resulted in an increasing number of graph processing frameworks being developed. However, the writing of graph processing applications is a non-trivial task, especially in the case of large graphs where distributed-memory programming and high performance computing techniques play a major role. Users of such graph processing frameworks may be specialised in their own domain such as data science for example, but with little knowledge about high performance computing techniques.

The introduction of the vertex-centric programming model in 2010, delivered via the framework *Pregel* [11], addressed this issue by providing a highly-abstracted programming model enabling users to develop graph processing applications while all parallelism and low-level considerations are offloaded to the underlying framework. In vertex-centric, computation is expressed from a vertex point of view, often referred to as *thinking like a vertex*, and inter-vertex communications are achieved with messages using the message-passing abstraction provided. Due to its ease of use, the vertex-centric programming model has rapidly grown in popularity, with vertex-centric frameworks flourishing across all types of hardware; including Central Processing Units (CPUs) [11, 12], Graphics Processing Units (GPUs) [13–16], Field Programmable Gate Arrays (FPGAs) [17, 18] and Network Interface Cards (NICs) [19]. In addition to targeting such a variety of hardware, vertex-centric frameworks have also spanned across programming languages. Although most frameworks are implemented using imperative programming languages, vertex-centric frameworks have also been developed using functional programming languages

too [20–22]. Furthermore, multiple memory technologies are present in the landscape of vertex-centric solutions; from traditional Dynamic Random Access Memory (DRAM) [11, 12] to Resistive Random Access Memory (ReRAM) [23], through both classic disk storage [24,25] and flash storage [26]. Moreover, in addition to Google, who introduced vertex-centric and developed *Pregel*, several other major companies invested research in vertex-centric processing, such as Microsoft in 2015 [19] who explored the use of Remote Direct Memory Accesses (RDMA), Facebook in 2017 [27] by improving the distributed-memory framework *Giraph* to process a trillion-edge graph and Huawei [16, 28–30] between 2014 to the present day who investigated out-of-core computation and neural network applications. Put simply, vertex-centric graph processing is ubiquitous and an Intel patent from 2019 [31] provided an apt summary of the popularity and benefits:

> A user specifies a graph algorithm as "vertex programs" following vertex-centric programming abstraction. This abstraction is chosen as an example here due to its popularity. A vertex program does not expose hardware details, so users without hardware expertise (e.g., data scientists) can create it.

However, the vertex-centric abstractions which benefit programmability often do so at the cost of performance. This is a common trade-off in high performance computing, where users have a choice between programmability or performance, with technologies that deliver both being few and far between. Furthermore, vertex-centric frameworks have proved to have a high memory consumption, for instance *Giraph* and *Pregel+* require 264GB and 109GB of memory respectively to process PageRank over a graph with fewer than 2 billion edges, thus representing approximately 8GB of data.

Therefore, these two limitations, high memory usage and performance overhead, limit the utility of vertex-centric for processing larger graphs which, considering the size of graphs is growing significantly, is a major limitation. Improving the memory efficiency of vertex-centric frameworks will not only enable shared-memory frameworks to process larger graphs, but will also benefit distributed memory frameworks as one can reduce the minimum number of nodes required to process a graph. The current state of the art is to work around the high memory consumption by favouring distributed memory solutions, however this results in significant communications overhead. Moreover, *Ligra* has shown that shared-memory graph processing is a viable solution, especially considering that High-Performance Computing (HPC) nodes typically contain 256GB or more of DRAM. *Ligra* has become a major framework for shared-memory graph processing and the state-of-the-art in vertex-subset-centric programming (as defined in Subsection 3.2.1), but this sacrifices the clean vertex-centric abstraction that promotes programmability for performance, and an equally competitive vertex-centric counterpart is yet to be designed.

Developing and discovering techniques which improve vertex-centric performance and memory efficiency will therefore enable vertex-centric frameworks to much better exploit shared memory parallelism. Existing vertex-centric frameworks could also benefit from such techniques, as these could likely be applied without requiring extensive code re-writing on behalf of their users.

Nonetheless, developing such techniques is not trivial, for example, there are numerous challenges when designing an efficient vertex-centric framework, including multiple layers of load imbalance, fine-grain synchronisations and unpredictable memory accesses both in terms of quantity and locality. Furthermore, the highly abstracted interface provided by vertex-centric results in little information being exposed to the underlying framework responsible for parallelisation and optimisation. Given programmer productivity is the major reason that vertex-centric is so popular, optimisations that promote performance or memory efficiency cannot reduce programmability.

Numerous optimisation techniques beneficial to vertex-centric performance have been developed, such as selective scheduling [24], communication channels [21], receiver-side scatter [32–34], vertex-mirroring [12, 35], vertex inactivation [21], incrementalisation [36] and message prioritisation [37]. However, the majority of optimisation techniques impact the abstractions and impact programmability. For instance, several optimisations require the removal of a vertex-centric abstraction, such as the message-passing abstraction which provides a means to communicate by the exchange of messages. Because the vertex-centric programming model is a coherent and interconnected set of abstractions, removing one likely results in the degradation, or removal, of other vertex-centric abstractions. In certain cases, mutations to the original vertex-centric programming model were too consequential for the resulting framework to still be considered vertex-centric. Although an attempt at preserving all vertex-centric abstractions [38] is present in the literature, the performance and memory efficiency exhibited turn out to suffer from a penalty of orders of magnitude.

The work conducted in this research aims to determine whether such a trade-off is unavoidable by attempting to design techniques that preserve vertex-centric programmability whilst maximising performance and memory efficiency. The overarching research hypothesis in this thesis is therefore formulated as follows:

Hypothesis:

In the context of vertex-centric, programmability can be preserved during the design of optimisations improving performance or memory efficiency.

To verify this research hypothesis, three research directions have been identified:

1. The investigation of such optimisation techniques in the context of shared-memory architectures, which limit the complexity of the underlying system.

2. The exploration of hardware technologies to overcome shared-memory limitations.

3. An assessment of whether the benefits obtained in the area of shared memory can also benefit distributed-memory architectures based on a new, bespoke, buffer design, in the scope of low to medium node counts.

The novel contributions made in these research directions, and reported in this thesis, are summarised as follows:

- A modular architecture, as well as a broad set of techniques for maintaining a minimum memory footprint, while maximising its performance to reach that of other non-vertex-centric frameworks. This is demonstrated through the implementation of a shared-memory framework: *iPregel*.

- An exploration of Non-Volatile Random Access Memory (NVRAM) in the context of shared-memory vertex-centric processing, with experiments run on graphs with between 250 and 750 billion edges. During this exploration, multiple data placement and movement techniques were implemented to evaluate the approach that best suits the asymmetric performance of read-writes, combined with the two-level DRAM-NVRAM memory system.

- Bringing application techniques developed in *iPregel* to distributed-memory and the introduction of a buffer design that comprises several vertex-centric phases such as message wrapping or message dispatch, as well as reducing the algorithmic complexity of a full round of message generation from $O(n \times log(n))$ to $O(n)$. These techniques and buffer design are demonstrated through the *DiP* distributed-memory framework.

- Two techniques that improve the memory footprint of the *DiP* buffer design: an Intra-Node Buffer Sharing (INBS) approach relying on MPI-3 shared-memory and Interval-Based Message Processing (IBMP) that enables the *DiP* framework to arbitrarily adapt its size to a memory limit set by the user.

- The fact that none of the techniques and methods presented above results in a degradation of the vertex-centric programmability, thus maintaining the key benefit delivered by this approach.

1.1 Papers published in this research

Some of the material used in this thesis has been published in the following papers:

- The design of the *iPregel* framework, including the multi-version module selection and the set of techniques introduced in Chapter 4, was published in:
  [1] Ludovic A. R. Capelli, Z. Hu, T. A. K. Zakian, "iPregel: A combiner-based in-memory shared-memory vertex-centric framework" in *Proceedings of the 47th International Conference on Parallel Processing Companion* (2018).
  DOI: 10.1145/3229710.3229719

- The thorough evaluation of the *iPregel* framework against several other shared-memory frameworks, which also focusses on including programmability as an additional criterion to performance and memory footprint, presented in Chapter 4 was published in:
  [2] Ludovic A. R. Capelli, Z. Hu, T. A. K. Zakian, N. Brown, J. M. Bull, "iPregel: Vertex-centric programmability vs memory efficiency and performance, why choose?" in *Journal of Parallel Computing 86* (2019) 45 – 56.
  DOI: 10.1016/j.parco.2019.04.005

- The set of optimisation techniques presented in Chapter 5, which focus on addressing the multiple performance challenges in vertex-centric programming, were published in:
  [3] Ludovic A. R. Capelli, N. Brown, J. M. Bull, "iPregel: Strategies to deal with an extreme form of irregularity in vertex-centric graph processing" in *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis* (2019).
  DOI: 10.1109/IA349570.2019.00013

- The investigation into non-volatile memory technology reported in Chapter 6, which expands the processing ability of the *iPregel* framework to graphs reaching almost a trillion edges, was published in:
  [4] Ludovic A. R. Capelli, N. Brown, J. M. Bull, "NVRAM as an enabler to new horizons in graph processing" in *Springer Nature Computer Science*, Volume 3, Article 385 (2022) 1–13.
  DOI: 10.1007/s42979-022-01317-4

1.2 Structure of the thesis

The rest of this thesis is structured as follows:

Chapter 2 introduces the vertex-centric programming model, the graph theory and HPC terminologies used throughout this thesis.

Chapter 3 reviews the existing literature by surveying different optimisation techniques and methods currently used in vertex-centric programming.

Chapter 4 presents the modular design and techniques used in the development of the shared-memory framework *iPregel*.

Chapter 5 explores the challenges associated with fine-grained synchronisation, irregular memory accesses and load imbalance in more detail, and the techniques that have been developed to address these.

Chapter 6 analyses the suitability of non-volatile memory for vertex-centric workloads.

Chapter 7 presents the development of the distributed-memory framework *DiP*, the creation of its interface, its buffer design and the porting of techniques used in *iPregel*.

Chapter 8 highlights future work, and discusses whether the investigations conducted in this thesis verify the overarching research hypothesis.

Chapter 2

Background

This chapter introduces underlying concepts from graph theory and high performance computing which are necessary to the understanding of this thesis. The vertex-centric programming model is then presented, along with the benchmarks used throughout the evaluation of the research.

2.1 Graph theory

2.1.1 Structure

A *graph* is a data structure comprising *nodes*, also referred to as *vertices*, which are linked with *edges*. An edge is either *directed* or *undirected*, where a directed edge is unidirectional, from a source vertex to a destination vertex. By contrast, an undirected edge between two vertices is bidirectional; the equivalent of two directed edges, one edge from the first vertex to the second, and inversely for the second edge. A *directed graph* is made up of directed edges, whilst an *undirected graph* consists of undirected edges. Additionally, weights can be assigned to edges, resulting in *weighted edges* of a *weighted graph*, or otherwise *unweighted edges* of an *unweighted graph*.

2.1.2 Connectivity

A directed edge linking a source vertex v_{src} to a destination vertex v_{dst} is said to be an *outgoing edge* of v_{src} and an *incoming edge* of v_{dst}, abbreviated *out-edge* and *in-edge* respectively. Vertices connected by an edge are called *neighbours*, so, in this example, v_{src} is said to be an *incoming neighbour* of v_{dst}, abbreviated *in-neighbour*, and v_{dst} is said to be an *outgoing neighbour* of v_{src}, or *out-neighbour*. The list of neighbours of a

given vertex is known as an *adjacency list* in undirected graphs, while vertices in directed graphs comprise both *out-adjacency* and *in-adjacency* lists.

2.1.3 Reachability

A *path* contains the list of vertices traversed from given source vertex v_{src} to eventually reach a destination vertex v_{dst}, if possible. Two vertices are said to be *connected* if there exists a path between them, and a vertex v_{dst} is *reachable* from a vertex v_{src} if both are connected.

In an unweighted graph, the *length* of a path is the number of edges that must be traversed, whereas in a weighted graph it is the sum of the weights of the traversed edges. Among all possible paths that exist between two connected vertices, the shortest path is known as the *distance* between these two vertices.

2.1.4 Metrics

The two main characteristics of a graph are its *order* and *size*. The terms graph order represent the number of vertices in the graph, whilst the terms graph size represent the number of edges that it contains. In this thesis, these terms are denoted by $|V|$ and $|E|$ respectively.

The *average degree* is the ratio of a graph's size to its order, where the *degree* of a vertex is the number of edges shared by this vertex. In the case of directed edges, the *out-degree* of a vertex represents the number of outgoing edges from this vertex, while the *in-degree* represents the number of incoming edges to this vertex.

Another metric is known as *graph diameter*, which is the distance between the two most distant nodes. This can be used to estimate the maximum number of iterations needed for a message generated from a given source vertex to be propagated to any destination vertex.

2.2 High performance computing

2.2.1 Architectures

For decades, a correlation between process size and processor performance was observed. However, although technology continues to permit further shrinking of process size, down to 2nm as of 2022; a plateau in the processor's clock frequency has been observed, at approximately 4GHz since the early 2000s [39].

The reason for this decorrelation is twofold. Firstly, it is due to the increased thermal loss induced by a higher clock rate, requiring the upgrade of the cooling solutions applied; ultimately exceeding what can be sustainable, both practically and commercially. Secondly, the energy required to power a transistor was once thought to be proportional to its dimensions, known as the Dennard scaling [40]. As transistors shrank, so did their power consumption, which was expected. However, when lower than 90nm, current leakage became a significant factor and limited further benefits.

2.2.1.1 Multi-core architectures

To address the inability to continue scaling processor clock frequencies, a shift has been observed since early 2000s from faster processors to higher processor core counts. For example, multiprocessors commonly found in HPC nowadays typically contain between 16 and 64 processing cores. This is readily observed in the Top500 list [41], containing the 500 most powerful commercially available computer systems, therefore suggesting that this approach will likely remain the trend for the foreseeable future.

2.2.1.2 Multithreading

Multithreading refers to the ability of a processing core to support multiple threads of execution, referred to as *hardware threads*, concurrently. This technique maximises the utilisation of CPU hardware resources and overall throughput by avoiding certain execution units from going idle, in the event of cache misses for instance.

Simultaneous MultiThreading (SMT) is the ability of a processing core to dispatch instructions originating from more than a single hardware thread context [42], permitting to hide what could otherwise be stalls in the execution flow due to cache misses or branch mispredictions for instance. When combined with *superscalar processors*, which are processors able to execute multiple instructions per cycle, SMT enables the simultaneous execution of multiple instructions originating from multiple threads.

2.2.1.3 Non-uniform memory access

Modern CPUs contain their own memory controllers and dedicated memory channels that connect directly to DRAM. However, modern architectures may contain multiple CPU sockets, each of which contains a CPU connected to its local RAM.

Communication between sockets is ensured via an interconnect, which remotely accesses data held in another socket's local DRAM and sends it to the requesting processor, rather than the processor being able to directly access it. This results in extra latency and potentially bandwidth restrictions. This characterises Non-Uniform Memory Access

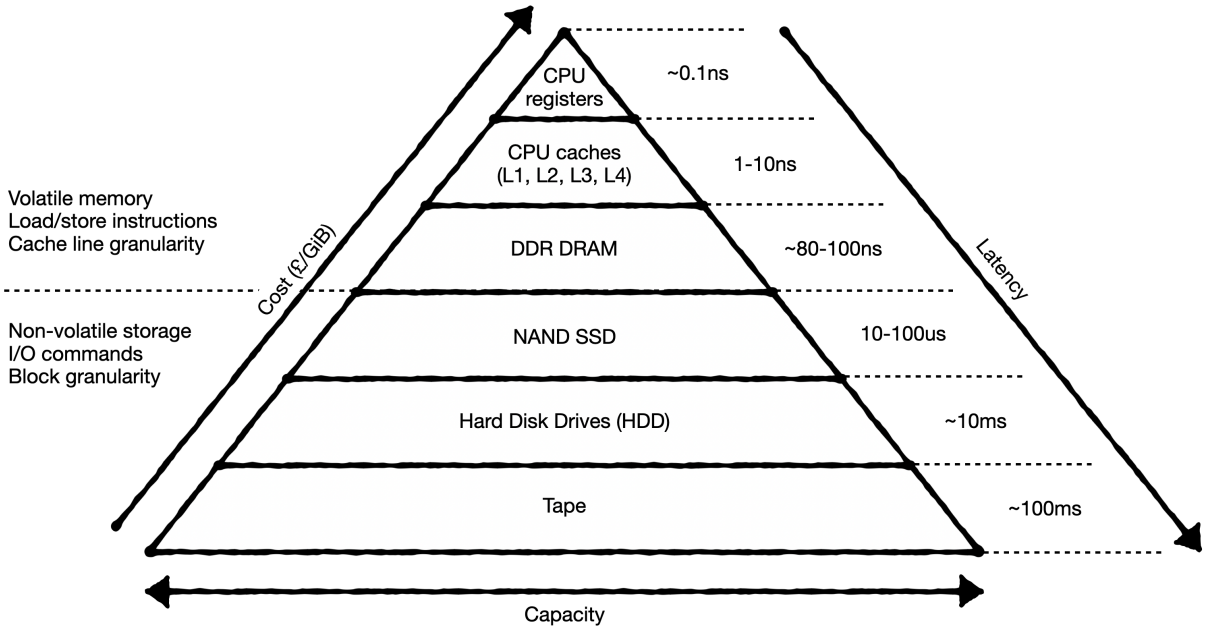

Figure 2.1: Memory storage hierarchy

(NUMA), and in this example, each socket / DRAM module forms what is referred to as a *NUMA-region*. It is common for modern HPC architectures to contain a NUMA region per processor, however there are also architectures such as Advanced Micro Devices (AMD) Rome in ARCHER2 that contain multiple NUMA regions per socket. Memory accesses originating from a NUMA region to data located in a different NUMA region are referred to as *cross-NUMA*.

The additional latency observed in data movement emphasises the importance of data placement, which is not always intuitive. For example, Linux, which runs all 500 of the Top500 list, applies a *first-touch policy* by default [43]. This specifies that the NUMA region from which the data is first accessed is determinant, not the one from which the corresponding allocation call was issued. Therefore, maximising performance requires a careful design of parallel data initialisation for instance to match the pattern of subsequent parallel accesses during execution. Similarly, software threads must be bound to a NUMA region, so that they are not moved to a different region during execution and memory accesses must then traverse the interconnect.

2.2.1.4 Non-volatile memory

Computers have access to several storage mediums, as shown in Figure 2.1 reproduced from [44], to serve from caches and main memory to file systems. *Volatile memory*, such as CPU registers, caches and DRAM; retains data only while the device is powered. Expensive, volatile memory is typically used in components where performance is sought.

Conversely, *non-volatile storage* does not lose data held when the device is no longer powered. This property makes non-volatile storage, such as Solid State Drives (SSDs), Hard Disk Drives (HDDs) and tape; suitable as secondary storage or long-term consistent storage.

Non-volatile memory stands as an intermediate layer between volatile memory and non-volatile storage, and its ability to provide load/store instructions and cache line granularity in addition to non-volatility allows it to act as memory or storage. With a latency typically under a microsecond, non-volatile memory remains considerably slower than its volatile counterpart. However, advances in technology have allowed non-volatile memory to alleviate, to some extent at-least, performance differences, making it a potential solution in increasing main memory at a competitive price. The use of this storage type in the context of vertex-centric is explored in detail in Chapter 6.

2.2.1.5 Graphics processing unit

Graphics processing units were originally developed for the generation of 3D graphics and contain multiprocessors, referred to as *Streaming Multiprocessor* (SM), comparable to large Single Instruction Multiple Data (SIMD) processors where a given instruction is issued to multiple data concurrently. SMs are rather characterised as Single Instruction Multiple Threads (SIMT), where cores execute instructions from threads in lock-step. Unlike SIMD processors, SIMT allows cores to access data locations that are not consecutive, as well as execute code with branches.

The large amount of raw computational power and high degree of parallelism available in GPU architectures attracted an increasing interest over the years towards processing other types of calculations. As technology progressed, GPUs gradually became more and more programmable, through what is known as General Purpose Graphics Processing Unit (GPGPU); allowing arbitrary code to be executed on SMs. The use of GPUs has proven successful in areas with highly computationally intensive workflows, such as certain scientific simulations and financial calculations.

However, GPUs remain devices embedding a distinct main memory, often referred to as *device memory* by contrast to *host memory* designating the computer's main memory. This implies that data must first be transferred to the GPU before processing, and results transferred back afterwards. However, the amount of memory available on the vast majority of GPUs is under 10GB, whereas that of HPC nodes varies between 128GB and 512GB. Therefore, processing large amounts of data can result in a high number of data transfers, putting pressure on a known limitation of GPUs. In addition, the load imbalance induced from irregular structures such as graphs may result in idle time for

large portions of a streaming multiprocessor, for instance when processing graph vertices with widely different numbers of neighbours. Based on these observations, the use of GPU architectures was not explored in this thesis.

2.2.2 Shared and distributed-memory parallelism

Processors have access to different levels of memory to store and load data. From L1 and L2 caches, which are typically local to a processor, up to L3 cache that is commonly shared with the entire multiprocessor, and main memory that is shared across all multiprocessors on the node.

Shared-memory parallelism is a form of parallelism that can be used by an application relying on multiple threads or processes, referred to as *workers* in the rest of this thesis, unless specified otherwise; that physically share the same memory space. In HPC, the dominant Application Programming Interface (API) allowing this type of parallelism is Open Multi-Processing [45] (OpenMP), which relies on the use of directives that are inserted as special comments into a serial source code. However, independently from the shared-memory programming solution, the use of shared-memory parallelism poses challenges around correctness, such as data-race conditions requiring synchronisation mechanisms. In addition, the mapping of threads and processes, determining which core executes which thread, as well as the binding, which specifies where threads can be remapped at runtime, play an important role in obtaining optimal performance.

Distributed-memory parallelism expands to processors that reside on different nodes, therefore not physically sharing memory. This form of parallelism allows applications to use multiple nodes by running multiple *distributed-memory workers* that can be mapped to processors residing on the same node, or different nodes. Therefore, distributed-memory parallelism increases the amount of computing power and memory resources available to an application. However, this ability comes at the expense of inter-node network communications. In HPC, the most widely used API for distributed-memory is Message-Passing Interface [46] (MPI); running multiple instances of a program, called MPI processes, on different processors, possibly residing on different nodes. Communications between MPI processes are achieved with explicit messages, using the routines provided by the API.

Although certain distributed-memory parallelism solutions such as MPI have gained the ability to leverage shared-memory parallelism by directly accessing the memory of other distributed-memory workers, they are most commonly combined with a shared-memory parallelism solution. In HPC, the resulting pair observed is often MPI and OpenMP, due to the highly optimised implementations available of their respective long-

lasting standard.

2.2.3 Metrics

2.2.3.1 Speedup

A common metric for measuring the effectiveness of a parallel algorithm or hardware is *speedup*, which quantifies the gain obtained from parallelisation through a ratio between the runtime of the serial version and that of the parallelised version, as shown in Equation 2.1.

$$U = \frac{T_S}{T_P} \tag{2.1}$$

where U is the speedup obtained, T_S is the runtime of the serial version and T_P is the runtime of the parallelised version. For example, a speedup of four means that the parallelised version is four times faster than the serial version.

The formula given in Equation 2.1 is derived from Amdahl's law [47], which takes into account the portion of the program that is serial in the calculation of the maximal theoretical gain achievable, as given in Equation 2.2.

$$U = \frac{1}{(1-q) + \frac{q \times T_P}{T_S}} \tag{2.2}$$

where q is the fraction of the application that is parallelisable, T_s is the runtime of the serial version of the parallelisable portion of the application and T_p is the runtime of the parallel version of the parallelisable portion of the application. It follows that Equation 2.1 is a special case of Equation 2.2 where $q = 1$, thus the entirety of the application is considered parallelisable.

2.2.3.2 Parallel efficiency

A limitation of speedup is that it does not take into consideration the number of workers being used, and thus lacks context. In shared-memory parallelism, for instance, a speedup of four is ideal when using four threads. Similarly, a speedup of eight is ideal when using eight threads. However, a speedup of four obtained from eight threads corresponds to half of the ideal speedup. This aspect of parallelism is quantified by *parallel efficiency*, which is a ratio between the speedup obtained and the amount of parallelism, as shown in Equation 2.3.

$$P_E = \frac{U}{W} \tag{2.3}$$

where P_E is the parallel efficiency, W is the number of workers used and U is the speedup obtained when using W workers.

2.2.3.3 Scalability

From Amdahl's law, one can calculate what is referred to as *strong scalability*, which expresses the variation of the speedup against the worker count, using a fixed total problem size. However, as the number of workers increases, the benefits obtained decrease because the parallelisable portion of the application gradually becomes less significant. To make use of additional workers, a second approach has been explored, known as Gustafson's law [48], increases the problem size proportionally to the increase of the worker count. This is referred to as *weak scalability*, which expresses the variation of the speedup against the worker count, using a constant per-worker problem size.

In shared-memory parallelism, workers typically are threads, however, they can be threads or processes in distributed-memory parallelism. Codes may exhibit different scaling depending on whether the total number of workers increases as a result of increasing the number of threads, or the number of processes. This can be more accurately analysed by referring to *thread scalability*, which represents the variation of the speedup as the number of threads per process increases, and *process scalability*, which represents the variation of the speedup as the number of processes increases.

Distributed-memory parallelism has an additional factor: the number of nodes. The process scalability of a program is likely to vary depending on the placement of said processes due to the resulting amount of communications that must be achieved over the network. The notion of *node scalability* addresses this limitation of process scalability by describing the variation of speedup as the number of nodes increases, better reflecting the impact of topology on the performance observed.

2.3 Vertex-centric

As mentioned in Chapter 1, the writing of graph processing applications is not a trivial task, especially at large scale where the use of parallelism and high performance computing techniques becomes crucial in obtaining performance. In addition to being error-prone, the parallelisation and optimisation is time consuming and requires expertise likely beyond that of the typical vertex-centric user.

The vertex-centric programming model introduced in 2010 [11] aims to address this by providing a new way to express graph computation: from a vertex perspective. Using vertex-centric the user can develop graph processing applications from a localised view

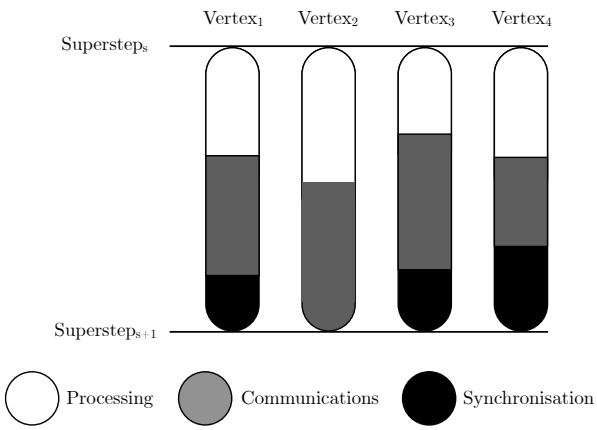

Figure 2.2: A Bulk-Synchronous Parallel superstep

of the graph and an interface consisting of a simple set of highly abstracted functions. Meanwhile, aspects important for performance, such as parallelism or low-level technicalities are offloaded to the underlying framework. This allows HPC experts to implement optimisation techniques, independently of the writing of the vertex-centric program by users. As it will be shown in the next subsection, vertices may be processed concurrently, which exposes a high degree of parallelism.

2.3.1 Execution flow

Vertex-centric programs follow an iterative execution flow based on the Bulk-Synchronous Parallel (BSP) model [49], where iterations, referred to as *supersteps*, comprise three phases, as illustrated in Figure 2.2.

The first phase, local computation, consists of applying the user-defined function `compute` to each *active* vertex (see definition in the next paragraph). During this phase, a vertex may update its state, read messages received from neighbours and prepare messages to be sent to neighbours. The second phase consists of performing communications, where messages generated by vertices are delivered to their recipient, and accessible in the subsequent superstep. The final phase is a global synchronisation, ensuring the coherence of the state of all vertices before proceeding with the next superstep. The design of the vertex-centric programming model was inspired by the Bulk-Synchronous Parallel model's preservation of semantics that are easy to reason about, therefore making the development of vertex-centric applications more intuitive and less error-prone.

The iterative execution flow, proceeding from one superstep to the next, continues until the termination condition is met. This termination commonly relies on the notion of a vertex's state which is either *active* or *inactive* as illustrated in Figure 2.3 and defined

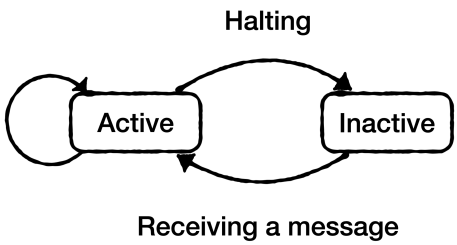

Figure 2.3: The finite state machine of a vertex state.

as follows:

- all vertices are active at the beginning of the first superstep.
- an active vertex becomes inactive when it *halts*, by calling the `halt` function.
- an inactive vertex becomes active solely by receiving a message.

As supersteps progress, the number of active vertices, which is counted at the end of a superstep and after all messages have been delivered, at the end of a superstep, after any pending messages have been delivered, may vary. When the number of active vertices reaches zero, the graph processing application terminates.

2.3.2 Benchmarks

Certain applications have been widely used to evaluate graph processing frameworks, including vertex-centric, and have become de facto benchmarks which are accepted by the community. The main three, used throughout this thesis, are PageRank, Connected Components and Single-Source Shortest Paths.

2.3.2.1 PageRank

First presented in 1998 [50], PageRank is an algorithm which evaluates the importance of vertices by using connectivity as the core metric, where the longer the in-adjacency list of a vertex, the more important it is.

PageRank was initially designed to better rank web pages listed in the results produced by search engines, where a web page citation importance is proportional to the number of hyperlinks pointing from and to that page. The original formula to calculate the PageRank of a web page is given in Equation 2.4.

$$P_R(A) = (1 - d) + d \times \sum_{p \in \Psi(A)} \frac{P_R(p)}{C(p)} \tag{2.4}$$

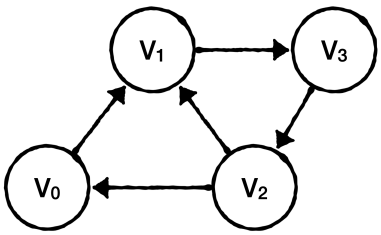

Figure 2.4: The graph used in the PageRank and Single-Source Shortest Paths simulations.

where $P_R(A)$ is the PageRank of page A, d is the damping factor, $C(p)$ is the number of links going out of page p and $\Psi(A)$ is the list of pages to which page A points.

From a vertex-centric point of view, PageRank has the interesting property that all vertices are active during the entire duration of the simulation. This minimises load balancing issues and allows experiments to focus on evaluating other performance factors.

Algorithm 1: Pseudo-code for the PageRank implementation in vertex-centric

```
1  begin
2      T ← 10                                   // The number of supersteps
3      d ← 0.85                                 // The damping factor
4      if Superstep_index = 0 then
5          Self.Value ← 1/V
6      else
7          Total ← 0
8          while Has_pending_message() do
9              M ← Fetch_next_message()
10             Total ← Total + M
11         Self.Value ← d/V + (1 - d) × Total
12     if Superstep_index < T then
13         SendToAllNeighbours(Self.value / Self.Out_degree)
14     else
15         Halt()
```

A pseudo-code implementation of PageRank in vertex-centric, based on the implementation presented in the original vertex-centric framework Pregel [11], is given in Algorithm 1. A simulation of this PageRank algorithm, assuming a maximum number of supersteps set to 4, on the graph presented in Figure 2.4 is illustrated in Table 2.1.

Superstep	Vertex	Messages received	Start value	End value	Messages generated	Halts?
0	0	-	-	0.25	0.25 to v_1	No
	1	-	-	0.25	0.25 to v_3	No
	2	-	-	0.25	0.13 to v_0, 0.13 to v_1	No
	3	-	-	0.25	0.25 to v_2	No
1	0	0.13	0.25	0.15	0.15 to v_1	No
	1	0.25, 0.13	0.25	0.36	0.36 to v_3	No
	2	0.25	0.25	0.25	0.13 to v_0, 0.13 to v_1	No
	3	0.25	0.25	0.25	0.25 v_2	No
2	0	0.13	0.15	0.15	0.15 to v_1	No
	1	0.15, 0.13	0.36	0.28	0.28 to v_3	No
	2	0.25	0.25	0.25	0.13 to v_0, 0.13 to v_1	No
	3	0.36	0.25	0.35	0.35 to v_2	No
3	0	0.13	0.15	0.15	-	Yes
	1	0.15, 0.13	0.28	0.28	-	Yes
	2	0.35	0.25	0.34	-	Yes
	3	0.28	0.35	0.28	-	Yes

Table 2.1: Simulation of a PageRank application with a maximum of 4 supersteps on the graph from Figure 2.4
(values are rounded to two decimal places)

2.3.2.2 Connected Components

The second benchmark presented in this chapter is Connected Components [51], which identifies all components of a graph. A component is a subgraph where each vertex is reachable by every other vertex, and the subgraph itself is not contained in another component.

Finding Connected Components can be part of larger clustering applications, or act as an optimisation technique that preprocesses the graph and provides a first insight into potential partitioning schemes.

A pseudo-code of the vertex-centric implementation of the Connected Components is given in Algorithm 2. In vertex-centric, components are found by propagating vertex identifiers throughout the graph and converging towards the smallest identifiers. Eventually, each vertex receives the identifier of all vertices it can reach. By discarding all but the smallest identifier, vertices forming a component will therefore all share the identifier of the vertex with the smallest identifier in the component. A simulation of this Connected Components algorithm on the graph presented in Figure 2.5 is given in Table 2.2.

In vertex-centric, the Connected Components benchmark exhibits a fundamental difference with PageRank, because vertices systematically halt as shown in line 14 of Al-

Algorithm 2: Pseudo-code for the Connected Components implementation in vertex-centric

```
1  begin
2      if Superstep_index = 0 then
3          Self.Value ← Self.Id
4          Send_to_all_neighbours(Self.Value)
5      else
6          Min_value ← Self.Value
7          while Has_pending_message() do
8              M ← Fetch_next_message()
9              if M < Min_value then
10                 Min_value ← M
11         if Min_value ≠ Self.Value then
12             Self.Value ← Min_value
13             Send_to_all_neighbours(Self.Value)
14     Halt()
```

Superstep	Vertex	Messages received	Start value	End value	Messages generated	Halts?
0	0	-	-	0	0 to v_1, 0 to v_2	Yes
	1	-	-	1	1 to v_0, 1 to v_2, 1 to v_3	Yes
	2	-	-	2	2 to v_0, 2 to v_1, 2 to v_3	Yes
	3	-	-	3	3 to v_1, 3 to v_2	Yes
1	0	1, 2	0	0	-	Yes
	1	0, 2, 3	1	0	0 to v_0, 0 to v_2, 0 to v_3	Yes
	2	0, 1, 3	2	0	0 to v_0, 0 to v_1, 0 to v_3	Yes
	3	1, 2	3	1	1 to v_1, 1 to v_2	Yes
2	0	0, 0	0	0	-	Yes
	1	0, 1	0	0	-	Yes
	2	0, 1	0	0	-	Yes
	3	0, 0	1	0	0 to v_1, 0 to v_2	Yes
3	0	-	0	0	-	Yes
	1	0	0	0	-	Yes
	2	0	0	0	-	Yes
	3	-	0	0	-	Yes

Table 2.2: Simulation of a Connected Components application on the graph from Figure 2.5

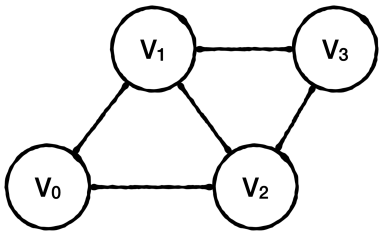

Figure 2.5: The graph used in the Connected Components simulation.

gorithm 2. This implies that the number of active vertices is expected to vary throughout supersteps, itself resulting in potential load balancing challenges. Unlike PageRank, the Connected Components application is typically run until no more vertices are active.

2.3.2.3 Single-Source Shortest Paths

The third benchmark presented in this chapter is Single-Source Shortest Paths (abbreviated SSSP) [52]. This selects a vertex and finds the shortest path from that vertex to every other vertex in the graph.

There are many applications of the SSSP algorithm, for example, to measure the time taken for units to be dispatched to every area of a city, or evaluate the relationship of an individual to their group by quantifying how "direct" their connections are.

A pseudo-code implementation of the vertex-centric implementation of the Connected Components is sketched in Algorithm 3. In vertex-centric, finding the Single-Source Shortest Paths is similar to Connected Components since both consist of propagating the lowest values obtained from the second superstep onwards. However, the major difference is that in SSSP the value to be sent is first incremented. Another difference in SSSP is during initialisation, all vertices apart from the source vertex initialise their value to ∞, representing a value strictly greater than any distance feasible from the source vertex, while the source vertex initialises its value to 0. A simulation of this SSSP algorithm on the graph presented in Figure 2.4 is given in Table 2.3.

Unlike PageRank and Connected Components, the Single-Source Shortest Paths benchmark exhibits a very low number of active vertices at first. Starting with a single active vertex, message propagation gradually activates a higher number of vertices, before decreasing as convergence is reached throughout the graph. This low number of active vertices results in a load imbalance greater than that observed in Connected Components.

Algorithm 3: Pseudo-code for the Single-Source Shortest Paths implementation in vertex-centric

```
1  begin
2      if Superstep_index = 0 then
3          if Self.Id = SOURCE_VERTEX then
4              Self.Value ← 0
5              Send_to_all_neighbours(Self.Value)
6          else
7              Self.Value ← ∞
8      else
9          Min_value ← Self.Value
10         while Has_pending_message() do
11             M ← Fetch_next_message()
12             if M < Min_value then
13                 Min_value ← M
14         if Min_value ≠ Self.Value then
15             Self.Value ← Min_value
16             Send_to_all_neighbours(Self.Value + 1)
17     Halt()
```

Superstep	Vertex	Messages received	Start value	End value	Messages generated	Halts?
0	0	-	-	∞	-	Yes
	1	-	-	∞	-	Yes
	2	-	-	∞	-	Yes
	3	-	-	0	1 to v_2	Yes
1	0	-	∞	∞	-	Yes
	1	-	∞	∞	-	Yes
	2	1	∞	1	2 to v_0, 2 to v_1	Yes
	3	-	0	0	-	Yes
2	0	2	∞	2	3 to v_1	Yes
	1	2	∞	2	3 to v_3	Yes
	2	-	1	1	-	Yes
	3	-	0	0	-	Yes
3	0	-	2	2	-	Yes
	1	3	2	2	-	Yes
	2	-	1	1	-	Yes
	3	3	0	0	-	Yes

Table 2.3: Simulation of a Single-Source Shortest Paths application, selecting vertex v_3 as the source vertex, on the graph from Figure 2.4

2.4 Conclusions

This chapter introduced the concepts from HPC and graph theory necessary to the understanding of this thesis. It also presented the vertex-centric programming model and the benchmarks selected for evaluations, illustrated with examples. The next chapter will describe in detail the work achieved by the graph community, focussing on the different types of optimisations techniques developed.

Chapter 3

Related work

3.1 Introduction

This chapter surveys the literature in areas relevant to the research conducted in this thesis. Techniques developed to optimise the vertex-centric programming model can be categorised into two groups. Firstly, approaches that ultimately result in the creation of a new programming model, introduced in Section 3.2. Secondly, techniques that remain compatible with the vertex-centric programming model are reported in Section 3.3. Section 3.4 then concludes with a brief analysis of the techniques and approaches presented.

3.2 New programming models

The popularity of vertex-centric programming has resulted in numerous attempts at optimising performance and memory efficiency. Some of these have grown to become an entirely new programming model, distinct from, but inspired by, the vertex-centric programming model.

3.2.1 Vertex-subset-centric

Ligra [53] is one of the most popular shared-memory frameworks for graph processing. It demonstrated, in 2013, that the evolution of hardware technology has enabled nodes to contain enough memory to process graphs made of tens or even hundreds of billions of edges. However, due to the size of the graphs considered, in existing literature, the emphasis was placed on distributed-memory solutions. Therefore, the authors of *Ligra* concluded that there was scope to design an efficient shared-memory framework able to process large-scale graphs.

The design of *Ligra* relies on the observation that many graph-based algorithms operate over small subsets of vertices. However, in *Pregel* computation is localised to individual vertices as it cannot be expressed at a more global scope than vertex-centric. Therefore, *Ligra* aims to support the processing of vertex subsets by shifting the scope of computation to vertex-subset-centric. An abstraction based on three components is proposed: two functions, `EdgeMap` and `VertexMap`, as well as a type `VertexSubset` that represents a subset of vertices in the graph processed.

The `EdgeMap` function accepts four arguments: a graph G, a `VertexSubset` S of G, a user-defined function F (which accepts two vertices and returns a boolean) and a user-defined function C (which accepts a vertex and returns a boolean). The `EdgeMap` function also returns a `VertexSubset`. The function F is then applied to edges whose source vertex is in S and target vertex evaluates to `true` from function C. The latter can be considered a filter, enabling the user to select or ignore neighbours based on user-defined criteria.

The `VertexMap` function accepts two input arguments and returns a `VertexSubset`. The input arguments are a user-defined function F, which accepts a vertex and returns a boolean, and a `VertexSubset` S. `VertexMap` then applies the user-defined function F to each vertex contained in the vertex subset S. Vertices from S that evaluate to `true` through function F are appended to a vertex subset, and subsequently returned by the vertex map function.

The workflow provided by *Ligra* is based around iterative execution that processes a given `VertexSubset` until a termination condition is reached, typically when the `VertexSubset` processed becomes an empty set. At the beginning of execution, the `VertexSubset` is initialised by the user and then updated by being passed to `EdgeMap` and `VertexMap` at every iteration, applying the user-defined functions specified until the termination condition is reached. By having placed vertex subsets at the centre of its design and execution flow, *Ligra* is categorised as a vertex-subset-centric framework [54].

Compared to the vertex-centric programming model, *Ligra* retains a global view of the graph and exposes explicit parallelism to the user as well as atomic operations, but does not provide a vertex halting feature or a message-passing interface. As stated in [53], given that the user-defined function passed to `VertexMap` and `EdgeMap` may be run in parallel, "the user must ensure parallel correctness". The characteristic of the vertex-centric programming model, where low-level considerations are hidden from the user to preserve a highly-abstracted interface, is therefore not reflected in *Ligra*.

Gemini [55] is a distributed-memory framework published in 2016. Described as computation-centric, it can be considered comparable to a distributed-memory equival-

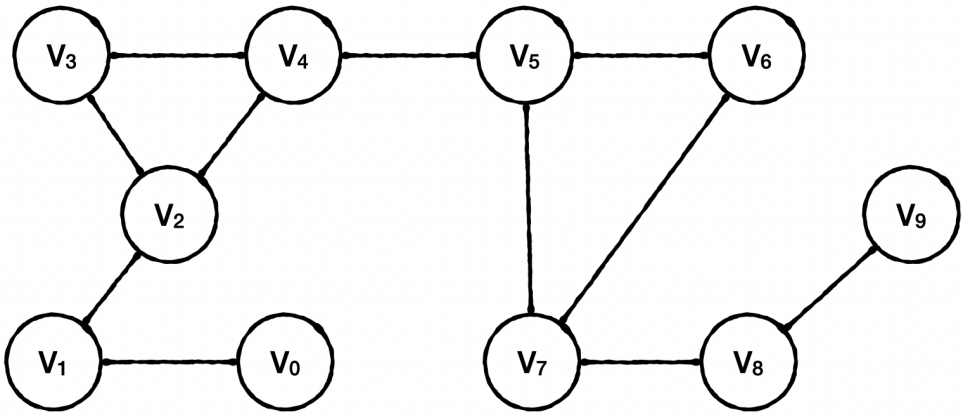

Figure 3.1: Undirected graph used in *Blogel* example

ent of *Ligra*, providing a similar interface, as well as having a workflow designed upon the concept of `VertexSet` which is very similar to *Ligra*'s `VertexSubset`. However, similarities also extend to limitations observed. For example, *Gemini* retains a global view of the graph, contrasting with the vertex-local view specific to vertex-centric programming. Furthermore, the *Gemini* framework exposes low-level parallelism too, such as requiring the explicit use of atomic operations, as well as lacking a message-passing abstraction.

3.2.2 Block-centric

The block-centric programming model was introduced in 2014 via *Blogel* [56]. This approach partitions the graph into blocks and allows applications to communicate at the block-level in order to hasten the propagation of messages.

Consider, as an example, the undirected graph given in Figure 3.1, which happens to be a connected component, and where each vertex can reach any other vertex. This graph has a graph diameter of 7, which means that at most 7 supersteps are needed for a message to propagate. When running Connected Components using the hash-min algorithm given in Subsubsection 2.3.2.2 for instance, the initial value 0 held in v_0 would therefore require 7 supersteps to reach vertex v_9.

By contrast, the graph can be partitioned via a block decomposition, for example, the one given in Figure 3.2 where each block is given a connected component. In *Blogel*, in the first superstep, the minimum value would be calculated in each block. Block 0 would therefore have a block value of 0, and block 1 would have a block value of 5. At the end of the superstep, these block values would be exchanged, and then propagated to the vertices held locally. At the end of the first superstep, all vertices would therefore obtain their final value.

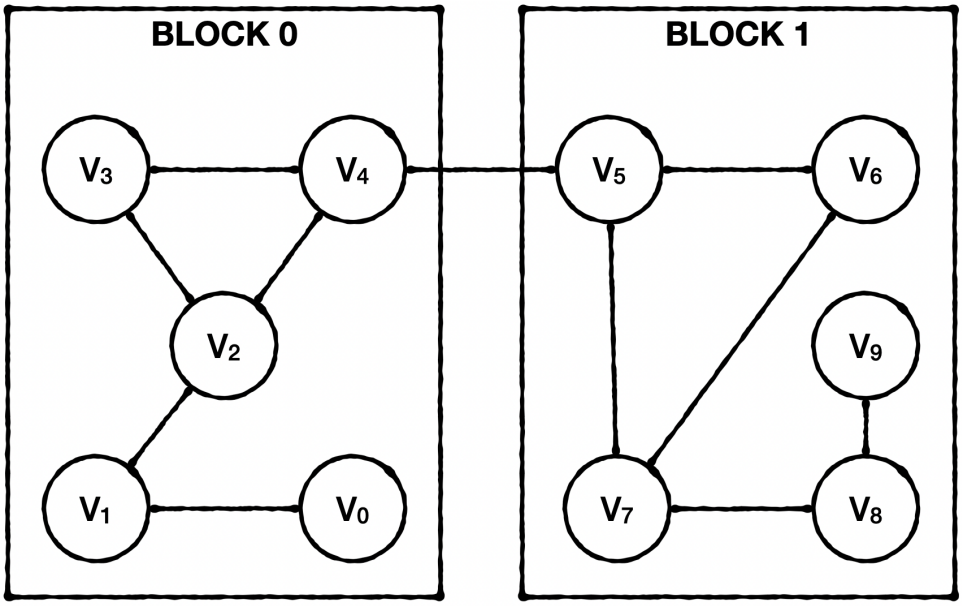

Figure 3.2: Example of a graph partitioning into blocks in *Blogel*

This is made possible by re-analysing the graph at the block-level. The new graph obtained would therefore contain two nodes (the two blocks) which implies that the graph diameter of this block-level graph is 1. This determines why any message value generated from a vertex within a block will take 1 superstep to reach any other vertex held in the other block.

This graph decomposition into Connected Components is the cornerstone of this programming model. *Blogel* is based on the observation that graphs with a skewed vertex distribution, where a few vertices are connected to many while many, typically contain a large connected component and numerous small ones. Their approach consists of building many individual blocks by converting each small connected component into a block, and splitting the large connected component into multiple blocks. By allowing reduction operations to be applied at the block-level, message propagation in single-block Connected Components is achieved within a single step. For a large connected component split over multiple blocks, the number of supersteps required for message propagation equals the number of blocks.

The *Blogel* programming model offers three modes: V-mode, B-mode and VB-mode. When using the V-mode, the interface and semantics are consistent with that of the vertex-centric programming model. An example is illustrated with PageRank, whose implementation in *Blogel* is given in Appendix C.

When enabling the B-mode, the interface can be considered a strict mapping of the vertex-centric interface to a block-centric level. The user-defined `compute` function is

now applied on blocks, and likewise for the `send_message` and `halt` functions. When all blocks have halted and no pending messages are left, the graph application terminates. The block-centric implementation of PageRank in *Blogel* is given in Appendix D.

When using V-mode, there is no noticeable difference compared to Pregel. However, to leverage the block-centric optimisation, additional code must be developed, which resulted in the 50-line PageRank implementation in V-mode growing to 250 lines required for B-mode.

The VB-mode is a hybrid which combines both V-mode and B-mode. Communications can occur at the vertex-level and block-level, and both vertex `compute` and block `compute` functions coexist. The termination condition is satisfied when all vertices, and all blocks, have halted and there are no pending messages.

This customisation of the API via different modes is an interesting approach to maintaining the programmability benefits of vertex-centric whilst maximising performance. However, this comes at the expense of the extra work necessary to develop a *Blogel* application leveraging the B-mode. As shown with the PageRank implementation, although the block-centric programming model used in B-mode is a direct transposition of the vertex-centric programming model at the block-level, it does not allow the user to write programs as concisely. Therefore it can be argued that this does not solve the underlying problem, as users must rely on B-mode for performance which imposes significant additional programming overhead.

GoFFish [57] was published in 2014 and is conceptually similar to *Blogel*, although the latter also developed its own graph partitioning techniques. The experiments conducted suggest the importance that an initial graph partitioning based on Connected Components may have for performance, not only directly as a consequence of the load balancing improvement obtained, but also indirectly through the kind of block-centric optimisations it enables. It must be noted, however, that these techniques had also been explored in the graph-centric programming model introduced in Giraph++ [58] published a year earlier, in 2013. These three independent frameworks, therefore, improved the performance of the vertex-centric programming model by targeting the message exchange phase, speeding up the message propagation through programming at a subgraph-level and partitioning the graph as a preprocessing phase.

3.3 Optimisations within vertex-centric

The second category presented in this chapter contains optimisations that are leveraged from within vertex-centric and often transparent to the user.

3.3.1 Sender-side combination

In distributed-memory frameworks, vertices are distributed across workers which are mapped to multiple nodes. Messages generated intended for vertices held by a worker residing on a different node are therefore queued for later sending. A major limitation of this approach is that the queue might reach a size equalling the graph size.

One of the optimisations introduced in the original Pregel [11] is *sender-side combination*, which aims to reduce the number of messages generated on the network by combining them before sending. This optimisation relies on the observation that, in certain algorithms, a vertex does not seek the value contained in each message individually but only the result of their reduction, such as a sum or minimum. In these cases such an intent can be expressed through the use of a *combiner*, which is a user-defined function that accepts two message values as input and returns their reduced result. *Pregel* does not provide any guarantee around which messages (if any) are combined, or in what order. Therefore, this implies that, as specified in [11], the combination operation must be commutative and associative.

Applying a combiner to queues allows workers to perform a combination pass upon their queued messages and reduce the size of these queues before sending. Consequently, a lower number of messages is ultimately sent over the network, alleviating pressure on network bandwidth and improving overall performance.

Despite being known since the original Pregel, there have been cases where the combiner optimisation is not implemented by other frameworks, such as *FemtoGraph* [38]. This is a major omission because, not leveraging the use of combiners on the sender-side guarantees that message queues may grow until containing a number of elements equal to the graph size. As it will be shown in Section 7.2, not supporting the use of combiners, or deferring their execution, results in a memory overhead that can be significant.

3.3.2 Receiver-side scatter

In distributed-memory frameworks, vertices may share edges with vertices held on a worker residing on a different node. Except in the theoretically possible, though highly unlikely, situation where a disconnected graph whose disjoint subgraphs can be mapped perfectly to distributed-memory workers, graph partitioning will result in cross-worker edges. An example of a graph G, distributed over two distributed-memory workers, is given in Figure 3.3. This partitioning places vertex V_0 on worker 0, and vertices V_1, V_2 and V_3 on worker 1, resulting in three cross-worker edges.

When vertex V_0 generates messages M_0, M_1 and M_2 to be sent to vertices V_0, V_1 and V_2 respectively, this results in three cross-worker messages, as illustrated by Figure 3.4. Such

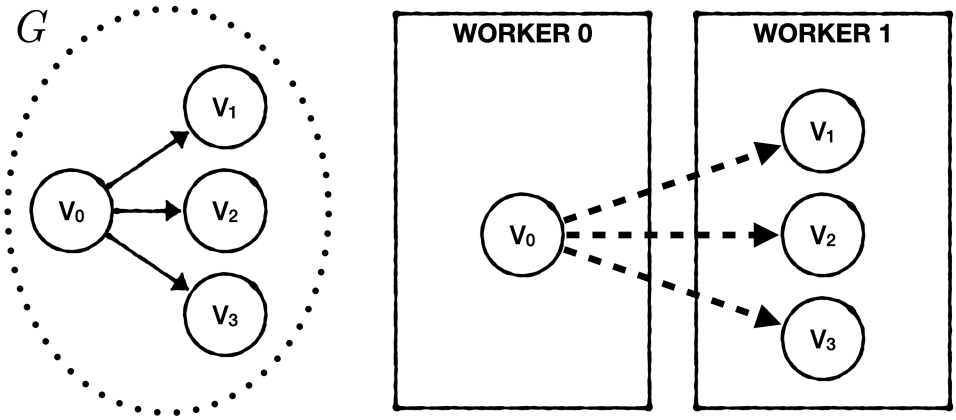

Figure 3.3: Example of a graph G, along with a distributed-memory partitioning

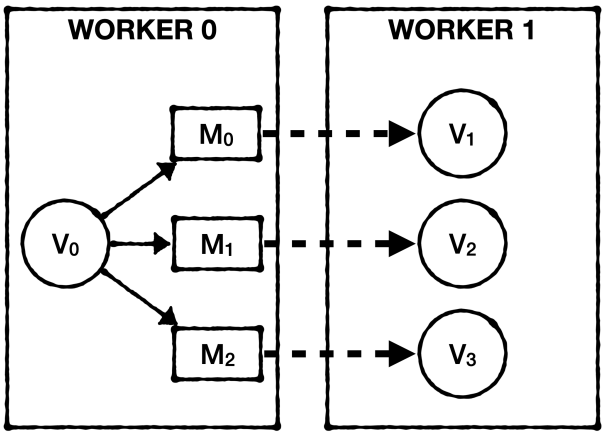

Figure 3.4: General cross-worker communication pattern

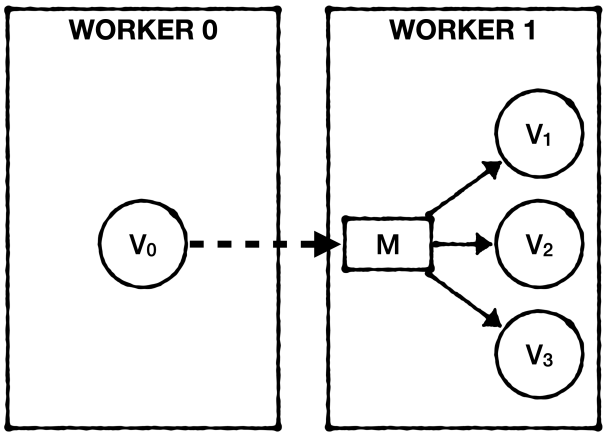

Figure 3.5: Example of a receiver-side scatter optimisation

a situation represents the most generic situation where no assumption is made about the vertex communication pattern when undertaking the decomposition of the graph across workers.

It follows that a vertex with a high out-degree, also referred to as a hub vertex, may generate a high number of cross-worker messages when communicating. In vertex-centric applications where vertex communications comprise neighbouring broadcasts, the messages generated by a given vertex contain the same value. In Figure 3.4, this translates to messages M_0, M_1 and M_2 containing an identical value, which creates a context where the *receiver-side scatter* optimisation can be leveraged. As demonstrated in *X-Pregel* [32], *LFGraph* [33], *GRE* [34], this technique is an equivalent to a multicast technique applied to message transmission, where it consists of sending the broadcast message once to each recipient worker. On receipt, the worker reroutes a copy of this message to each of the recipient vertices that it holds. An example of the receiver-side scatter technique applied to this situation is depicted in Figure 3.4 is given in Figure 3.5.

Whilst this technique reduces the overall amount of cross-worker communication generated by vertices, in two out of the three frameworks cited, there was a reduction in abstraction. This shifted the vertex-centric model into either a subgraph-centric programming model (*X-Pregel*) or a scatter-combine computation model paired with an agent-based data model (*GRE*).

3.3.3 Vertex-mirroring

In graphs with highly skewed distributions, hub vertices are frequently a major cause of load imbalance in vertex-centric programs. Therefore, efforts have been made to handle such vertices more efficiently, including the receiver-side scatter technique presented in

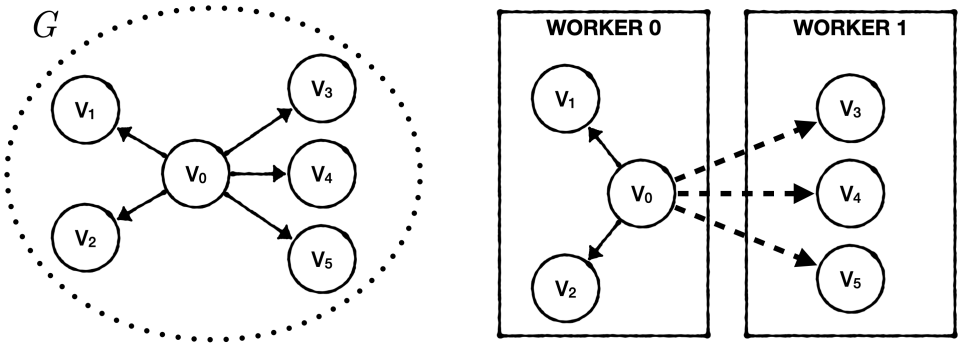

Figure 3.6: Example of a graph G, along with a distributed-memory partitioning

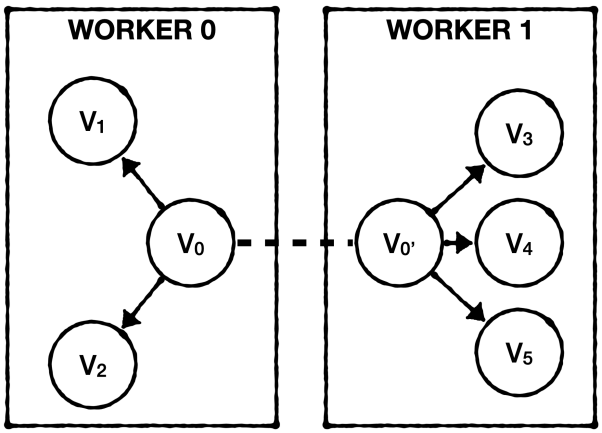

Figure 3.7: Application of the vertex-mirroring technique on vertex v_0

Subsection 3.3.2. A similar effect is obtained with another technique, called *vertex-mirroring*, and first introduced in *Pregel+* [12].

The vertex-mirroring technique consists of virtually splitting a vertex v by duplicating it upon every worker which holds vertices in the adjacency list of v, similarly to ghost vertices/edges in *Distributed GraphLab* [35]. These mirrors act as worker-local repeaters upon receipt of a message from their original vertex. Using the graph illustrated in Figure 3.6 as an example, vertex v_0 on worker 0 would be mirrored into vertex v_0' on worker 1, as illustrated in Figure 3.7. In situations where vertex v_0 broadcasts a message M to its neighbours, the message can be delivered directly to recipient vertices on the same worker as v_0. For recipients residing on a different worker, the message is sent to the corresponding vertex mirror instead, which in turn will broadcast the message to the recipients held on the vertex mirror worker.

This technique, therefore, helps alleviate the impact on the performance of load imbalance generated by hub vertices. Selecting the mirroring threshold, which is the degree above which vertices are mirrored, remains a challenging task and as reported in Figure 12 in [12], may result in a performance difference of a factor of two depending on its

value. It should be noted that, in addition to its similarity with the receive-side scatter technique presented in Subsection 3.3.2, the vertex-mirroring technique can also be considered a vertex-centric equivalent of the vertex-cut technique introduced in *PowerGraph* [59]. The authors of *Pregel+* highlight that the vertex-mirroring technique is also similar to the large adjacency-list partitioning technique used in *GPS* [60]. These observations demonstrate that the efficient handling of hub vertices has attracted significant attention both in general graph processing and in vertex-centric.

3.3.4 Request-respond paradigm

The second optimisation introduced in *Pregel+* [12] was the *request-respond* paradigm, which focusses on the structure of communications in vertex-centric programs, which are solely vertex-to-vertex. This characteristic results in any ping-pong interactions, for example, a vertex requesting the value of another vertex, requiring two supersteps to complete. Inefficiencies arise when multiple source vertices request the same target vertex value. For example, in a scenario where there are L source vertices, such request-responds would naively require $2 \times L$ messages using classic vertex-centric communications.

The request-respond technique presented in *Pregel+* addresses this issue by allowing vertices to explicitly express their intent to request the value of a certain vertex by calling the function `request` and passing it the identifier of the target vertex. The target vertex of any such request is registered, if it is not present already, into a table which tracks all target vertices of interest. At the end of the superstep, workers first exchange batches of ordinary *Pregel* messages, and then proceed with the sending of the sets of target vertex identifiers. Upon receipt of a request list, a worker fetches, for each target vertex, the corresponding vertex value and returns it along with the vertex identifier. This set of records is then received by the requesting worker and, in the subsequent superstep, any vertex requesting the value can do so by passing the identifier of the target vertex to the `get_resp` function. Vertices can also preemptively generate such responses by calling the `respond` function.

The benefit of this technique is that, regardless of the number of source vertices generating requests for a specific target vertex, at most one request and one response are exchanged at the worker-level, decreasing the number of messages to $min(W, L)$, where W is the number of workers and L the number of source vertices.

However, to leverage this optimisation, vertex-centric programs written in *Pregel+* must implement a `respond` function, which will be called to fetch the value of the calling vertex. This means that applications written in *Pregel+* already would need to be rewritten from the pure vertex-centric approach to leverage the request-respond

optimisation provided.

3.3.5 Communication channels

The concept of *communication channels* was first introduced into a vertex-centric approach in 2017 in the *channel-based Pregel system* [61], with the aim of improving vertex-centric communications. This is based on the observation that vertex-centric implementations of certain algorithms, such as minimum spanning forest, require multiple computation phases which themselves often involve different communication patterns.

This results in two disadvantages, the first, given that the type and structure of the data contained may vary, is that the message type must be large enough to encode any potential payload. The second issue is the lack of information at the framework-level to infer optimisations in the communication patterns used, due to all the messages being alike.

To address these issues, communication channels were introduced which encode predefined communication patterns that can be used to perform different phases in the vertex-centric application. Certain channels require a list of edges to operate on, and where edges of interest must have been added to the channel's list using `add_edge` beforehand. For instance, one of the channels is `ScatterCombine`, which sends the provided value along all specified edges, and then subsequently combines this value with that on the receiving vertex using the user-defined `combine` function. Numerous predefined communication channels are provided, for instance one implementing *Pregel+*'s request-respond technique, as discussed in Subsection 3.3.4.

An additional benefit of communication channels is their ability to be composed, and this is especially useful in supporting algorithms where multiple types of communication patterns may be needed, such as the Shiloach-Vishkin algorithm [62]. The corresponding implementation in the *channel-based Pregel system*, given in Appendix F, uses 3 different communication channels, `ScatterCombine`, `PushCombiner` and `RequestRespond`.

Communication channels successfully address both the issues identified above. Providing the framework with a set of channels, each tailored to a specific use, allows vertex-centric programs implemented in the *channel-based Pregel system* to benefit from individually optimised communication patterns both for performance and message input type.

However, the major limitation is that to enable communication channels, vertex-centric programs must be rewritten. Additionally, users must understand the exact semantics of each communication channel, as well as which rewritings are needed for this channel, in addition to assessing their potential suitability for the underlying communic-

ation pattern used in a given algorithm phase. Analysis of the applicability of communications channel based on an algorithm's characteristics might be considered beyond the original set of expectations in vertex-centric programming, where such low-level concerns are offloaded to the framework while the user is meant to focus solely on developing a graph processing application.

3.3.6 Selective scheduling

In vertex-centric algorithms, it is common for computation to converge faster on certain parts of the graph than others. To address this, the *selective scheduling* optimisation was introduced by *GraphChi* [24] which aims to focus computation on vertices that are yet to reach convergence.

This technique, similar to the approach of `VertexSubset` in *Ligra* and presented in Subsection 3.2.1, consists of vertices explicitly scheduling other vertices for processing in the following superstep. At the beginning of the first superstep, all vertices are scheduled for processing whereas only vertices explicitly scheduled are processed in subsequent supersteps.

For example, consider where a vertex broadcasts a value predicted to result in a noticeable change, that is above the convergence threshold specified, in certain neighbouring values. A vertex in this situation can schedule such neighbours for processing, allowing other neighbours to potentially never be processed should they not receive a value resulting in a change above the convergence threshold.

The benefit of selective scheduling is in the time saved by skipping inactive vertices. In addition to saving time undertaking a vertex check, which may become noticeable on graphs with a high order, there is also a load balancing improvement associated with this technique. Traditionally, vertices that must be processed are grouped into chunks, which are then assigned to workers. However, chunks may contain different proportions of inactive vertices, resulting in workers receiving different workloads. Although certain dynamic load balancing techniques may help partially alleviate this effect, a better approach is to begin with a set of vertices that are already pruned to active vertices as this will result in every chunk containing no vertices to be skipped.

However, the need for this technique may be considered as a consequence of having lost certain vertex-centric abstractions. As explained in [24], *GraphChi* does not provide the message-passing abstraction of the vertex-centric programming model. This goes against the original vertex-centric execution flow, where inactive vertices become active again upon receipt of a message. Because the signal is no longer present to wake up vertices, the corresponding opposite, the vertex halting feature, was also removed in *GraphChi*.

Removing these two abstractions resulted in vertices no longer holding a state indicating whether they are active, or inactive. Processing such vertices can therefore no longer distinguish vertices that may have already reached convergence, as normally indicated with an inactive state in vertex-centric programs. To address this the selective scheduling technique reconstructs the set of active vertices, at every superstep.

The drawback of selective scheduling is that it exposes the construction of the set of active vertices to the users, who must manually append vertices to the list of vertices to process as part of their application code. This involvement of the user in the underlying scheduling mechanism of the framework further degrades the original benefit of the vertex-centric programming model, originally providing a highly abstracted interface to the user while offloading low-level technical considerations to the underlying framework.

3.3.7 Vertex inactivation

The replacement of the message-passing abstraction in favour of promoting faster convergence through direct memory accesses, observed in the selective scheduling technique presented in Subsection 3.3.6, is also found in *Palgol* [21]. The limitations highlighted around the selective scheduling technique, where the removal of the vertex-centric message-passing abstraction fundamentally breaks the original execution flow of vertex-centric programs where vertices are normally made active again upon receipt of such a message.

In *Palgol*, this resulted in the creation of a *vertex inactivation* technique which preserves the vertex halting feature, but changes its semantics. A vertex calling `halt` in *Palgol* becomes inactive, like in *Pregel*, but the difference is that it may never become active again. This is explained by the authors in [21]:

> In *Pregel*, an inactive vertex can be activated by receiving messages, but such semantics is unsuitable for *Palgol* since we already hide message-passing from programmers. Instead, a stopped vertex in *Palgol* will become immutable and never perform any subsequent local computation, but other vertices can still access its fields. This feature is still experimental and we do not further discuss it in this paper; it is, however, essential for achieving the performance reported in section 4.

As acknowledged by its authors, and similarly to the selective scheduling technique, seeking performance gains by removing one of the vertex-centric abstractions, here the message-passing abstraction, ultimately led to the removal or degradation of other vertex-centric abstractions.

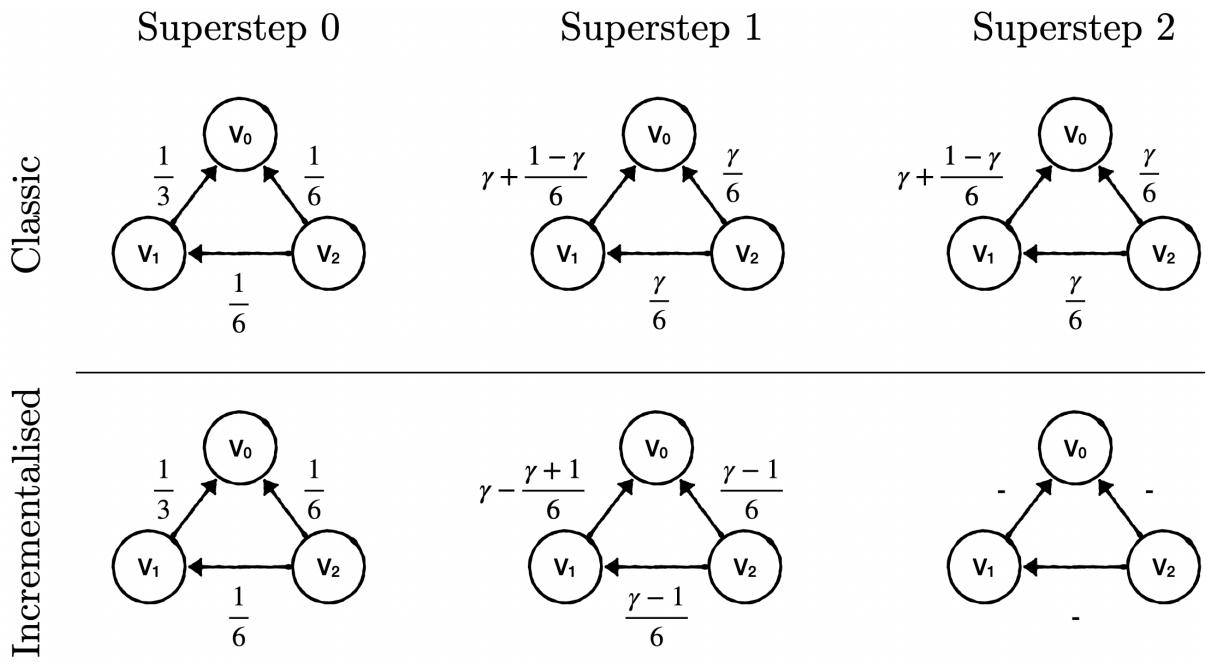

Figure 3.8: Example of an incrementalisation in PageRank, for the first three supersteps, assuming a damping factor γ

3.3.8 Incrementalisation

In vertex-centric, incrementalisation is the refactoring of message exchanges to no longer represent whole values, but increments. For instance, considering the PageRank algorithm introduced in Subsubsection 2.3.2.1, one characteristic is that, at any given superstep i, active vertices calculate their new value based on the sum of values received from incoming neighbours. In this case, messages can be refactored such that they no longer represent a whole value sent from a vertex, but instead the difference, or delta, from the previous message. This allows vertices to preserve their original value throughout supersteps, and only update it with the differences received from other vertices. An example is given in Figure 3.8, with both classic and incremental execution flows for a graph containing three vertices, where the value of a vertex is updated according to Equation 3.1:

$$S_i = \begin{cases} \dfrac{1}{|V|} & \text{if first superstep} \\ \gamma + (1-\gamma)\displaystyle\sum_{m \in M} m & \text{otherwise} \end{cases} \tag{3.1}$$

where S_i is the updated value of the vertex at superstep i, $|V|$ is the graph order, γ is the damping factor and M is the list of messages received. The message generated by vertices contains their value divided by the number of out-neighbours. When moving

towards the incremental approach shown in Figure 3.8, a vertex simply updates its value according to Equation 3.2:

$$S_i = \begin{cases} \frac{1}{V} & \text{if first superstep} \\ \gamma + (1-\gamma)\sum_{m \in M} m & \text{if second superstep} \\ S_{i-1} + \sum_{m \in M} m & \text{otherwise} \end{cases} \tag{3.2}$$

where S_i is the value of that vertex at superstep i. It should be noted that, in equation that in Equation 3.2, messages in M are incremental messages, by contrast to whole messages in Equation 3.1. As shown in Figure 3.8, the incrementalisation of PageRank allows faster convergence and reduces the number of messages exchanged.

However, incrementalising a vertex-centric program also requires in application rewriting, and most often significant application level code changes. This suffers the trade-off encountered between performance and vertex-centric programmability. The ΔV framework [36] enabled the automatic incrementalisation of vertex-centric programs using pagerank as a driver. Exploration of incrementalisation in vertex-centric programs is still active [63].

3.3.9 Message prioritisation

Another approach to restructuring vertex-centric communications is Slow Passing Fast Consuming (SPFC). This technique is motivated by the observation that, in vertex-centric programs, certain messages have a higher impact on reaching convergence than others. The approach aims at reducing the total number of messages generated by selecting only those messages from vertices with highest priority.

To prioritise vertices, a formula or metric determining a vertex priority must be defined, with messages generated then taking the priority of the sender vertex. The authors provide a list of recommended prioritisation strategies for common graph processing applications. For instance, in SSSP they recommended that priority is given to vertices with small identifiers, whereas in PageRank vertices with small in-degree are recommended for priority. In addition, a threshold indicating the maximum number of messages that can be generated during the superstep must be specified.

When a vertex expresses an intent to send messages, it is queued by the message-passing scheduler. At the end of the superstep, should this vertex be in the set of prioritised vertices, it will be selected and its messages sent. Eventually, all queued

vertices will be selected for message transmission, although this may occur in different supersteps.

However a limitation with this approach is that, by queuing vertices and delaying their message sending, intermediate states of vertices may be lost. For instance, a vertex broadcasting its value at superstep i, while being already queued since superstep $i - 1$, where the corresponding broadcast meant to happen at superstep $i - 1$ is lost. It depends upon the algorithm whether this impacts correctness, however for SSSP the most recent state is always the only one relevant and such loss of preceding values does not matter. However, for pagerank all intermediate states are required and this would impact correctness. To address this issue, *SPFC* provides a sender-side accumulator that combines the message value already in the queue with a new one. In the case of PageRank for example, these values are summed. When the vertex is scheduled for message sending, the value transmitted therefore contains information from both messages.

In *SPFC*, the prioritisation strategy and threshold play an important role in the overall impact on performance. An unsuitable prioritisation strategy may cause important vertices to be scheduled late, thus increasing the number of supersteps, as well as the number of low-value messages. The threshold is also important, where having a value too high may increase the number of supersteps as only a select few vertices would be systematically prioritised for communication, eventually outweighing the benefits obtained from the message reduction. However, having a threshold value too low may yield a performance gain lower than the overhead of the message-passing scheduling, thus ultimately degrading performance. Experiments conducted in [37] demonstrate that different thresholds may result in a performance difference of an order of magnitude.

From a programmability point of view, the approach presented in *SPFC* exposes low-level considerations to the user, who needs to understand the underlying message-passing scheduling architecture, as well as defining both a suitable prioritisation strategy and an appropriate threshold.

3.4 Conclusions

In this chapter, numerous vertex-centric optimisation techniques have been presented, as shown in Table 3.1. Their analysis in Section 3.2 revealed that an entire set of optimisations resulted in the removal or modification of vertex-centric abstractions, ultimately mutating the vertex-centric programming model into an entirely new programming model with a broader scope, from vertex-subset-centric to block-centric (the latter being also referred to as subgraph-centric or graph-centric).

Technique	New programming model	Within vertex-centric
Vertex-subset-centric	✓	
Block-centric	✓	
Sender-side combination		✓
Receiver-side scatter		✓
Vertex-mirroring		✓
Request-respond paradigm		✓
Communication channels		✓
Selective scheduling		✓
Vertex inactivation		✓
Incrementalisation		✓
Message prioritisation		✓

Table 3.1: Summary of optimisation techniques surveyed and their categorisation depending on their impact on the vertex-centric programming model.

Then, Section 3.3 listed optimisation techniques that did not result in the creation of a new programming model. However, it was observed that most techniques eventually degraded or removed certain vertex-centric abstractions, such as the suppression of the message-passing abstraction, the removal (or change of semantics) of the vertex halting feature and the exposition of low-level considerations such as message scheduling and vertex selection.

The survey conducted in this chapter, therefore, indicates how difficult the design of vertex-centric optimisations can be when programmability must be preserved. However, it may also suggest that research in such programmability-preserving optimisations is an area of vertex-centric yet to be explored further.

Chapter 4

Establishing a new state-of-the-art in vertex-centric shared-memory processing

4.1 Introduction

Almost a decade ago, the vertex-centric model was introduced through *Pregel* [11], and thanks to its expressiveness and scalability, this model rapidly gained in popularity. Many vertex-centric applications were soon developed, for example social network analysis [64] and data analytics [65, 66].

The majority of existing vertex-centric frameworks use distributed-memory parallelism, out-of-core computation where external memory is used, or both. Such approaches enable the processing of graphs which are too large to fit in the memory of a single node. However, these approaches also include overheads such as network communications for distributed-memory systems, and disk IO for out-of-core solutions. Approaches that only rely on the shared memory of a single node are another option, and do not suffer from the aforementioned disadvantages. However, the fundamental limitation of this approach is the amount of resources, specifically memory, available within a single node.

Surveys indicate that vertex-centric frameworks typically have a large memory footprint [67], reaching up to 264GB required to process PageRank over a graph of fewer than two billion edges [25], thus representing approximately 8GB of data. Such overhead is incompatible with a viable in-memory shared-memory framework, therefore resulting in a need to greatly reduce the memory footprint of vertex-centric frameworks. In addition, the existing in-memory shared-memory vertex-centric frameworks do not significantly outperform their distributed memory or out-of-core counterparts. Therefore, arguably

the potential of the in-memory shared-memory architecture, which was illustrated with *Ligra* [53] for vertex-subset-centric programming, is yet to be fully realised in vertex-centric programming.

As described in Chapter 3, vertex-centric frameworks typically incur a compromise between abstraction provided to the user, performance achieved and memory footprint generated. To improve performance or reduce memory footprint, vertex-centric frameworks commonly abandon certain features or abstractions from the vertex-centric model, which can significantly impact its cornerstone of programmability. Meanwhile, maximising performance and memory efficiency remains crucial to vertex-centric frameworks to support the processing of current and next generation graphs.

The contributions made towards these objectives can be described as follows:

- **A highly modular design** allowing optimisations to be enabled without requiring application rewriting, implemented in our in-memory shared-memory framework, *iPregel*.

- **A set of optimisation techniques** that leverage vertex-centric combiners, automatically preselect vertices to run based on communication pattern analysis, and efficiently address vertices.

- **Programmability-independent optimisations**, where techniques implemented do not require, or result in; the degradation, or removal, of vertex-centric abstractions.

- **A thorough evaluation of vertex-centric frameworks** across three major attributes: performance, memory efficiency and programmability.

The rest of this chapter is organised as follows: Section 4.2 presents related work and Section 4.3 provides an overview of the *iPregel* framework, from its interface and implementation to the optimisations designed and how they can be leveraged by the user, followed by Section 4.4 which presents the application benchmarks selected. Section 4.5 presents experiments designed, and results collected, to assess the competitiveness and viability of in-memory shared-memory frameworks in vertex-centric programming. Then, Section 4.6 focusses on providing a thorough comparison between shared-memory frameworks, before Section 4.7 draws conclusions and discusses potential further work directions.

4.2 Related work

The majority of existing vertex-centric frameworks exploit distributed-memory parallelism, for example *Pregel+* [12] and *Giraph* [68]. These frameworks are categorised as in-memory, where they exclusively use DRAM for storage. Distributed memory parallelism enables the use of larger memory spaces, decomposed across the physical nodes. Memory is typically the determining factor when it comes to the number of nodes required to process a graph. An increased number of nodes requires more network communications, which exacerbates the fundamental bottleneck of distributed-memory parallelism.

Although, in theory, distributed-memory systems using in-memory storage can process graphs of any size, given enough nodes, a trend is observed towards the development of distributed-memory systems also relying on out-of-core computation, such as *GraphLab* [69], *Pregelix* [70] and *GraphD* [25]. Given their ability to use both DRAM and disk storage, out-of-core solutions typically offload unused data to disk and only keep in DRAM data that is currently needed. The resulting disk IO is the challenge faced by such frameworks, which often hide the corresponding latency using overlapping techniques for instance.

To avoid the overheads associated with distributed memory parallelism, certain frameworks moved to shared-memory parallelism, with out-of-core computation. This can be observed with *GraphChi* [24], which is a spin-off of its distributed-memory counterpart *GraphLab* [69]. *GraphChi* [24] is a single-node vertex-centric framework able to process, in theory at-least, graphs of any size by relying on disk storage as an extension of memory. In *GraphChi*, the graph is divided into disjoint intervals, each of which is represented by a shard that stores all incoming edges of the vertices in that interval, on disk. Shards[1] are then loaded, in turn, into memory and the vertices belonging to the interval they represent are processed concurrently. With this design, *GraphChi* is able to process graphs that do not fit in memory, overcoming the fundamental limitation of single-node frameworks, at the expense of costly disk accesses.

There are other frameworks that are entirely in-memory shared memory, such as *Ligra* [53]. This shared-memory framework using in-memory storage was a significant improvement in the state-of-the-art as it enabled the processing of large graphs in shared memory. Its authors argue that the amount of memory available in high-end nodes is sufficient to process graphs with hundreds of billions of edges. More commonly, modern cluster nodes have between 64GB and 512GB of memory, which remains sufficient to process graphs up to a hundred billion edges in *Ligra*. Therefore, single-node frameworks that rely exclusively on in-memory storage such as *Ligra* can be viable in large-scale

[1] When determining the size of a shard, *GraphChi* ensures that it is sufficiently small to fit in-memory

```
/**
 * @brief This function performs the actual superstep calculations of a
 * vertex.
 * @param[inout] me The vertex to process.
 **/
void ip_compute(struct ip_vertex_t* me);

/**
 * @brief This function combines two messages into one.
 * @param[inout] old The existing message in the vertex mailbox.
 * @param[in] new The message that arrived for receipt.
 **/
void ip_combine(IP_MESSAGE_TYPE* old, IP_MESSAGE_TYPE new);
```

Figure 4.1: User-defined functions of *iPregel*

processing too. Experiments conducted [53] show that *Ligra* is able to scale to graphs containing almost 13 billion directed edges while preserving a parallel efficiency between 45% and 80% on 40 cores. However, as described in Subsection 3.2.1, from a programming perspective, *Ligra* is described as a vertex-subset-centric framework [54].

FemtoGraph [38] is another example of a framework relying on shared-memory parallelism and in-memory storage. The major difference from *Ligra* is that *FemtoGraph* fully preserves all vertex-centric abstractions. However, based on the results reported in [38], at best *FemtoGraph* provides little to no performance gain compared to existing graph processing frameworks. In addition, it also appears to suffer from significant performance overhead at a low number of threads.

4.3 Overview of iPregel

iPregel was developed as a shared-memory, vertex-centric framework to act as a vehicle for exploring and developing appropriate techniques. This enables such techniques to be evaluated in a sterile environment without the baggage of existing technologies.

4.3.1 Interface

In *iPregel*, the user is provided with a simple Application Programming Interface (API), in which they must define the `compute` and `combine` functions, the signatures of which are illustrated in Figure 4.1. The `compute` function contains the computation to execute on each vertex, whereas the `combine` function is related to the message combination feature, which is detailed further later in Subsubection 4.3.4.3.

Appendix A illustrates supporting functions provided by *iPregel* that allow the user

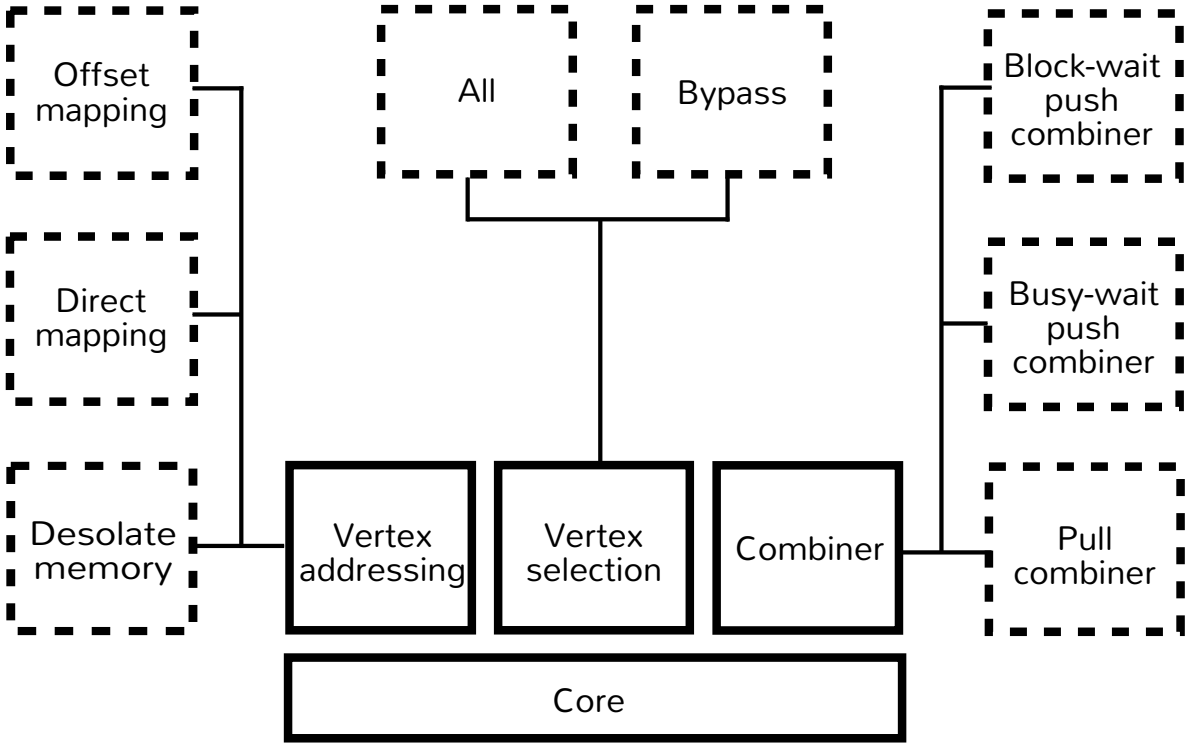

Figure 4.2: Structure of the *iPregel* framework

to track superstep progression, read messages received from the previous superstep, send a message to a specific out-neighbour or all out-neighbours at once, and halt the vertex currently processed respectively.

Although *iPregel* relies on shared-memory parallelism, communications are achieved via a message-passing abstraction. The motivation behind this choice for shared-memory is multifold; firstly, it protects the user from potential data-races that arbitrary memory accesses could allow. Secondly, the use of direct memory accesses requires the programmer to know exactly where to write information, which implies exposing implementation details to the user. Finally, this approach provides *iPregel* with the freedom to optimise the underlying communication mechanisms whilst preserving a consistent interface for the user.

4.3.2 Architecture

The *iPregel* framework is developed in C and parallelised using the shared-memory API OpenMP [45]. In programming, optimisations may be always applicable, or require a set of assumptions to be satisfied as a precondition. Generally, the latter are not considered because they come at the expense of software flexibility. In *iPregel* however, this is mitigated via the multi-version design illustrated in Figure 4.2.

The internals of *iPregel* are structured around a core that is unique and static, and modules that have alternative implementations. Each module version is optimised for a specific set of assumptions. Therefore, with this design, *iPregel* remains flexible while able to apply assumption-specific optimisations. The three modules visible in Figure 4.2, as well as their versions, are presented later in this chapter, in Subsubsections 4.3.4.1, 4.3.4.3 and 4.3.4.4.

4.3.2.1 Multi-version module selection

In order to keep lower-level optimisations concerns hidden from the user, the internal modular approach of *iPregel* is abstracted away from them. Nonetheless, the user's knowledge about the graph processing algorithm is important information that may be useful for performance optimisations. As such, users must be provided with a means to express this additional information. In *iPregel* this is achieved via defines passed via compilation flags, and was selected because it is already widely used in programming, and allows application source codes to remain unmodified. An example of this is where many vertex-centric algorithms exclusively require neighbour broadcasts for communication, and by passing this information to *iPregel* the framework is potentially able to leverage this.

4.3.2.2 The core

The core is the static and central part of *iPregel* is where the multi-version selection takes place. The core also acts as the interface to *iPregel* by providing all function declarations, which are consistent across module versions. Certain functions are version-dependent, where, although they share the same function prototype, there are different implementations for each module. This design choice allows any additional implementation version to be easily plugged-in and for users to write their code once this can then leverage all modules versions. The other functions provided by *iPregel* have a single implementation across module versions as these are responsible for auxiliary aspects such as keeping track of the supersteps, or the total number of vertices in the application. However, as mentioned earlier, not all functions are defined by *iPregel*.

In addition to functions defined by *iPregel*, there are also functions declared in *iPregel* that must be defined by the user. The two major ones, listed in Figure 4.1, consist of `ip_compute` and `ip_combine`. The former contains the code to execute on each active vertex at each superstep, and the latter is called each time a vertex with a message in its mailbox receives a new message.

4.3.2.3 Vertex representation

As an in-memory shared-memory solution, *iPregel* is bound to the memory available on the single node it runs on. Therefore, reducing its memory footprint is essential. Since the cornerstone of vertex-centric models is the vertices themselves, an effort was made towards the design of *iPregel* regarding vertex representation.

In vertex-centric frameworks, a vertex contains the attributes that are specified by the user and internals that are used by the framework itself, such as the active state of the vertex. Generally, vertex-centric frameworks rely on object-oriented languages like C++ and use a base class that contains all internals. That same class also has virtual methods equivalent to the `ip_compute` and `ip_combine` functions presented in Figure 4.1. The user then derives their own class from the base class and customises it. However, due to the presence of virtual methods, a virtual table is created. This results in every vertex object carrying an additional hidden pointer, which increases the total memory footprint.

To avoid this memory overhead in *iPregel*, vertices are represented with structures. They contain arbitrary members provided by the user, as well as internals required by the framework. Due to the multi-version design of *iPregel*, vertex internals also have alternative implementations, but again this is abstracted away from the user. A macro `IP_VERTEX_INTERNALS` is used to conceal the internals of *iPregel*, where the user defines the structure `struct ip_vertex_t` and includes the macro inside, then can append any additional member needed in their application.

Having multiple possible vertex internals also enables *iPregel* to reduce its memory footprint. For instance, some applications require vertices to track both their in and out-neighbours, while some others require only the former. Following a single vertex internals design, the structure would have to assume the most conservative design and store both in and out-neighbours. By contrast, *iPregel* proposes several tailor-made internals (in only, out only, in and out) that take into account the module versions selected and the compilation flags passed from the user as explained in Subsection 4.3.2.

4.3.2.4 Graphs accepted

Following the approach of many other vertex-centric frameworks, *iPregel* expects vertex identifiers to be integral numbers, which is the case in the vast majority of graphs. In addition, *iPregel* requires vertex identifiers to be consecutive, and accepts static graphs only, which means the graph structure needs to be defined a priori and can not change during execution.

4.3.3 Implementation

The *iPregel* framework is developed in C and uses OpenMP [45] to support shared-memory concurrency. Vertex structures and the list of neighbour identifiers are stored in two, distinct, global arrays shared with all threads. The vertex workload is distributed using the default static schedule in OpenMP, which results in the total number of vertices being evenly distributed across all threads and no work-stealing strategy is required.

4.3.4 Optimisations

This subsection presents the optimisation strategies employed by *iPregel*, which follow by the overarching philosophy of *iPregel*, *not resulting in user source code rewriting, or exposing low-level details to the user*. Consequently the approach adopted has been for optimisations to be enabled explicitly via compilation flags, thus leaving the user source code unmodified.

4.3.4.1 Selection bypass

The first phase in vertex-centric frameworks consists of selecting the vertices to execute, and this is well accepted to be a tricky aspect of vertex-centric models [71]. The naive approach is to check the status of each vertex and process those that are active. However, for inactive vertices, these checks are pointless and result in wasted memory accesses. This is important because frameworks that use in-memory storage and shared-memory parallelism already place high pressure on memory bandwidth. Therefore, keeping unproductive memory accesses to a minimum prevents aggravating that pressure. The naive approach becomes especially problematic in programs that contain a small number of active vertices and many inactive ones, thus resulting in many wasted checks.

The selection phase was analysed, where this phase typically decides to run a vertex if at least one of the following conditions is met:

1. It is the first superstep
2. The vertex is already active (i.e., it did not halt when it was last processed)
3. The vertex received a message during the previous superstep

Condition 1 becomes false at the end of the first superstep. Thus, from the second superstep onwards, a vertex is active if and only if conditions 2 or 3 are met. One cannot assert which condition it is unless the algorithm exhibits a *systematic halt*, where every time a vertex is processed it halts at the end of the compute function, because

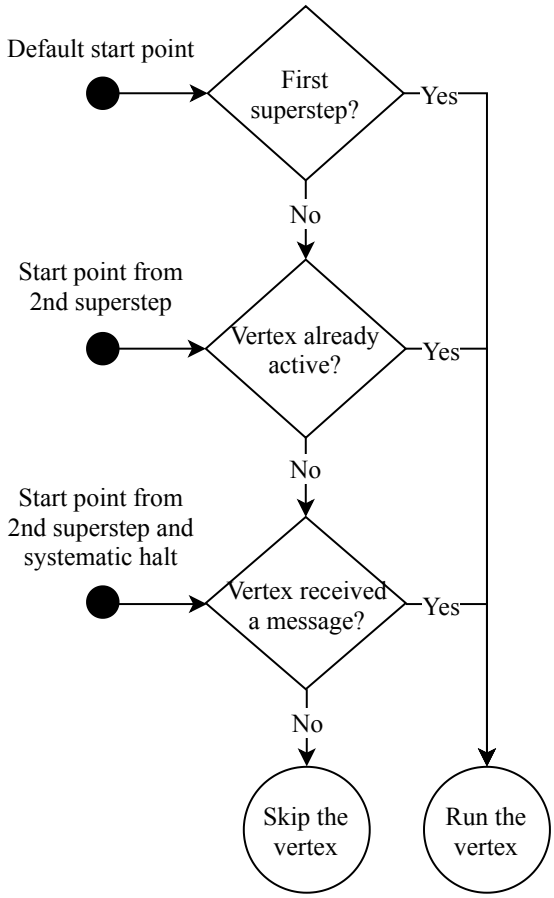

Figure 4.3: Execution flow of the vertex selection mechanism

this algorithmic particularity guarantees that condition 2 is always false. This is the case for the Connected Components and SSSP benchmarks presented in Subsections 4.4.2 and 4.4.3. By contrast, in the PageRank benchmark presented in Subsection 4.4.1, a processed vertex will not halt if the number of supersteps elapsed is less than a predefined threshold. Therefore, in PageRank a vertex may be active in superstep n without having received a message in superstep $n - 1$.

In the *systematic halt* situation, however, this configuration is not possible. Indeed since condition 1 is false after the first superstep and condition 2 is always false, only condition 3 remains where a vertex is active if, and only if, it received a message in the previous superstep. This is illustrated in Figure 4.3, and because of this property the list of active vertices for superstep $n + 1$ can be established by monitoring message exchanges during superstep n and determining which vertices are the recipients of these exchanged messages. This is why, when an algorithm exposes *systematic halt*, *iPregel* can monitor message exchanges and automatically determine which vertices are active, i.e. need processing, in the next superstep.

This approach that has been developed as part of this research is known as *selection bypass* and integrating it is straightforward as it can be embedded in the function called by vertices to send messages (*ip_send_message* and *ip_broadcast* as given in Appendix A). This modification consists of appending the recipient vertex identifier to the list of vertices that must be run during the next superstep. However, one must avoid duplicate identifiers so that a given vertex is not processed multiple times. This is again straightforward because when a vertex sends a message the worker that processes that vertex must check if the recipient vertex already has a message in its mailbox to determine whether it should apply the message combination presented in Subsubsection 4.3.4.3. From there, integrating the *selection bypass* feature consists of the worker adding the recipient vertex identifier to the list of vertices to execute during the next superstep if that recipient vertex's mailbox is empty. Multiple threads accessing the same list can result in data-races, and therefore in *iPregel* each worker maintains its own list. At the end of every superstep, these lists are merged into a single one. This list is then split evenly across all workers, which only need to process the vertices in that list, without having to check their active status or the presence of pending messages.

There are multiple benefits from the selection bypass technique that has been developed here. Firstly, not having to check the status of each vertex saves memory accesses and removes possible branch mispredictions on the vertex active state (execute if active, skip if inactive) since vertices in the merged list are known to be active. Secondly, this feature improves load balancing because threads receive exclusively vertices that are guar-

anteed to require processing. This contrasts against the naive approach, where threads may receive identical numbers of vertices, potentially containing significantly different proportions of active and inactive vertices. In summary, the selection bypass technique makes the active vertex distribution optimal with regard to the number of active vertices per worker.

The selection bypass technique introduced in this subsubsection can be seen as somewhat equivalent to frontiers in *Ligra* or selective scheduling in *GraphChi*. However a major difference is that the selection bypass allows the frontier to be automatically deduced from algorithmic analysis, rather than explicit manual user input.

4.3.4.2 Message exchange

Typically vertex-centric applications are communication-intensive, and therefore optimising the message exchange mechanism can result in substantial performance improvements.

There are two means by which a message can be transmitted; firstly the sender can push it to a recipient's mailbox, or secondly the recipient can pull it from a sender's outbox. The push version can result in race conditions in the event of multiple vertex messages being pushed to the same recipient's mailbox concurrently. In *iPregel*, this is prevented with the use of busy-waiting locks which are more efficient than their block-waiting counterparts given that the combination operation is typically very small. By contrast, the push version can be implemented without any locks if the combination operation corresponds to an atomic operation. In *Ligra* [53], the user can exploit lock-free combinations by writing a second, atomic, implementation of their combiner. However, providing this optimisation without involving additional code writing is tricky and may require a code parsing phase to determine whether a given combiner code can be atomically processed. Due to this, the design decision was taken for *iPregel* not to leverage push-based lock-free combiner operation.

The pull-based approach, due to the read-only nature of potential inter-thread interactions, has the advantage of being data-race free. Thus, threads can process message exchanges in parallel with no synchronisation required. Vertices must have a mailbox to fetch messages, as well as an outbox in which they can buffer messages to send. Each message in the outbox must be annotated with the recipient identifier, so that each out-neighbour knows which message(s) to fetch. However, this annotation approach would result in additional memory overhead, which when considering the number of messages involved would be significant over large graphs.

However, if the same value is to be sent to all out-neighbours via a broadcast, and if

vertices issue at most one broadcast each per superstep, then only one message needs to be stored in the vertex's outbox. As such, there is no need to annotate the recipient's identifier since the message is intended for every out-neighbour. It was observed that this assumption holds in numerous vertex-centric applications, given that communications are typically performed via broadcasts to neighbours; from graph analytics algorithms such as PageRank [11], Connected Components [12], Single-Source Shortest Paths [11], Breadth-First Search [53], Betweenness Centrality [72] and graph radii estimation [53]; to community detection algorithms, such as including Vertex-Coloring [73] and K-Core [74] to name a few.

However, to support this lock-free design, *iPregel* must check, for each vertex, the presence of a message in the outbox of every in-neighbour, which results in numerous memory accesses. In applications that expose a low number of active vertices, this optimisation generates a high number of memory accesses that consist of checking an empty outbox, hence wasting memory bandwidth and generating unproductive extra work. Although *Ligra* can dynamically switch between the push and pull communications at runtime via a user-defined threshold, *iPregel* must be configured whether to use the former or the latter via a compilation flag. The user must therefore determine experimentally whether it is beneficial in their case to enable this optimisation.

4.3.4.3 Message combination

Once the location of the recipient vertex is found, the message to be sent must be appended to the recipient's inbox, and this is where the combination takes place (and where the user's `ip_combine` function is called). Concretely, when a vertex sends a message, if its recipient's inbox is empty then the message is added, otherwise, it is combined with the existing message. It follows that shared-memory combiners guarantee that vertices will have *at most* one message in their inbox. This greatly reduces memory consumption and makes memory consumption more predictable, because no dynamically resizeable structure is required to contain inbox messages. From an implementation perspective, several techniques are discussed in this section. It should be noted that techniques presented in this section are independent of, and therefore compatible with, the selection bypass (see Subsubsection 4.3.4.1) and efficient vertex addressing (see Subsubsection 4.3.4.4) techniques.

Push-cased combiner The first combiner presented in this section is referred to as the *push-based combiner*. In the push-based approach, multiple vertices may send a message to the same recipient concurrently, which results in a potential data-race which must be

prevented via synchronisation. This is achieved with the use of a lock that is acquired in turn by the threads processing their the sender vertices, serialising access to the protected data (the receiver's vertex mailbox).

The most common locking technique, known as *block-waiting synchronisation*, consists of blocking the threads whilst these are waiting to acquire the lock. These blocked threads are paused and placed in a waiting queue, from which they will be removed once they have acquired the lock. By putting threads to sleep this mechanism frees up CPU resources which can then be allocated to other threads. However, this does result in some overhead because managing the queue of waiting threads requires a more advanced lock structure which, in addition to involving extra processing, also requires more memory.

This contrasts with the other common form of synchronisation known as *busy-waiting synchronisation*, where threads repeatedly attempt to acquire the lock until they eventually succeed. This technique is generally avoided because, as threads are not put to sleep, the CPU spins on attempting to acquire the lock and thus is kept busy by its thread in this phase. However, this approach does have two advantages over its block-waiting counterpart. Firstly, when the critical section[2] is very small, for instance, combiners which typically consist of a compare-and-replace operation, busy-waiting locks can be more reactive because they do not incur thread pausing and resuming overheads. Secondly, not having to handle the overheads of block-waiting (e.g. pausing, queueing, de-queueing, and resuming) makes them simpler and therefore reduces the bookkeeping compute and memory requirements.

In *gcc*, the compiler used in this work, the block-waiting and busy-waiting synchronisations are implemented with mutexes and spinlocks respectively (the latter requires GNU99 extensions). The former requires 40 bytes while the latter only 4 bytes, which represents a reduction of 90% for block-waiting. Since there is one lock per inbox and one inbox per vertex, this memory requirement is multiplied by the total number of vertices. For instance, considering the Wikipedia and United States of America (USA) graphs used in this research, which are given in Table 4.1, switching from mutexes to spinlocks reduces the memory footprint of the data-race protection from 730 and 958 megabytes respectively, to 73 and 96 megabytes. This is valuable in *iPregel* where being memory efficient, and thus able to handle larger graphs in shared memory, is a major objective.

Pull-based combiner The combination process is explained thus far in this section has assumed a push-based approach, where the sender places a message into the recipient's inbox. However, adapting this for the pull-based approach is also possible, where recipients

[2]The region of code protected by a lock.

fetch messages sent by their in-neighbours, and this is known as the *pull-based combiner*. This technique is designed upon the observation that a high number of vertex-centric algorithms use neighbouring broadcasts[3] as their unique means of communication. In other words, every time a vertex communicates, an identical value is sent to all out-neighbours. This behaviour is present, for instance, in the PageRank benchmark introduced in Subsubsection 2.3.2.1.

The Pull-based combiner requires reversing the way communications are designed and consists of three phases. Firstly, vertices must be provided with their in-neighbours' identifiers to locate the senders where they will fetch messages from. Secondly, a sending vertex must buffer the message meant for broadcast in an outbox, as well as updating its internal state to indicate that it has a message to broadcast. Thirdly, at the end of every superstep each vertex must iterate through *all* of its in-neighbours' outboxes, fetch the broadcast message (if any) and then add it to its own inbox (or combine it with an existing message).

With this technique, inter-vertex interactions are exclusively read-only (i.e: fetching messages) while write actions (i.e: combination) are only intra-vertex. Since a vertex is processed by a single thread, it follows that threads never modify the value of a vertex they do not process. Therefore, there is no risk of a data-race, meaning that that pull-based combiners provide a key improvement for parallelism performance, namely a race-free design. However, the overall benefits delivered by this approach are influenced by two factors:

1. **The ratio of active to inactive vertices** because each vertex must fetch messages from its in-neighbours at every superstep. Therefore, the more active vertices contained by in-neighbours, the fewer unproductive checks.

2. **The number of in-neighbours** because each vertex must iterate through every one of its in-neighbours. Consequently, the fewer in-neighbours, the faster this iteration will be undertaken.

Irrespective, locks are no longer needed, therefore avoiding the memory overhead needed by data-race protections.

Because vertices in *iPregel* are provided with their list of in and out-neighbours at creation, from an implementation perspective, each vertex stores a pointer to its array of in-neighbours as well as a corresponding counter, identically for out-neighbours. However, the vertex structure may contain-neighbour information that will never be used, resulting in a waste of memory. One approach would be to set the unused counters to 0 and the

[3]By opposition to graph-wise broadcast.

unused pointers to null, however, storing unused pointers and counters wastes several bytes per vertex, surging to a total of approximately 250MB[4] for a 20 million vertex graph, such as those used for experimentation later in Subsubsection 4.5.1.4. It was therefore decided to indicate needed neighbour information through compilation flags such as `IP_NEEDS_OUT_NEIGHBOUR_IDS`, which enables *iPregel* to select the lightest structure to represent a vertex and keep its memory footprint to a minimum.

Single message mailboxes With the use of combiners, vertex mailboxes can have two states, empty or containing one message. Upon receipt of a new message, an empty mailbox stores it as-is, whereas a mailbox with an existing message combines the new message and existing one together. Irrespective, at most one message is contained within a vertex's mailbox. This allows the mailbox in *iPregel* to be of size one message only, and avoid the use of dynamically resizeable data structures as well as the memory overhead they incur. As a result, this design is a major factor in reducing the memory requirements of *iPregel*.

4.3.4.4 Efficient vertex addressing

Vertex-centric models typically involve a high number of communications between vertices, hence the importance of quick message delivery. This section focuses on the first step of message delivery, finding the recipient vertex.

The vertex addressing mechanism is conventionally achieved with hashmaps matching vertex identifiers against their locations. However, this intermediate layer in the vertex addressing process incurs additional memory accesses, increases the memory footprint and exposes poor data locality that is inherent to hashmaps. In-memory shared-memory solutions such as *iPregel* typically store all vertices in a single array, so the location of a vertex is its corresponding index in that array. Based upon the observation that most graphs use integral numbers as vertex identifiers, the approach adopted in this research proposes to semantically enrich vertex identifiers so they also represent their vertex location.

The first strategy presented in this section is called *Direct Mapping*, where vertices are stored in the global array at the index equal to their identifier. For example, a vertex with identifier 5 resides at index 5 in the vertex array. This approach provides an overhead-free addressing mechanism but requires identifiers to start at 0 since *iPregel* is developed in C which is 0-indexed.

[4]Assuming a 64-bit operating system, hence 8-byte pointers, and an unsigned int counter, being 4-byte long in most implementations.

However, it may be the case that vertex identifiers start at an arbitrary number, the use of an offset must therefore be considered, resulting in an *Offset Mapping*. This offset is then subtracted from the vertex identifier to find the corresponding location. Consequently, this offset provides an index-identifier matching consisting of a simple subtraction, which involves a marginal overhead only.

Direct mapping can still be used in situations requiring offset mapping, with the technique named *Desolate Memory*. By forcing direct mapping, vertices will reside at the array index matching their identifier. Since in this case, vertex identifiers contain an offset, the array elements whose indexes are lower than the actual offset will be unused, resulting in wasted memory. However, for graphs whose indexes start above 0, for instance at 1, using desolate memory incurs the waste of a single element, which can be argued is a reasonable memory sacrifice to benefit from direct mapping. Furthermore, regardless of the number of elements unused, certain operating systems such as Linux provide what is referred to as a first-touch policy, where the actual memory allocation for data is deferred until it is accessed for the first time. Therefore, with this policy, which is enabled by default on Linux; elements residing at indexes below that of the lowest vertex identifier for instance, which are never accessed, will not directly[5] trigger an actual memory allocation.

The addressing mechanism presented in this section allows for an efficient mapping between a vertex identifier and its location in memory. In addition, every addressing technique presented in this section can be used in conjunction with the selection bypass mechanism introduced in Subsubsection 4.3.4.1.

4.4 Benchmarking applications

In this research, as explained in Subsection 2.3.2, frameworks are evaluated across three applications widely used as benchmarks in the vertex-centric community: PageRank, Connected Components and Single-Source Shortest Paths.

4.4.1 PageRank

As introduced in Subsubsection 2.3.2.1, the PageRank algorithm is designed to order web pages based on their importance calculated from the number of hyperlinks pointing to them.

The *iPregel* implementation of a PageRank algorithm presented in Figure 4.4 is based upon the original *Pregel* version introduced in [11]. During the first superstep, each

[5]although they might, indirectly, still have a corresponding memory allocated, through the paging system, due to the allocation from another vertex close in memory

vertex begins with an initial PageRank value of one divided by the number of vertices and broadcasts its PageRank value divided by its number of out-neighbours. From the next superstep onwards, each vertex sums the PageRank values received from its in-neighbours, and then it updates its current PageRank value and broadcasts this. This is repeated for a predefined number of supersteps, after which vertices halt and the execution terminates. In practice, however, a PageRank application would typically run until some convergence criteria is reached.

As explained above, when a vertex receives messages, it sums their values, and this operation is both associative and commutative. Therefore a combiner can be used as explained in Subsubsection 4.3.4.3. In Figure 4.4, the reader can observe that implementing this combiner requires limited code changes by the user, they must just define the `combine` function and write a short code representing the operation to apply. Furthermore, communications performed during the PageRank calculations consist exclusively of broadcasts, with a maximum of one broadcast per vertex per superstep. According to Subsubsection 4.3.4.2, this property means that PageRank is compatible with the pull-based communication model, which can be enabled in *iPregel* via a compilation flag. However since vertices halt only after a certain number of supersteps, compared to halting at every superstep, PageRank is not compatible with the *iPregel* selection bypass optimisation presented in Subsubsection 4.3.4.1.

It should be noted that PageRank experiments reported in this chapter are run over 30 iterations.

4.4.2 Connected Components

As described in Subsubsection 2.3.2.2, computing the Connected Components of a graph consists of finding all disjoint subsets of that graph such that each subset comprises only of vertices that can reach each other. There are several possible vertex-centric algorithms to compute Connected Components, the one selected in *iPregel* is often referred to as Hash-Min. This relies upon the propagation of vertex identifiers to locate, for each vertex, the minimum vertex identifier reachable. This computation converges and thus the algorithm terminates when vertices find the minimum vertex identifier they can reach. Finally, vertices having reached the same minimum vertex identifier belong to the same connected component.

The *iPregel* implementation, illustrated in Figure 4.5, begins with vertices initialising their value to their own vertex identifier, before broadcasting this to all out-neighbours. Subsequently, vertices identify the minimum vertex identifier received from their in-neighbours. They then update their value if the minimum vertex identifier obtained

```
1  const int ROUNDS = 10;
2  void ip_compute(struct ip_vertex_t* me) {
3      if(ip_is_first_superstep()) {
4          me->value = initial_value;
5      }
6      else {
7          IP_MESSAGE_TYPE sum = 0.0;
8          IP_MESSAGE_TYPE msg;
9          while(ip_get_next_message(me, &msg)) {
10             sum += msg;
11         }
12         msg = ratio + 0.85 * sum;
13         me->value = msg;
14     }
15     if(ip_get_superstep() < ROUNDS) {
16         ip_broadcast(me, me->value / me->out_neighbour_count);
17     }
18     else {
19         ip_vote_to_halt(me);
20     }
21 }
22 void ip_combine(IP_MESSAGE_TYPE* old, IP_MESSAGE_TYPE new) {
23     *old += new;
24 }
```

Figure 4.4: PageRank implemented in *iPregel*

is smaller, in which case this is then broadcast to the out-neighbours to continue the propagation. Since vertices may obtain the minimum vertex identifier reachable at any superstep, they always halt at the end of a superstep.

This broadcast characteristic means that the Connected Components implementation described here is compatible with the pull-based communications introduced in Subsubsection 4.3.4.2. Moreover, the combination operation which is applied to incoming messages, finding the minimum value, is associative and commutative. Therefore, combiners can be leveraged for Connected Components and, similarly to the PageRank combiner implementation of Figure 4.4, the Connected Components equivalent is also a *combine* function comprising a couple of lines of code. Furthermore, because vertices halt at the end of every superstep, Connected Components is also suitable for the selection bypass optimisation presented in Subsubsection 4.3.4.1

4.4.3 Single-Source Shortest Paths

The SSSP algorithm, described in Subsubsection 2.3.2.3, consists of selecting a vertex as the source and finding the distance between this source vertex and every other vertex in the graph. In the version of this benchmark used here it is assumed that all edge weights

```
1  void ip_compute(struct ip_vertex_t* me) {
2      if(ip_is_first_superstep()) {
3          me->value = me->id;
4          ip_broadcast(me, me->value);
5      }
6      else {
7          IP_MESSAGE_TYPE old_value = me->value;
8          IP_MESSAGE_TYPE msg;
9          while(ip_get_next_message(me, &msg)) {
10             me->value = min(me->value, msg);
11         }
12         if(me->value < old_value) {
13             ip_broadcast(me, me->value);
14         }
15     }
16     ip_vote_to_halt(me);
17 }
18 void ip_combine(IP_MESSAGE_TYPE* old, IP_MESSAGE_TYPE new) {
19     if(*old > new) {
20         *old = new;
21     }
22 }
```

Figure 4.5: Connected components implemented in *iPregel*

equal 1.

The *iPregel* implementation sketched in Figure 4.6 is based upon the original *Pregel* version introduced in [11], which is considered a distributed version of the Bellman-Ford algorithm [54] and is also the implementation used in *Ligra*. During the first superstep, the source vertex initialises its value to 0 and begins the propagation by broadcasting this incremented by 1 (representing the assumed edge weight of 1). Concurrently, all other vertices initialise their value to INF (which is a value greater than the longest distance possible in the graph). From the second superstep onwards, vertices calculate the potential minimum distance obtained from messages received. In the event this distance is smaller than the current vertex value, the vertex updates its value and broadcasts this incremented by 1. Finally, vertices halt at the end of every superstep since they may have obtained their final minimum distance at any superstep.

This SSSP algorithm exposes the same characteristics as Connected Components, where vertices halt at the end of every superstep, communications are performed only via broadcasts, there is a maximum of one broadcast per vertex per superstep, and the algorithm contains a combination operation that is associative and commutative, which calculates the minimum distance. As a consequence, the SSSP implementation can be optimised using the selection bypass technique presented in Subsubsection 4.3.4.1, the pull-based communications discussed in Subsubsection 4.3.4.2 and the leverage of

```
1  void ip_compute(struct ip_vertex_t* me) {
2      if(ip_is_first_superstep()) {
3          if(me->id == SSSP_SOURCE) {
4              me->value = 0;
5              ip_broadcast(me, me->value + 1);
6          }
7          else {
8              me->value = INF;
9          }
10     } else {
11         IP_MESSAGE_TYPE mindist = INF;
12         IP_MESSAGE_TYPE msg;
13         while(ip_get_next_message(me, &msg)) {
14             mindist = min(mindist, msg);
15         }
16         if(mindist < me->value) {
17             me->value = mindist;
18             ip_broadcast(me, me->value + 1);
19         }
20     }
21     ip_vote_to_halt(me);
22 }
23 void ip_combine(IP_MESSAGE_TYPE* old, IP_MESSAGE_TYPE new) {
24     if(*old > new) {
25         *old = new;
26     }
27 }
```

Figure 4.6: Unweighted SSSP implemented in *iPregel*

combiners introduced in Subsubsection 4.3.4.3 respectively.

Note that benchmark implementations from other frameworks were amended, as required, to become algorithmically equivalent such as homogenising the predefined iteration number in PageRank for instance.

It should be noted that the SSSP experiments contained in this chapter use the vertex identified by '2' as the source.

4.5 Assessing in-memory shared-memory viability

The first series of experiments conducted in this chapter focus on the competitiveness of in-memory shared-memory vertex-centric frameworks and determining whether they are a viable solution for graphs processing. Two sets of experiments are described, the first evaluating the performance of several *iPregel* versions, and then secondly comparing these against an external vertex-centric framework.

4.5.1 Experimental setup

4.5.1.1 Framework considered

Although *FemtoGraph* is the most similar framework to *iPregel*, namely in-memory shared-memory vertex-centric, a comparison cannot be made because the former does not yield correct results. Therefore, the comparison must include a framework that has an architecture different from that of *iPregel*. In other words, either out-of-core computation or distributed-memory parallelism must be allowed. It has been decided to compare against the state-of-the-art in-memory distributed-memory system, the vertex-centric framework *Pregel+*, because despite suffering from network communications, this benefits from additional memory and processing power.

4.5.1.2 Computing environment

Experiments are run on Amazon EC2 using *m4.large* instances, which provide 8GB of Dual In-line Memory Module (DIMM) memory, 2 cores of an Intel(R) Xeon(R) CPU E5-2686 v4, clocked at 2.30GHz, and a maximum network bandwidth of 450Mbps. Instances are set up with Ubuntu 16.04.3 Long-Term Support (LTS) 64-bit operating system. The objective behind this setup was to target a machine that has a performance comparable to that of a modern laptop, supporting the argument that such experiments can be run on reasonably affordable hardware.

iPregel is compiled with `gcc` version 5.4.0, using C99 standard by default and GNU99 extensions when using spinlocks (see Subsubsection 4.3.4.3). The optimisation level is set to `-O2`, and to exploit both cores available on the EC2 instance, two OpenMP threads are used.

Pregel+ is compiled with `mpic++` (MPICH version 3.2), using `g++` version 5.4.0 with C++11 standard as the underlying C++ compiler. The optimisation level is set to `-O2`, and to exploit both cores available on each EC2 instance, two MPI processes are created per node.

4.5.1.3 Methodology

The experiments are initially run five times and are repeated until the margin of error that is obtained represents less than 1% of the average runtime, given a confidence level of 99%. The timings collected report superstep execution time, that is, graph preprocessing and graph loading are not included. Memory consumption also measured, represented by the *maximum resident set size*[6] as returned by the bash command `time -v`.

[6]This stands for the maximum amount of memory taken by a program throughout its execution.

Table 4.1: Graphs used in the comparison with *Pregel+*

Name	Graph order	Graph size	Average degree
Wikipedia	18,268,992	172,183,984	9.42
USA Road network	23,947,347	58,333,344	2.44

4.5.1.4 Graphs used

The experiments presented in this section are conducted on the graphs presented in Table 4.1, namely the Wikipedia graph[7] at KONECT [75], and the USA roads graph[8] at DIMACS [76].

The two graphs being used in the experiments are actual, real-world graphs that are accessible to the public. This characteristic enables comparisons to be made with prior research findings, as the graphs have already been featured in existing literature [12]. Moreover, despite having comparable graph orders, the density of the graphs varies significantly, with one being nearly four times denser than the other. This difference provides an opportunity for the experiments to impose various levels of stress on the message processing and combination mechanism. Consequently, the outcomes produced by the experiments are representative of a broader range of graphs, rather than being limited to a specific type. The flexibility of the graph density also permits a wider range of experiments to be conducted on these graphs, allowing researchers to explore different approaches to message processing and combination, and determine which ones work best under varying conditions.

The graphs are constructed with consecutive indexes that commence at 1 and are processed by *iPregel* using offset mapping along with desolate memory, as described in Subsubsection 4.3.4.4.

4.5.2 Results

4.5.2.1 Performance of iPregel Versions

The experiments presented in this section intend to evaluate the performance impact of the selection bypass and combination techniques presented in Subsubsections 4.3.4.1 and 4.3.4.3. Each of the three benchmarks presented in Subsection 2.3.2 is executed using every compatible version of *iPregel*. There are in total six versions possible for CC and SSSP: three combiners (see Subsubection 4.3.4.3) which can be used with or

[7] http://konect.uni-koblenz.de/networks/dbpedia-link
[8] http://www.dis.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.USA.gr.gz

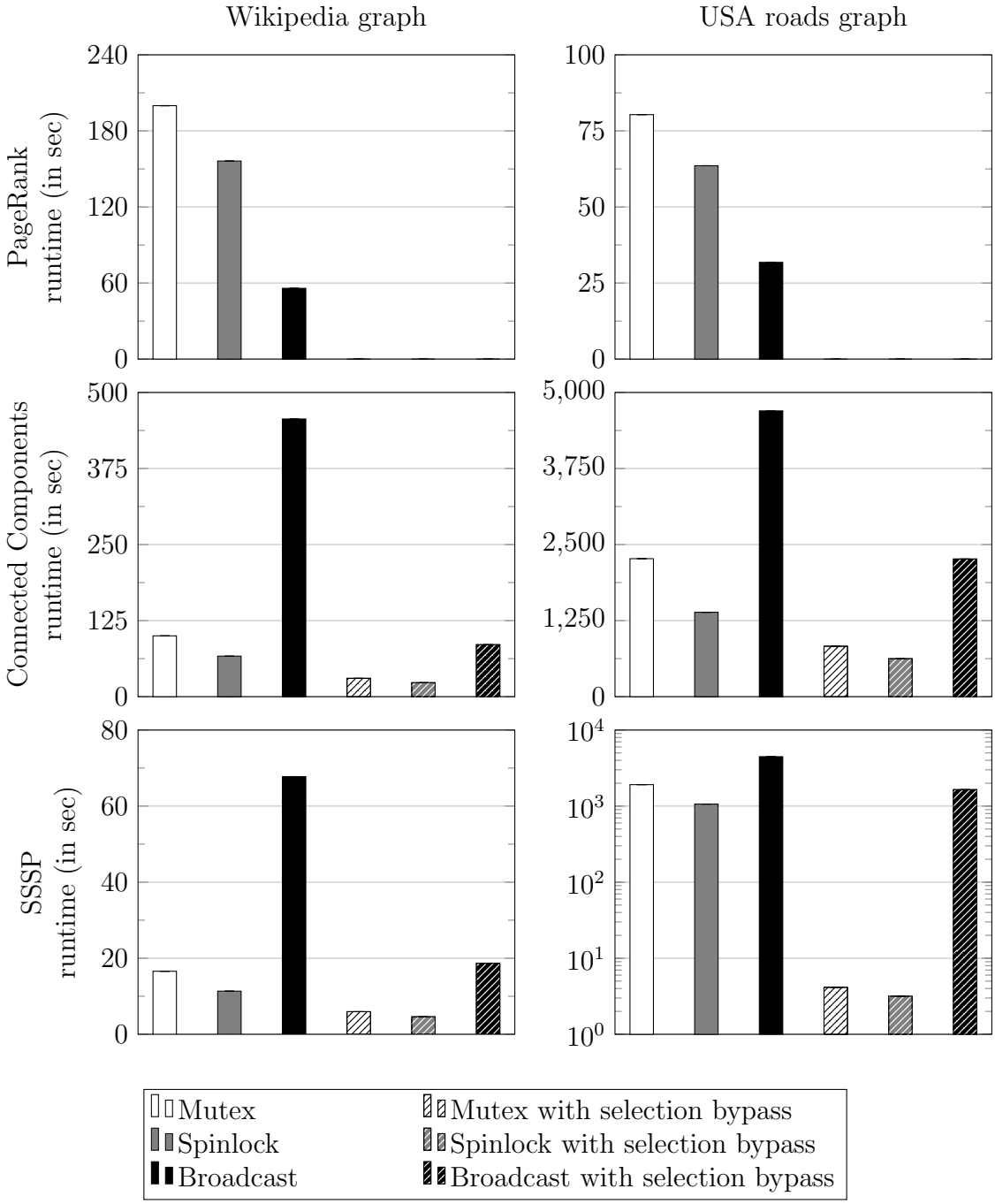

Figure 4.7: Runtime (in seconds) of *iPregel* on PageRank, CC and SSSP as the version varies

without the selection bypass (see Subsubsection 4.3.4.1). For PageRank, only the three versions without selection bypass are implemented. CC and SSSP are implemented in all six versions. The execution times of these experiments are reported in Figure 4.7.

On both graphs, we observe that PageRank execution times drop by approximately 30% between the mutex and spinlock versions. It is the broadcast version however which benefits most by moving to spinlocks as this halves the execution time and is the only version capable of processing 30 PageRank iterations in under a minute.

The runtimes of CC and SSSP are similar in that broadcast versions are slower than mutexes, themselves slower than spinlocks. In addition, all combiner versions become faster when they exploit the selection bypass. Therefore, the optimal version is always the spinlock combiner with selection bypass and the worst is always the broadcast version without selection bypass. By moving from the Wikipedia to the USA roads graph, the speedup between these two fastest and the slowest versions increases from 7.5 seconds to 20 seconds for CC, and from 15 seconds to 1,400 seconds for SSSP.

To understand these differences two factors must be considered, the ratio of active vertices compared to the total number of vertices, and the graph's density.

This first factor directly impacts the performance of the broadcast version (implemented via pull-based combiner) as explained in Subsubsection 4.3.4.3. PageRank offers an optimal ratio since all vertices stay active during the entire execution time. By comparison, CC and SSSP expose lower ratios, continuously decreasing and constantly low respectively. This first factor explains why the broadcast version performed well in PageRank and poorly for both CC and SSSP.

The second factor is the key to explaining the significant increase in SSSP performance. A lower density means a smaller average out-degree, which results in a slower propagation of messages, thus a high number of supersteps to completely reach a graph. In SSSP, the number of active vertices is low overall, beginning at one, increasing in the early phase of the propagation, before decreasing in the late phase of propagation; which is suitable for the selection bypass. As shown in Table 4.1, the density of the USA road network graph is nearly four times lower than the Wikipedia graph's. Combined with the low number of active vertices in SSSP, this lower density explains the performance difference when using selection bypass or not. CC reaches a very low number of active vertices only in the late supersteps, which partially mitigates the benefits of a lower graph density.

In this first round of experiments the broadcast version is the fastest for PageRank, as is the spinlock with selection bypass version for both CC and SSSP. Having identified these optimal configurations, it is now instructive to compare these against the state-of-the-art in-memory distributed-memory solution: *Pregel+*.

4.5.2.2 Comparison against Pregel+

As explained in Subsubsection 4.5.1.1, the distributed-memory framework *Pregel+* is selected for comparisons. However one must be careful when comparing shared-memory and distributed-memory solutions on a single node, as this can disadvantage the distributed-memory solution because it will have some overhead for distributed parallelism. Comparing over multiple nodes disadvantages the shared-memory solution because it can not take advantage of these. As a consequence, experiments presented in this subsection are twofold. Firstly, they compare the performance of *Pregel+* and *iPregel* on a single node. Secondly, they determine the number of nodes required by *Pregel+* to outperform *iPregel*, referred to as *lead change* in the rest of this chapter; by scaling up *Pregel+*, providing it with two additional cores for every new node included while *iPregel* remains single-node throughout all experiments due to its shared-memory architecture.

Results presented in this section are collected from experiments that are run with a maximum of 16 nodes. The *lead change* may not always be observed within this interval, in which case extrapolation is used by assuming the efficiency between 8 and 16 nodes to stay constant every time the number of nodes is doubled[9]. The same extrapolation method is used backwards to estimate the runtimes for the number of nodes under which *Pregel+* fails to complete due to insufficient memory. The timings that have been collected are presented in Figure 4.8.

Across both graphs, the execution time of *Pregel+* on PageRank remains stable at approximately 200 seconds. Conversely, that of *iPregel* decreases by 43%, from slightly less than on the Wikipedia graph to about 30 seconds with the USA road graph. The *lead change* occurs at 11 nodes on the Wikipedia graph and is estimated at 30 nodes for the USA graph.

To analyse the timings obtained for PageRank, it must first be considered that the *iPregel* version used here implements the pull-based combiner that was introduced in Subsubsection 4.3.4.3. As it was explained in Subsubsection 4.5.2.1, the characteristics of PageRank mean that it is optimal for the pull-based combiner, making *iPregel* several times faster than *Pregel+*; by a factor of 3.57 and 6.47 on Wikipedia and USA graphs respectively.

Furthermore, the runtime obtained from *iPregel* to process PageRank on the USA roads graph is nearly half that on the Wikipedia graph. As explained in Subsubsection 4.3.4.3, the number of in-neighbours is the second performance factor of pull-based combiner. Therefore, the variation of the number of in-neighbours (almost four times

[9]Given an efficiency of x between 8 and 16 nodes, the runtime of 32 nodes is projected assuming an efficiency of x between 16 and 32 nodes.

lower on the USA roads graph) allows the pull-based version of *iPregel* to yield the performance improvement observed.

The experiments involving the SSSP benchmark exposes two particularities: insufficient memory failures and the biggest difference between the performance of *Pregel+* and *iPregel*. On the Wikipedia graph, it can be seen that *iPregel* is approximately 7 times faster than *Pregel+*, with almost 5 seconds compared to 33 seconds. This difference is increased by an order of magnitude on the USA graph, where the runtime of *Pregel+* increases by more than 560% to reach 221 seconds while that of *iPregel* falls by 30% to approximately 3 seconds; making it 69.7 times faster than its distributed rival. Although the *lead change* is reached at 13 nodes for the Wikipedia graph, it is estimated that it would require more than 15,000 nodes for the USA graph. The low number of active vertices and the low graph density of the USA graph provide optimal conditions to exploit the potential of selection bypass.

The third set of experiments were run on CC, and contain the longest execution times observed of almost one hour. Although *iPregel* and *Pregel+* process the Wikipedia graph in less than 25 and 150 seconds respectively, their runtime increases to more than 10 and 50 minutes respectively[10] for the USA road network graph. The *lead change* remains constant, however, requiring 11 nodes for both graphs.

The interesting feature of CC is its variation of the number of active vertices throughout the computation, where the algorithm starts with all vertices active, and progressively these then halt as computation proceeds. Consequently, as most of the execution time is spent on supersteps containing a medium number of active vertices, this algorithm does not allow the selection bypass technique to reach its full potential as was the case with SSSP. Furthermore, the low density of the USA graph significantly slows down the propagation of the deactivation, which consequently increases the runtime. CC is the only application in which the speedup between *iPregel* and *Pregel+* decreases from the Wikipedia to the USA graph. Indeed, while *iPregel* is 6.5 times faster than *Pregel+* on the former, it is only 5 times faster on the latter.

Across all experiments, *iPregel* outperforms *Pregel+* on a single node, where the former was naturally at an advantage due to its shared-memory design. However, at least 10 additional nodes are needed by *Pregel+* to outperform *iPregel*. Certain configurations such as SSSP on the USA graph contain too few active vertices during too many supersteps and this makes it impossible for *Pregel+* to outperform *iPregel* within a reasonable number of nodes.

[10]Exact results are 624.13 and 3,065.03 seconds.

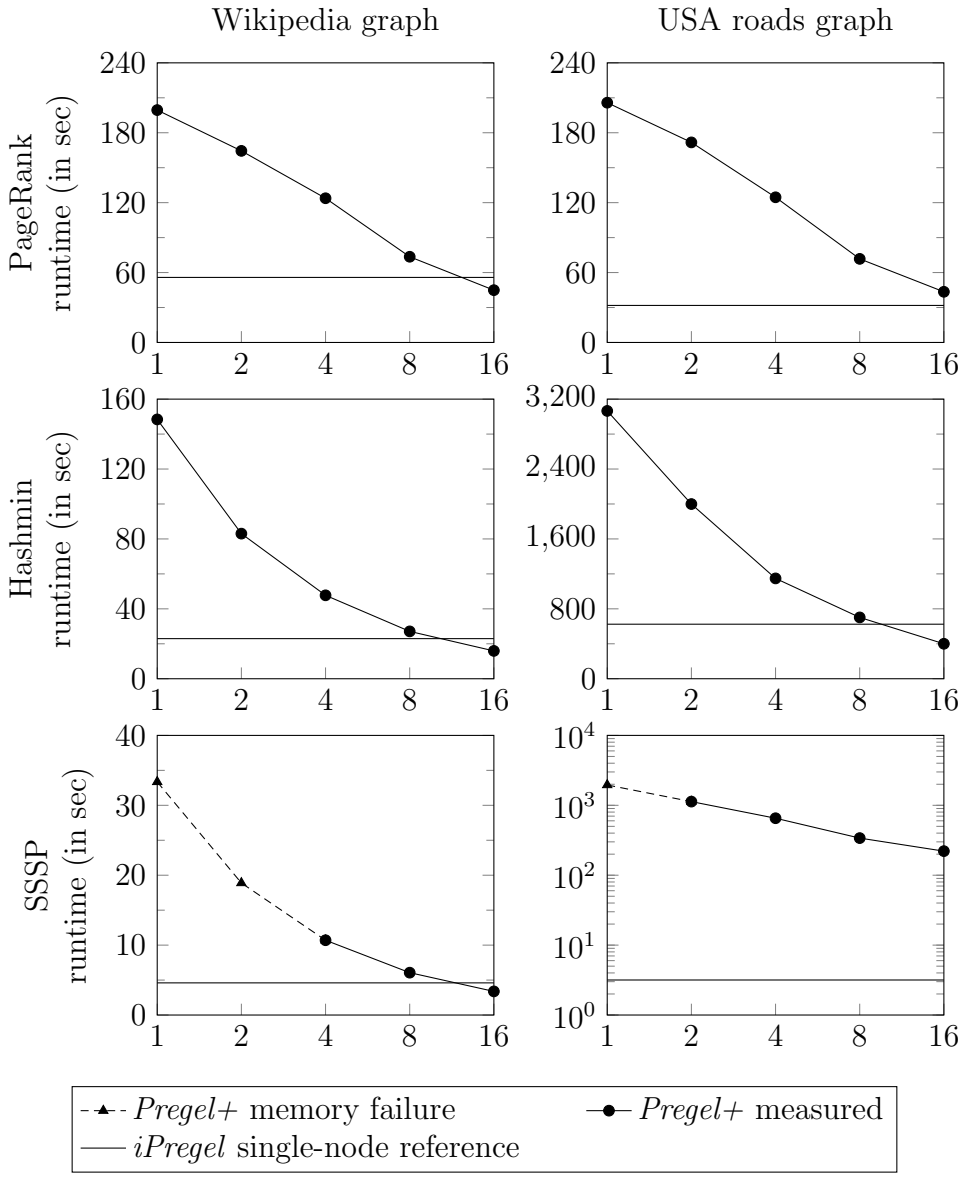

Figure 4.8: Variation of the *Pregel+* runtime (in seconds) of PageRank, Hashmin and SSSP as the number of nodes varies

Table 4.2: Graphs used for further *iPregel* memory footprint experiments

Name	Graph order	Graph size
Twitter (MPI)	52,579,682	1,963,263,821
Friendster	68,349,466	2,586,147,869

4.5.2.3 Memory footprint

The memory footprint of vertex-centric models is a known weakness [67]. Yet, reducing the memory requirements is crucial to in-memory shared-memory frameworks like *iPregel*. Indeed, being lighter on memory requirements increases the number of vertices and edges that can be processed using a specific amount of memory. This is why the memory footprint of *iPregel* was also measured during the experiments presented in Subsection 4.6.1.

For benchmarks involving the Wikipedia graph, both mutex versions (with and without selection bypass) required 2GB of memory, whilst their spinlock counterparts needed 1.5GB. However, the use of selection bypass increased the memory footprint of the broadcast version from 1.5GB to 2.5GB. This is due to the out-neighbours information that is needed by the selection bypass on top of the in-neighbours information required by the pull-based combiner used in the broadcast version. For the USA graph, it was observed that the memory consumption of all versions increased by 10% compared to the Wikipedia graph. This is because vertices require more memory than edges which are typically just integers. In this case, between the Wikipedia and USA graphs, 100 million fewer edges do not compensate for the 5 million additional vertices in the USA graph.

Overall, between 1.5GB and 2.8GB of memory were necessary for *iPregel*, out of a total of 8GB available.

Throughout experiments presented in this chapter, *iPregel* used at most 35% of the 8GB available in memory. To estimate the maximal size of graphs that *iPregel* can process with a node containing 8GB, additional experiments were conducted. The Twitter (MPI) graph presented in Table 4.2 is a KONECT [75] graph publicly available[11]. This graph is selected due to existing results describing the memory footprint of *Pregel+* and *GraphLab* on this same graph. It must be reminded that the memory footprint of in-memory frameworks includes that of the graph itself. In order to distinguish the graph from the memory overhead generated by the framework, the graph binary size is calculated. This takes into account that vertices store their identifier as well as those of their out-neighbours, and assumes 4-byte vertex identifiers. However, it excludes information

[11] http://konect.uni-koblenz.de/networks/twitter_mpi

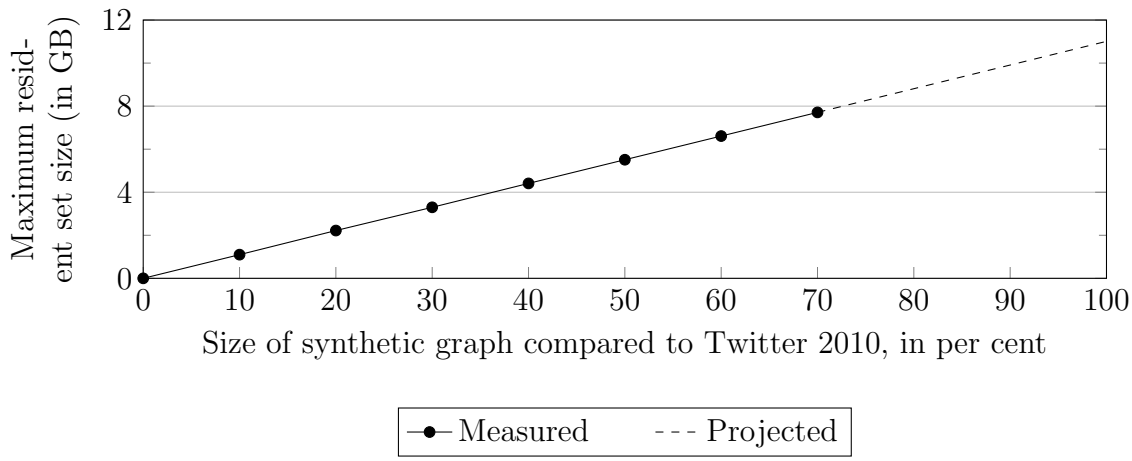

Figure 4.9: Variation of the *iPregel* maximum resident set size (in GB) to execute PageRank against the size of synthetic Twitter graph used

specific to vertex-centric applications (such as the rank value in PageRank) and internals required by frameworks, which are considered as part of the total memory overhead. The binary size of the Twitter graph is calculated to be 8GB; it follows that *iPregel* cannot process this graph with 8GB of RAM. Instead, an incremental approach was used to determine the breaking point of *iPregel* with 8GB of memory. Concretely, several synthetic graphs were generated[12], with a number of vertices and edges proportional to the original Twitter graph. A synthetic graph which is described as 20% contains a fifth of the number of vertices and a fifth of the number of edges of the original Twitter graph. PageRank was then run by *iPregel* on each of the synthetic graphs, from the smallest to the largest, until the experiments exhausted available memory. The results obtained are reported in Figure 4.9. Up to 70% of the Twitter graph can be processed before memory failure occurs. Therefore, *iPregel* is able to run PageRank on a graph comprising 37 million vertices and 1.4 billion edges under 8GB of memory.

However, existing results reported in [25] about *Pregel+* and *Giraph* consider PageRank run on the entire graph. For *iPregel*, the linear extrapolation drawn in Figure 4.9 indicates that 11GB would be required. To verify this, a new Amazon EC2 instance was deployed, the *m4.xlarge*, which contains 16GB of memory. PageRank was then run on a synthetic graph with a size identical to that of the original Twitter graph. In total, *iPregel* needed 11.01GB to execute PageRank on the complete graph, compared to *Pregel+* which requires 109GB and *Giraph* which needs 264GB.

The memory footprint of *iPregel* is therefore 10 times smaller than that of *Pregel+*

[12]The out-degree distribution is not preserved but this has no impact on the size of the graph or the memory footprint of *iPregel*

and 25 times smaller than that of *Giraph*. Excluding the 8GB allocated to the graph, out of the 11GB taken by *iPregel*, 3GB are due to overheads imposed by the framework and how it stores the graph. Comparatively, the overheads of *Pregel+* and *Giraph* are 101GB and 256GB respectively; equivalent to 33 and 85 times that of *iPregel*.

Another experiment was conducted to estimate the biggest graph that *iPregel* could process when run on a node containing 16GB of RAM. To that end, two online graph collections KONECT [75] and SNAP [77] were parsed and the largest graph available overall was selected. It turns out to be the Friendster graph from KONECT, publicly available[13]. As reported in Table 4.2, this graph contains approximately 70 million vertices and 2.5 billion edges. These results reveal that *iPregel* can process PageRank on the Friendster graph with 14.45GB of memory. Therefore, *iPregel* is able to process a multi-billion edge graph in under 16GB of RAM.

The difference observed in memory footprints between the frameworks can be explained by two factors. Firstly, there are advantages inherent to the in-memory shared-memory design. For instance, shared-memory systems manage local communications only. This contrasts with distributed-memory systems, in which messages between remote vertices are passed over the network, and typically involving the storage of sending and receiving buffers such as in *Pregel+*. Additionally, for the receiver node to know how to dispatch the messages that have been received to their individual target vertex, messages are wrapped with the vertex identifier of the recipient vertex. This results in larger messages, and hence a memory overhead. Another advantage of the shared-memory structure is that it avoids the storage of redundant information. Frameworks that are exclusively distributed-memory based exploit intra-node parallelism by creating multiple distributed workers per node. This leads to many instances of the application and distributed software environment being stored in the memory of every node. These redundant copies, therefore, waste memory. Finally, frameworks that rely on distributed-memory, or out-of-core computation, manage vertices that may reside in memory, on disk, or on a remote node. They must therefore use an additional addressing layer which stores where each vertex currently resides, and this approach also increases the overall memory footprint.

In addition to these inherent shared-memory benefits, the design adopted in *iPregel* also aims to minimise memory overhead. The use of combiners, as explained in Subsubsection 4.3.4.3, limits the memory requirements of each vertex's mailbox to one message only. This avoids the use of dynamically resizeable structures, such as queues, replacing them instead with a single variable of the message type. Furthermore, the

[13] http://konect.uni-koblenz.de/networks/Friendster

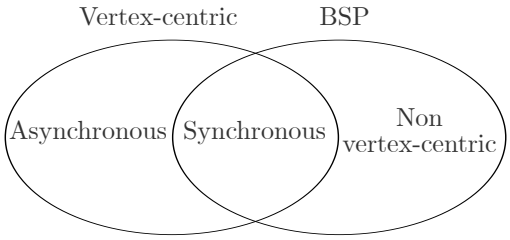

Figure 4.10: Comparison of vertex-centric and BSP models of computation.

memory overhead of the data-race protection mechanism can be reduced to zero when using the lock-free structure provided by the pull-based combiner presented in Subsubsection 4.3.4.3. The multi-version design of *iPregel* also plays an important role in its overall memory footprint. Indeed, by selecting at compile-time the appropriate structures to use, *iPregel* does not include vertex attributes that would be unused or left empty, such as in-neighbours. Finally, as explained in Subsubsection 4.3.2.3, the use of structures, as opposed to C++ objects, in *iPregel* avoids the hidden virtual pointer that is embedded in each vertex object when using derived classes like in *Pregel+*.

4.6 Evaluating the complete triptych

The second series of experiments conducted in this chapter aims to evaluate the programmability of *iPregel* against performance and memory footprint. Given that the viability of shared-memory vertex-centric frameworks against distributed-memory has been demonstrated in Section 4.5, this section focusses solely on shared-memory frameworks. This allows experiments to help more accurately assess the competitiveness of *iPregel* regarding the triptych considered in this research.

4.6.1 Experimental setup

4.6.1.1 Frameworks considered

Figure 4.10, which is taken from [54], highlights three types of frameworks that can be considered. Among these categories, *iPregel* belongs in the overlapping area as it combines vertex-centric programming and synchronous execution. To provide a comparison of *iPregel* against a variety of frameworks, the framework commonly considered as the reference in each category has been selected.

GraphChi The first vertex-centric framework to leverage out-of-core computation [24], *GraphChi*, belongs to the category of vertex-centric frameworks that exploit asynchronous

```
1  void update(graphchi_vertex<VertexDataType,EdgeDataType> &v,
2              graphchi_context &ctx) {
3      if(ctx.iteration == 0) {
4          set_data(v, vertex_values[v.id()]);
5          if(v.id() == SOURCE_VERTEX) {
6              for(int j = 0; j < v.num_outedges(); j++) {
7                  ctx.scheduler->add_task(v.outedge(j)->vertex_id());
8              }
9          }
10     }
11     else {
12         vid_t curmin = v.get_data();
13         for(int i = 0; i < v.num_inedges(); i++) {
14             if(curmin > neighbor_value(v.inedge(i))) {
15                 curmin = neighbor_value(v.inedge(i));
16             }
17         }
18         if(curmin < v.get_data() 1) {
19             curmin++;
20             set_data(v, curmin);
21             for(int i = 0; i < v.num_outedges(); i++) {
22                 if(curmin<neighbor_value(v.outedge(i))-1) {
23                     ctx.scheduler->add_task(v.outedge(i)->vertex_id());
24                 }
25             }
26         }
27     }
28 }
```

Figure 4.11: Compute function for SSSP in *GraphChi*

execution, shown on the left in Figure 4.10. For a graph that can fit entirely in memory, the out-of-core nature of *GraphChi* makes comparisons with an in-memory framework, such as *iPregel*, unfair. However, *GraphChi* also provides in-memory-only implementations[14] of its algorithms, which are automatically chosen by *GraphChi* when memory allows. Therefore, when running a *GraphChi* application, based on the amount of RAM available, as specified by the user via a runtime parameter; *GraphChi* estimates the amount of memory required for its in-memory version, and selects this if the memory available is sufficient. The experiments presented in this chapter use the in-memory version of the implementations provided by *GraphChi*, the exception being the Single Source Shortest Path (SSSP) for which no implementation is provided. We therefore developed an in-memory implementation for SSSP, given in Figure 4.11.

Another particularity of *GraphChi* is its asynchronous execution flow, where vertex updates are immediately visible, unlike its synchronous counterparts where updates take effect only in the following superstep. The advantage of the former is to reach convergence

[14]Available at https://github.com/GraphChi/graphchi-cpp

faster, while the latter is easier to reason about by providing clearer semantics.

FemtoGraph *FemtoGraph* [38] is a shared-memory vertex-centric framework that exclusively uses in-memory storage and synchronous execution. It therefore belongs to the middle category shown in Figure 4.10, like *iPregel*. However, *FemtoGraph*[15] is designed and hard-coded for PageRank, in which all vertices are run at every superstep. As a consequence, *FemtoGraph* does not implement a vertex selection mechanism because it processes each vertex at every superstep, without checking its active status or the presence of pending messages in its mailbox. By contrast, the Connected Components and SSSP benchmarks do require vertices to be selected since they may become inactive during the computation. As a result, such algorithms cannot be implemented in *FemtoGraph* without rewriting parts of the framework itself. Furthermore, we have not observed correct results across all the graphs tested. Nonetheless, *FemtoGraph* remains an interesting reference since it is the only other vertex-centric framework specifically designed for in-memory storage and synchronous execution, like *iPregel*.

Ligra Out of the three categories illustrated in Figure 4.10, *Ligra*[16] belongs to the rightmost: non-vertex-centric frameworks, which includes vertex-subset-centric, with synchronous execution. Its approach, described as vertex-subset-centric in [54] as opposed to vertex-centric, consists of dividing the graph processed into subsets, which are run in turn. *Ligra* executes on each subset two functions defined by the user: one to apply to every vertex and one to apply to every edge. Additionally, the user must implement the `compute` function, which in *Ligra* is the function that defines the overall execution flow of the application, from a graph-centric view. For instance, the user is in charge of writing the main loop, as well as its termination condition, within which they must explicitly pass the graph to the vertex and edge functions they defined earlier. Nonetheless, *Ligra* is a graph processing framework that relies on shared-memory parallelism, in-memory storage and synchronous execution, so in that regard, it acts as a non-vertex-centric counterpart of *iPregel*.

4.6.1.2 Computing environment

Experiments are run on Cirrus, an HPE/SGI Apollo 8600 system, in which each compute node is equipped with two 18-core Intel Xeon E5-2695 (Broadwell) series processors. Each compute node contains 256GB of RAM made up of two Non-Uniform Memory Access

[15] Available at https://github.com/DataSys-IIT/FemtoGraph
[16] Available at https://github.com/jshun/ligra

(NUMA) regions each containing 128GB. All nodes run the CentOS 7 Linux operating system.

iPregel is compiled with *gcc* version 6.3.0, using C99 standard (GNU99 extensions when using spinlocks) and is parallelised with OpenMP. *Ligra* supports OpenMP and Cilk Plus parallelisation. To make the comparison with *iPregel* consistent, the OpenMP version was selected. The frameworks *Ligra*, *GraphChi* and *FemtoGraph*, which are developed in C++, are compiled with `g++` version 6.3.0, using C++14 standard. The optimisation level is set to `-O3` for all frameworks.

4.6.1.3 Methodology

The timings reported include only the processing time, that is, graph loading and result writing are not included. The second factor measured during experiments is the *resident set size*, which represents the peak memory usage of an application over its entire runtime. Unlike the performance, which is assessed purely on the runtime, the memory peak usage includes all phases of an application, which are graph loading, processing and result writing. The motivation in using this metric is to assess whether a framework can process a graph given a certain amount of memory, which is conditional upon the success of all phases, not only the processing.

4.6.1.4 Graphs used

Table 4.3 lists the graphs processed in the experiments conducted in this work, where $|V|$ is the graph order, and $|E|$ is the graph size. They are real-world graphs selected from the online collection Stanford Network Analysis Project [77] (SNAP) and cover a range of sizes ranging from millions to billions of edges. These graphs are undirected, and therefore the total number of directed edges is twice the amount presented. The smallest graph, the Database and Logic Programming Bibliography graph[17] (DBLP), is a real-world graph that represents the eponymous computer science bibliography. LiveJournal[18], Orkut[19] and Friendster[20] are network graphs about blogging, social and gaming respectively.

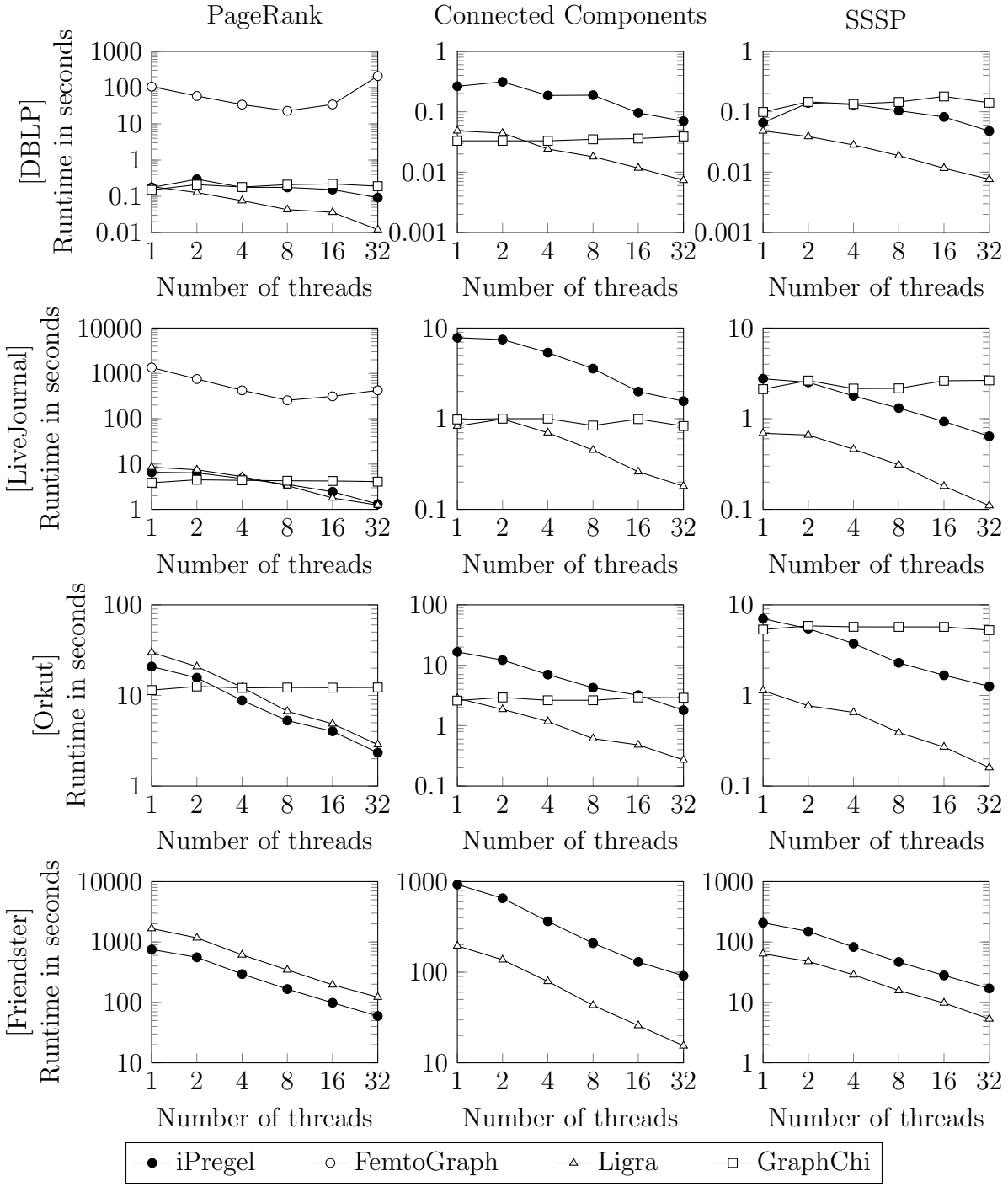

Figure 4.12: Variation of *iPregel*, *Ligra*, *GraphChi* and *FemtoGraph* runtimes (in seconds) against the number of nodes used, for each benchmark application, per graph.

Table 4.3: Graphs used for the experiments in this section.

Name	$\|V\|$	$\|E\|$
DBLP	317,080	1,049,866
Live Journal	4,036,538	34,681,189
Orkut	3,072,441	117,185,083
Friendster	65,608,366	1,806,067,135

4.6.2 Results

4.6.2.1 Performance

Figure 4.12 illustrates the results of the three benchmarks across the four different frameworks, with different graphs. For PageRank, illustrated in the left column of Figure 4.12, it is observed that the *iPregel* version is 70 to 2,300 times faster than its *FemtoGraph* counterpart[21]. *GraphChi* and *Ligra* outperform *FemtoGraph* too, resulting in a maximum speedup of 700 and 17,000 respectively. The best sequential performance, regardless of the graph, is achieved by *GraphChi*[22], and this is due to its asynchronous execution which enables vertices to read values updated by other vertices during this same superstep. However, this asynchronicity delivers no performance gain when the number of threads increases and eventually performance decreases compared to both *iPregel* and *Ligra*. The results reported for PageRank in Figure 4.12 demonstrate that the thread scalability of *iPregel* is similar to that of *Ligra*. They also suggest better graph scalability[23] in *iPregel*. In addition, Figure 4.12 shows that the bigger the graph, the better the thread scalability of *iPregel*. The performance differences observed for PageRank between *Ligra* and *iPregel* can be explained using these three factors. On the smallest graph DBLP, *Ligra* begins with a sequential runtime lower than *iPregel* and also provides better thread scalability. With the Live Journal graph, the number of vertices and edges are multiplied by 10 and 30 respectively. At this point, *iPregel* now begins to outperform *Ligra* at 1 thread, and provides thread scalability better than with DBLP graph. Despite these performance benefits, the strong thread scalability of *Ligra* eventually enables it to outperform *iPregel*

[17]https://snap.stanford.edu/data/com-DBLP.html
[18]https://snap.stanford.edu/data/com-LiveJournal.html
[19]https://snap.stanford.edu/data/com-Orkut.html
[20]https://snap.stanford.edu/data/com-Friendster.html
[21]*FemtoGraph*'s timings for Orkut and Friendster graphs could not be collected due to abnormal termination and out-of-memory failure respectively.
[22]*GraphChi*'s timings for the Friendster graph could not be collected due to the number of file descriptors needed, approximately 21,000, being beyond our allowed limit.
[23]The capacity to provide performance gains when the size of the graph increases.

Table 4.4: Minimum, average and maximum speedup of *Ligra* over *iPregel* when processing the Connected Components of each graph, across all numbers of threads tested.

Graph	Min	Avg	Max
DBLP	5.47	8.07	10.44
Live Journal	7.52	8.17	9.43
Orkut	5.72	6.47	7.77
Friendster	4.60	4.99	5.92

beyond 8 threads. When considering the hundred-million-edge graph Orkut, however, with PageRank, *iPregel* outperforms *Ligra* at 1 thread and manages to remain ahead across all numbers of threads due to the thread scalability. With the billion-edge graph Friendster, *iPregel* now provides thread scalability as good as that of *Ligra*, with the runtime being half of that reported by *Ligra* across all numbers of threads. Overall, for PageRank, the best performance at 32 threads is achieved by *iPregel*.

For the Connected Components benchmark, whose results are shown in the middle column of Figure 4.12, certain patterns already observed for PageRank can be observed. Namely, *GraphChi* delivers no thread scalability, which enables *Ligra* and *iPregel* to become competitive at a higher number of threads. However, the performance achieved by *GraphChi*, due to its asynchronicity, is rarely equalled by *iPregel*, even at 32 threads. Nonetheless, *iPregel* continues to exhibit better graph scaling than *Ligra* as illustrated in Table 4.4. The speed up of *Ligra* over *iPregel* decreases as the graph grows, although always remaining greater than 1. The vertex-centric *iPregel* remains up to 10 times slower than the vertex-subset-centric *Ligra* which leverages atomic combiners. Overall, there are however two major differences between the results observed for Connected Components and PageRank. Firstly, the optimal sequential performance is now achieved by both *GraphChi* and *Ligra*. Secondly, the thread scalability of *iPregel* is as good as *Ligra*'s on all graphs.

The timings collected in SSSP, presented in the right column of Figure 4.12, follow patterns found in the timings gathered for the Connected Components. Indeed, although *Ligra* remains several times faster than *iPregel*, the performance difference diminishes as the size of the graph increases. In between these two extremes stands *GraphChi*, faster than *iPregel* on low numbers of threads but due to its poor scalability it falls behind as the number of threads increases. Furthermore, *iPregel* continues to deliver thread scalability as good as that of *Ligra*.

Table 4.5: Maximum resident set size (in GB) of each framework tested across all graphs processed, for each application executed. (Abbreviations used: ABT = Abnormal Termination, OOM = Out Of Memory, FDO = File Descriptor Overflow)

	Graph	iPregel	FemtoGraph	GraphChi	Ligra
PageRank	DBLP	0.07	3.26	0.07	0.04
	Live Journal	0.48	51.95	1.41	0.51
	Orkut	1.08	ABT	3.91	1.10
	Friendster	20.45	OOM	FDO	21.43
CC	DBLP	0.15	-	1.06	0.03
	Live Journal	0.42	-	2.49	0.48
	Orkut	1.03	-	7.58	1.07
	Friendster	20.94	-	FDO	20.45
SSSP	DBLP	0.14	-	0.10	0.02
	Live Journal	0.47	-	2.49	0.42
	Orkut	1.07	-	7.57	1.04
	Friendster	19.91	-	FDO	18.19

4.6.2.2 Memory footprint

The memory footprints collected from these different benchmark runs are reported in Table 4.5. It can be seen that *FemtoGraph* is up to 100 times less memory efficient than *iPregel*, eventually resulting in an out-of-memory failure for Friendster. The high memory overhead required by *FemtoGraph* is partly due to the lack of message combination. Indeed, each vertex is provided with a mailbox that contains space for 100 messages while *iPregel* mailboxes only store the combined message, as described in Subsubsection 4.3.4.3. When processing Friendster graph's 65 million vertices (see Table 4.3), the *FemtoGraph* mailbox requires 26GB[24] while that of *iPregel* uses 0.26GB. In addition to resulting in message losses when a vertex receives more than 100 messages, the *FemtoGraph* design also wastes memory for vertices that receive fewer than 100 messages. It was not possible to process Orkut with *FemtoGraph* due to an abnormal termination.

According to Table 4.5, *GraphChi* is approximately 40 times more memory efficient than *FemtoGraph*, resulting in a memory footprint that is within the same order of magnitude as *iPregel*. Nonetheless, *GraphChi* is between 3 and 6 times less memory efficient on average. Despite providing an in-memory version of several applications, *GraphChi* remains a framework tailored for out-of-core computation, and it is therefore understandable that memory usage is not as optimised as that of a pure in-memory

[24] 65 million vertices storing 100 4-byte integers each

```
1  void Compute(MessageIterator* msgs) {
2      if(superstep() >= 1) {
3          double sum = 0;
4          for(; !msgs->Done(); msgs->Next()) {
5              sum += msgs->Value();
6          }
7          *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
8      }
9      if(superstep() < 10) {
10         const int64 n = GetOutEdgeIterator().size();
11         SendMessageToAllNeighbors(GetValue() / n);
12     }
13     else {
14         VoteToHalt();
15     }
16 }
```

Figure 4.13: Compute function for PageRank in Pregel.

framework like *iPregel* or *Ligra*.

In Table 4.5 it can be observed that the memory footprint of *Ligra* is similar to that of *iPregel*. In the majority of experiments, the difference is smaller than 60MB. The maximum difference in favour of *Ligra* is for SSSP on Friendster, where its memory footprint is 1.72GB (or 9%) smaller than that of *iPregel*. Conversely, running PageRank on the Friendster graph is where *iPregel* makes the biggest difference in its favour with 20.45GB against 21.43GB for *Ligra*; saving 0.98GB (approximately 5%). Among the two frameworks, the best in terms of memory efficiency depends on the benchmark and graph being processed, *Ligra* proves to be more efficient than *iPregel* 7 times, while the contrary is observed 5 times. As a consequence, *iPregel* manages to provide a vertex-centric interface with a memory footprint as competitive as its non-vertex-centric counterpart.

4.6.2.3 Programmability

In this section, the programmability of the different frameworks is evaluated by comparing against the vertex-centric interface provided by Pregel. Although *Pregel* is available within Google exclusively, its implementations for benchmarks used in this chapter are given in the original chapter [11].

PageRank is the only one implemented by all four frameworks considered in this chapter, so it was selected as the reference benchmark. The PageRank implementation using the original *Pregel* framework is illustrated in Figure 4.13, taken from [11]. Three characteristics can be observed which can be used as evaluation criteria:

1. A vertex-centric interface; representing the fundamental advantage of the *Pregel*

Table 4.6: Evaluation of frameworks considered against the programmability criteria defined from the *Pregel* implementation of PageRank.

Framework	*iPregel*	*FemtoGraph*	*GraphChi*	*Ligra*
Vertex-centric interface	Yes	Yes	Yes	No
Encapsulated attributes	Yes	Yes	No	No
Vertex halting	Yes	Yes	No	No

API with regard to programmability.

2. Encapsulated vertex data, that is, data specific to vertices are stored in vertices themselves, such as the rank for PageRank. This contrasts with another possible approach where vertices would fetch their rank from a global structure shared across all vertices. The latter however would require the user to be aware of the underlying addressing algorithm between a vertex identifier and the corresponding position in the global structure. As a result, encapsulating vertex attributes improves programmability by letting the framework handle the vertex addressing while exposing a less error-prone programming interface to the user.

3. The completion of a vertex is expressed via a halting function. This is the cornerstone of vertex selection and algorithm termination, yet it requires very little work from the user, simply calling the halting function.

Table 4.6 reports these three programmability attributes against the four graph processing frameworks that have been considered here. As can be seen by their implementations given in Figures 4.4 and 4.14, they offer a highly abstracted vertex-centric interface, where vertex-specific information is encapsulated in the vertices and the halting mechanism is invoked by vertices using a simple function call.

GraphChi also provides a vertex-centric interface; however, in its implementation of PageRank in Figure 4.15 vertex ranks are contained in a single array, *pr*. As a result, the user is responsible for handling the vertex addressing, and they must manipulate this global structure from a centralised view and not a vertex-centric one. In addition, *GraphChi* performs the vertex selection via a vertex scheduler, which can be disabled for algorithms such as PageRank, resulting in no halting mechanism available at the vertex-level. Although the algorithm termination is *based on all vertices voting to halt* according to Pregel [11], in the *GraphChi* version of PageRank it is determined by the main function, where a maximum number of iterations is defined. The property of PageRank where all vertices are active at every superstep is not shared for the majority of graph processing

```
1  void compute(queue<message*, fixed_sized<true>>* messages) {
2      if(graph->superstepcount >= 1) {
3          double sum = 0;
4          message* m;
5          while(messages->pop(m)) {
6              sum += m->data;
7          }
8          data->weight = 0.15 / graph->size() + 0.85 * sum;
9      }
10     if(graph->superstepcount < 10) {
11         const long n = outEdges.size();
12         sendMessageToNodes(neighbors, data->weight / n);
13     }
14     else {
15         voteToHalt();
16     }
17 }
```

Figure 4.14: Compute function for PageRank in *FemtoGraph*.

algorithms such as CC or SSSP, therefore requiring a vertex selection mechanism. In *GraphChi*, this is achieved via a vertex scheduler that must be explicitly enabled or disabled by the user, then called in user code when processing each vertex. Indeed, for an algorithm that requires vertex selection such as SSSP, vertices that send a message must then explicitly call the scheduler and schedule the recipient vertex for execution. This approach has the disadvantage of exposing implementation-level details to the user. By contrast, *iPregel* abstracts the vertex selection inside the call to the halting function. Furthermore, the selection bypass optimisation presented in Subsubsection 4.3.4.1 is enabled via a compilation flag, without requiring a modification in the user application source code. That allows the user to rely on a consistent programming interface across all applications, unlike *GraphChi* where, for instance, vertices do not halt in PageRank whereas they do in SSSP, and sending a message must be followed by an explicit schedule of the recipient vertex in SSSP, while it does not in PageRank.

Finally, Figure 4.16 reports the *Ligra* implementation of PageRank. For a fair of comparison, the source code selection that calculates the convergence of PageRank was ignored because other frameworks (including *iPregel*) do not provide this. This is beneficial for *Ligra* from a programmability perspective because it hides details about convergence calculations from the code. Nonetheless, none of the criteria presented in Table 4.6 are observed for *Ligra*, although this is understandable for a framework that is not vertex-centric but vertex-subset-centric. The source code explicitly exposes parallelism to the user in two aspects. Firstly, syntactically, as can be seen in Figure 4.16 with the use of parallel for loops wrapped in curly brackets. Secondly, semantically, as *Ligra* states

```
1  void update(graphchi_vertex<VertexDataType, EdgeDataType>& v,
2              graphchi_context& ctx) {
3      if(ctx.iteration == 0) {
4          pr[v.id()] = 1.0 / ctx.nvertices;
5      }
6      else if(ctx.iteration > 0) {
7          float sum = 0.0;
8          for(int i = 0; i < v.num_inedges(); i++) {
9              sum += pr[v.inedge(i)->vertexid];
10         }
11         pr[v.id()] = 0.15 / ctx.nvertices + 0.85 * sum;
12         if(v.outc > 0) {
13             pr[v.id()] /= v.outc;
14         }
15     }
16     if(ctx.iteration < 10) {
17         v.set_data(v.outc > 0 ? pr[v.id()] * v.outc : pr[v.id()]);
18     }
19 }
```

Figure 4.15: Compute function for PageRank in *GraphChi*.

in [53], the function provided to edgeMap "*can run in parallel, so the user must ensure parallel correctness*". Therefore, the user is in part responsible for the thread safety of *Ligra*. Moreover, the iterative structure of computation as well as dynamic memory allocations and deallocations are undertaken directly by the user, as illustrated in Figure 4.16. This is a disadvantage that the designers of *Ligra* have accepted to obtain increased performance, but such concerns are too low-level for the user. Furthermore, the `compute` function only outlines the general computation flow. The edge map and vertex map functions must be defined by the user as well, which are given in Appendix B for PageRank. As a consequence, the resulting implementation of PageRank on *Ligra* far exceeds that of *iPregel*, by a factor of three: 76 lines of code compared to 24 lines. Also, as explained in Subsubsection 4.3.4.2, *Ligra* provides atomic combination as an additional optimisation, which is enabled by the user providing a second version of the update function. This requires the user to be aware of the atomicity potential of their combination operation, as well as being able to implement it atomically using *Ligra*'s functions.

4.7 Conclusions and future work

This chapter has presented the vertex-centric framework *iPregel*, which leverages in-memory storage and shared-memory parallelism. The optimisation techniques developed address multiple performance challenges in vertex-centric programs, from vertex selection to message combination and vertex indexing. The underlying modular design of *iPregel*

```
1  template <class vertex>
2  void Compute(graph<vertex>& GA, commandLine P) {
3      long maxIters = 10;
4      long iter = 0;
5      const intE n = GA.n;
6      double one_over_n = 1 / (double)n;
7      double* p_curr = newA(double, n);
8      {parallel_for(long i = 0; i < n; i++) {
9          p_curr[i] = one_over_n;
10     }}
11     double* p_next = newA(double, n);
12     {parallel_for(long i = 0; i < n; i++) {
13         p_next[i] = 0;
14     }}
15     bool* frontier = newA(bool, n);
16     {parallel_for(long i = 0; i < n; i++) {
17         frontier[i] = 1;
18     }}
19     vertexSubset F(n, n, frontier);
20     while(iter++ < maxIters) {
21         edgeMap(GA, F, PR_F<vertex>(p_curr, p_next, GA.V), 0, no_output);
22         vertexMap(F, PR_Vertex_F(p_curr, p_next, 0.85, n));
23         vertexMap(F, PR_Vertex_Reset(p_curr));
24         swap(p_curr, p_next);
25     }
26     F.del();
27     free(p_curr);
28     free(p_next);
29 }
```

Figure 4.16: Compute function for PageRank in *Ligra*.

allows the user to pass additional information about the graph and the application via compilation flags, which are leveraged by *iPregel* to apply optimisations techniques specifically designed for the sets of assumptions expressed through flags. This allows *iPregel* to exploit assumption-specific optimisations without sacrificing software flexibility.

Experiments conducted in the first part of this chapter demonstrated that *iPregel* outperforms a state-of-the-art in-memory distributed-memory vertex-centric framework. On a single node, timings collected show that *iPregel* is always faster than *Pregel+*, by a median factor of 6.5. At worst, *iPregel* is 3.5 times faster, and up to more than 600 times faster in the best case. Additionally, performance achieved by *iPregel* remains competitive even when considering multi-nodes. Indeed, the timings collected reveal that at least 11 nodes are needed by *Pregel+* to equal or outperform *iPregel*.

iPregel also demonstrates that the performance gains obtained do not come at the expense of memory consumption. In fact, the memory footprint of *iPregel* is an order of magnitude smaller than its in-memory distributed-memory counterparts, requiring needing 11GB whilst the latter requires up to a quarter of a terabyte. Further experiments demonstrated that *iPregel*, albeit single-node, can process multi-billion edge graphs under 16GB of memory, or the USA road network using less memory than that of a smartphone. The results collected demonstrate that there is room for optimisation in the memory footprint of vertex-centric frameworks. By outperforming the *Pregel+* distributed-memory counterpart in all experiments, as well as delivering a memory efficiency an order of magnitude better, *iPregel* demonstrates that in-memory shared-memory vertex-centric frameworks are a viable solution for graph processing.

Then, the trade-off observed between vertex-centric programmability, performance and memory was analysed in the second part of this chapter. Experiments illustrated this compromise, where the *FemtoGraph* framework differs from *Ligra* and *GraphChi* by successfully preserving vertex-centric programmability, at the expense of performance, up to 17,000 times and 700 times worse than that of *Ligra* and *GraphChi* respectively, as well as a memory footprint up to orders of magnitude larger.

Experiments also demonstrated that the framework developed in this research, *iPregel*, provides the best of both worlds. The optimisation techniques that have been developed in this research enable *iPregel* to bridge the weaknesses of *FemtoGraph* in terms of memory efficiency and performance. Results show that the memory efficiency of *iPregel* equals that of *Ligra*, which is the most memory-efficient framework tested. *iPregel* was also up to 100 times more memory efficient than *FemtoGraph*, and up to 7 times more memory efficient than *GraphChi*. This additional memory efficiency enables *iPregel* to process graphs that *FemtoGraph* cannot because its memory footprint exceeds the

available memory. Regarding performance, the maximum speedup of *GraphChi* or *Ligra* over *iPregel* is at most 10 times, which is up to 1,700 times less than the speedup they can achieve over *FemtoGraph*. The performance observed on *iPregel* is between 70 times and 2,300 times greater than that of *FemtoGraph*. For PageRank, *iPregel* delivers a competitive performance, outperforming all other frameworks considered in half of the graphs tested. Most importantly, the performance and memory efficiency gains that have been observed without sacrificing programmability, such as removing the message-passing interface, exposing parallelism or atomic operations.

Therefore, the two-phase investigation reported in this chapter demonstrates that the *iPregel* framework not only proves to be a viable solution for graph processing, but also overcomes the fundamental compromise in vertex-centric frameworks by demonstrating an approach where programmability no longer impacts memory efficiency, and can deliver competitive performance. Consequently *iPregel* is the first shared-memory vertex-centric framework able to scale to a multi-billion edge graph without sacrificing the programmability benefits of vertex-centric.

Further improvements of *iPregel* could include the design and implementation of atomic combiners that do not impact programmability of the interface exposed to the user. Furthermore, the multi-threaded performance observed would benefit from additional investigations into load-balancing strategies and work-stealing techniques. Given that certain algorithms require a concatenation as part of the combination operation, further exploration in this area could improve the flexibility of this framework.

Chapter 5

Tackling the irregularity inherent in vertex-centric

5.1 Introduction

Following the evaluation of our *iPregel* framework against other shared-memory vertex-centric frameworks in Chapter 4, it was observed that *iPregel* delivered competitive performance and memory usage without sacrificing the high productivity programming abstractions of this model. Subsequently, the focus was on improving *iPregel* by designing additional techniques, and as highlighted in Section 4.7, *iPregel* would benefit from optimisations around load-balancing and internal parallelism.

A fundamental challenge is that, whilst the vertex-centric programming model provides significant programmability advantages, and exposes a large degree of latent parallelism, it suffers from numerous irregularities that impact performance. The vertex-centric model exhibits properties that can be more widely described by the class of irregular applications. These are:

- **Fine-grain synchronisation**: the communications in vertex-centric programs occur at a vertex's mailbox level, hence any data-race protection must be implemented on a per-vertex basis.

- **Unpredictable memory access patterns**: broadcasting a message from a vertex to its neighbours results in a number of messages that is specific to each vertex. This is further aggravated by the highly skewed distribution of degrees, where the number of neighbours may significantly vary from a vertex to the next. Furthermore, recipient vertices are unlikely to reside next to each other in memory, making these memory accesses unpredictable both in terms of quantity and location.

- **Load imbalance**: the number of active vertices may vary drastically from one superstep to the next. Effectively this means that the number of neighbours that a vertex needs to communicate with will change from one superstep to the next.

Vertex-centric programs are inherently difficult to optimise because they tend to follow the form of a short user-provided source code, resulting in the underlying framework being provided with little information to leverage for performance optimisation. In the meantime, since programmability is the essence of vertex-centric, any performance optimisations that have been developed must not degrade the programmability properties of the framework.

However, numerous attempts to optimise the vertex-centric model have negatively impacted upon the ability for users to easily to easily program using this paradigm, highlighting the tension between performance optimisation and programmability [2]. By contrast, the approach developed in this research integrates the preservation of vertex-centric programmability as its core. To that end, all the optimisations discussed in this chapter are encapsulated within the *iPregel* framework, requiring no user source modification to take advantage of them.

The main contributions presented in this chapter can be summarised as follows:

- **A hybrid combiner** designed to couple lock-free and lock-based interactions in order to efficiently handle fine-grain synchronisations.
- **The externalisation of vertex attributes** to better cope with unpredictable memory accesses by improving the cache efficiency through the grouping of vertex attributes based on their access frequency.
- **An edge-centric workload representation** that ameliorates load imbalance whilst preserving both the vertex-centric paradigm and *Pregel* user interface.

The rest of this chapter is organised as follows: Section 5.2 depicts the context in which this research takes place. Sections 5.3, 5.4 and 5.5 introduce the optimisations considered in this chapter. Section 5.6 describes the environment in which experiments were conducted while Section 5.7 presents and analyses the results obtained. This chapter then concludes in Section 5.8; summarising the findings of this work as well as discussing potential future work directions.

5.2 Related work

There have been some attempts by the vertex-centric community to address the implications of irregular workloads present in graph processing algorithms. As shown in

Chapter 3, the literature shows that bypassing the costly selection of active vertices has been addressed, whether it has to be done manually by the user such as in *GraphChi* [24], or automatically due to the analysis of algorithmic patterns as introduced in early *iPregel* work [1]. However, the dispatching of active vertices to workers remains an unsolved challenge. Indeed, accurately evaluating the workload contained in these active vertices is key to an efficient workload dispatch. The common approach in vertex-centric frameworks consists of distributing an equal number of active vertices to each worker. However, this approach is sub-optimal due to the power-law degree distribution that typically underpins the graphs processed. This observation led to the development of PowerGraph [59], where the authors adopted a more edge-centric approach, which resulted in an entirely new interface based upon the scatter-apply-gather design instead of the typical *Pregel* single user-defined function. As such, many of the abstraction and programmability benefits of the vertex-centric model were lost.

The edge-centric approach was taken one step further by X-Stream [78], whose implementation and interface are designed entirely from an edge-centric perspective, exposing an edge-centric Gather-Apply-Scatter interface to the user. The underlying motivation for this design was to address the randomness of memory accesses by reading the graph's edges sequentially. Nonetheless, edge-centric frameworks, which by definition are no longer vertex-centric and hence cannot provide the same benefits, demonstrate that addressing the irregularity challenges associated with vertex-centric without sacrificing certain aspects of the actual programming model is not a trivial task.

This difficulty in addressing vertex-centric challenges is also illustrated in the implementations for the fine-grain synchronisations required in vertex-centric programs during message exchange. Typically, communications are redesigned as pull-based [24] so that they are lock-free, or a semaphore is needed for each vertex mailbox where appropriate. An alternative approach for the latter is to implement the message combination as a compare-and-swap, as used by *Ligra* [53]. However, despite providing performance benefits, this design reduces the level of abstraction and requires the end programmer to interact with the framework at a lower level, potentially rewriting certain parts of their code, in addition to raising additional restrictions that will be discussed in details later in this chapter.

As described in this section, optimisations for dealing with irregular workloads in vertex-centric frameworks typically result in the sacrifice of features of programmability of the vertex-centric model. By contrast, in Chapter 4 it was demonstrated that vertex-centric optimisations could be designed without this disadvantage, and in this chapter a similar approach is followed to address the irregularities inherent in the model.

5.3 Fine-grain synchronisation

Vertex-centric programs require each vertex's mailbox to be protected against potential data-races. The write-interactions with that mailbox are achieved during combination, which means that combiners are a key area for optimisations since any improvement to their design will directly impact the performance observed. To implement combiners, two designs are available:

- **Lock based approach**: a classic design where vertices acquire the lock held on the recipient vertex, check if that recipient vertex already received a message, and if so the sender vertex combines the existing message with the new one. Otherwise, the new message is stored, before releasing the lock.

- **Compare-and-Swap**: a lock-free design where vertices retrieve the existing message from their recipient's vertex mailbox, combine it and then push it back using a compare-and-swap operation. This operation checks if the value of the message is identical to that read earlier. If so, the message is updated with the new value and returns `true`, otherwise the value has changed, which implies that another vertex updated this recipient's mailbox first, in which case the entire operation is repeated until it eventually succeeds.

The second approach has the advantage of avoiding locks, however, it systematically combines a new message with the existing one, therefore relying on the assumption that mailboxes begin with a default message value that is neutral to the combination operation. For instance, in a combination operation that sums messages, vertices' mailboxes would begin each superstep with a message value of 0. This requirement for a neutral value implies that either the user must be constrained to a set of predefined combination operations whose neutral values are hardcoded, or else the user must somehow declare the neutral value for any combination operation that they write. In the *Ligra* version of PageRank, for instance, the combination operation is a sum (thus having the neutral value 0). For the user, this results in manually resetting each vertex mailbox to 0 at the end of every superstep.

The second drawback of a pure compare-and-swap design comes from the lack of a notion of empty mailboxes. Mailboxes always have a message, either representing the result of a combination or the default neutral value. Therefore, a vertex knows that it has received a message if its mailbox message value is different from the neutral value. However, in a scenario where the combination operation would result in the neutral value itself, the vertex would assume it has not received a message, whilst in fact, it has. In vertex-centric programs, this can lead to incorrect outputs since receiving messages is

```
1  void apply_cas(IP_VERTEX_TYPE* dst, IP_MESSAGE_TYPE msg){
2      IP_MESSAGE_TYPE old_msg = dst->msg_next;
3      IP_MESSAGE_TYPE new_msg = old_msg;
4      ip_combine(&new_msg, msg);
5      while(new_msg != old_msg &&
6            !atomic_compare_exchange_strong(&dst->msg_next, &old_msg,
7                                            new_msg)) {
8          old_msg = dst->msg_next;
9          new_msg = old_msg;
10         ip_combine(&new_msg, msg);
11     }
12 }
13
14 void ip_send_message(IP_VERTEX_ID_TYPE dst_id, IP_MESSAGE_TYPE msg) {
15     IP_VERTEX_TYPE* dst=ip_get_vertex_by_id(dst_id);
16     if(dst->has_msg_next) {
17         apply_cas(dst, msg);
18     }
19     else {
20         ip_lock_acquire(&dst->lock);
21         if(dst->has_msg_next) {
22             ip_lock_release(&dst->lock);
23             apply_cas(dst, msg);
24         }
25         else {
26             dst->message_next = msg;
27             dst->has_message_next = true;
28             ip_lock_release(&dst->lock);
29         }
30     }
31 }
```

Figure 5.1: Implementation in *iPregel* of the hybrid combiner

what reactivates inactive vertices. To obtain an approach that combines the best of both techniques, that is exploiting compare-and-swap while keeping the notion of an empty mailbox, as well as enabling the user to define any arbitrary combination operation, a hybrid combiner was designed that leveraged lock-based and lock-free interactions with a recipient's mailbox. The implementation of this combiner is provided in Figure 5.1, where vertex attributes have been shortened for brevity. In this example, `ip_combine` is the user-defined combination function, `has_msg_next` is the flag indicating whether the vertex has already received a message during this superstep and `msg_next` is the message itself (whose value is meaningful only if the flag is `true`).

As shown in Figure 5.1, the hybrid combiner couples lock-free and lock-based interactions. Correctness comes from the guarantee that if the `has_msg_next` flag of a recipient vertex is `true`, that the value held in that vertex mailbox has been set. Indeed, as soon as the `has_msg_next` flag is `true`, potential compare-and-swap combinations may ex-

ecute concurrently on that vertex from other workers. Therefore, the value that they will fetch from any recipient vertex's mailbox must have been set by that time.

To provide this guarantee, when a worker pushes its first message to a recipient's mailbox, the message is stored (line 26) before setting the flag to `true` (line 27). Furthermore, to avoid a potential out-of-order execution, a full memory barrier is required in between these operations. This barrier is provided by declaring the `has_msg_next` flag as atomic, using C11 atomics, which implicitly enforce a sequentially consistent memory model. Without such a barrier, an out-of-order execution could result in a recipient's vertex entering a state where its `has_msg_next` flag is set to `true` while not having the `msg_next` message yet set. Another worker attempting to push a message to that vertex mailbox would therefore check the flag, observe that it is `true` and apply a compare-and-swap with the message that is still unset. Finally, having the `has_msg_next` flag as atomic implies that the read at line 16 and write at line 27 are also atomic, guaranteeing that a read cannot happen on a flag partially written to memory.

The rest of the hybrid combiner is straightforward; workers check if the recipient's vertex already holds a message and if so they use a compare-and-swap combination, otherwise they acquire the lock. When the lock is acquired by a worker it again checks against the recipient vertex's flag in case, while it was waiting to acquire the lock, another worker that was holding that lock pushed the first message into that recipient's mailbox. In this event, the recipient's vertex now has a mailbox containing a message and the worker can release the lock and use the compare-and-swap combination. Otherwise, it continues to hold the lock and performs the first message push to that recipient vertex's mailbox, ensuring that store operations are issued in the order explained earlier.

5.4 Unpredictable memory access patterns

In vertex-centric programs, the irregularity in memory accesses is two-fold. Firstly, the power-law distribution that commonly underpins graph processing, results in vertices having widely different numbers of neighbours. Secondly, the inherent irregular structure of graphs means that each vertex may be connected to any other arbitrary vertex. In other words, the data for neighbouring vertices may reside at any location in memory, and is therefore highly unlikely to be contiguous with each other. As a consequence, when a vertex broadcasts a message to its neighbours, there are an arbitrary number of memory accesses which must be undertaken, and each of these accessing an arbitrary location in memory. This section presents optimisations designed in response to analysing these unpredictable memory access patterns.

```
1  void ip_fetch_broadcast_messages(IP_VERTEX_TYPE* v) {
2      IP_NEIGHBOUR_COUNT_TYPE i = 0;
3      while(i<v->in_neighbour_count &&
4          !ip_get_vertex_by_id(v->in_neighbours[i])->has_broadcast_message) {
5          i++;
6      }
7      if(i >= v->in_neighbour_count) {
8          v->has_message = false;
9      }
10     else {
11         v->has_message = true;
12         v->message =
13             ip_get_vertex_by_id(v->in_neighbours[i])->broadcast_message;
14         i++;
15         IP_VERTEX_TYPE* temp_vertex = NULL;
16         while(i < v->in_neighbour_count) {
17             temp_vertex = ip_get_vertex_by_id(v->in_neighbours[i]);
18             if(temp_vertex->has_broadcast_message) {
19                 ip_combine(&v->message, temp_vertex->broadcast_message);
20             }
21             i++;
22         }
23     }
24 }
```

Figure 5.2: Implementation of the message fetching phase in the single-broadcast version of *iPregel*

5.4.1 Vertex structure externalisation

Whilst they are unpredictable, these memory access patterns do expose one regularity, which is that attributes are accessed from a neighbour's vertex structure. For instance, in the single-broadcast version of *iPregel*, where communication recipients pull messages from their sender's outbox, profiling reveals that up to 99% of the runtime is occupied in iterating through all incoming neighbours and retrieving messages from those who are flagged as holding a message to be broadcast. The function that is responsible for message fetching is illustrated in Figure 5.2, and it can be seen that the loop at line 16 iterates through incoming neighbours, checking whether each neighbour is flagged as having a message to broadcast and if so, fetches the corresponding message from their mailbox.

However, as per lines 12-13 show, the only vertex attributes that are accessed in this loop are `has_broadcast_message` and (potentially) `broadcast_message`. Because these attributes are part of the vertex's structure, they are therefore grouped with other vertex attributes such as the vertex's identifier or user-defined value. As a consequence, the cache line(s) in which these attributes are stored are also filled with attributes that will never be used in this loop.

This cache pollution can be minimised by reorganising the vertex structure, externalising the frequently accessed attributes into their own structure. Following this approach, one array contains structures comprising the `has_broadcast_message` and `broadcast_message` attributes, whilst the other array contains structures holding the rest of the vertex attributes. Therefore, when entering the loop at line 16, such a design means that cache lines are only loaded with useful attributes.

5.4.2 Software prefetching

The aim of software prefetching is to enable the programmer to direct the CPU to prefetch data into its cache when the patterns of data access might not be obvious. An example of this is that of indirect memory accesses, where the value from a variable or returned by a function is subsequently used as the array index. This can be especially useful for applications whose memory accesses are irregular, as such situations are notoriously difficult for the hardware prefetcher to correctly predict.

Instead, based upon higher level application knowledge the programmer can use the `_mm_prefetch` function call found in `xmmintrin.h` to fetch data ahead of time. Whilst some benefit to this approach has been demonstrated [79], there is a sensitivity between fetching at the right time, prefetching too early (and the data is then flushed from cache before it is used) or too late (where data access occurs before the prefetch completes).

Furthermore, vertex-centric programs are inherently bandwidth-intensive and as such, any optimisation putting pressure on memory bandwidth is thus unlikely to yield any benefit. This is what has been observed in our experiments, and therefore software prefetching optimisations were not included in the experiments presented in Section 5.7.

5.5 Irregular workloads

A common irregularity that parallel programs face is that of load imbalance, where workers have different amounts of work allocated to them. Vertex-centric programs, where vertices can become inactive during execution and contain different numbers of edges, are prone to load imbalance due to the number of active vertices changing dynamically during the run.

5.5.1 Workload evaluation proxy

Finding the right proxy to evaluate a workload is crucial because it lays down the foundations upon which to build more advanced strategies such as load balancing. Logically,

implementations of the vertex-centric programming model represent their workload in terms of vertices. Although this is accurate with regular data structures, the graphs processed by vertex-centric programs typically follow a power-law distribution, resulting in widely different numbers of neighbours per vertex. Additionally, the runtime of typical vertex-centric programs is dominated by communications and not computation. While the latter is related to the number of vertices, the former depends on the number of edges. Based upon this observation, the hypothesis was that the workload of a thread, which results in the number of combination operations performed and memory writes, or reads, is better expressed as being correlated to the number of outgoing, or incoming, neighbours.

5.5.2 Work distribution

When parallelising a `for` loop in OpenMP by using the `for` construct, one can apply the `schedule` clause, which is provided with a kind that directs how chunks of loop iterations will be distributed to threads, as well as an optional parameter specifying how many iterations comprise a single chunk.

One of the scheduling kinds provided by OpenMP is `dynamic`, and this specifies that chunks of iterations will be distributed on a first-come-first-served basis. This allows threads that have been assigned smaller chunks to be assigned more chunks, thus improving load balancing.

Therefore, the `dynamic` scheduling kind has been included in the set of optimisation techniques evaluated in this chapter. However, to be compatible with this technique, the code must be within a `for` loop whose iteration set distribution can be freely managed by OpenMP. This suits all versions of *iPregel*, except those relying on the edge-centric workload, described in Subsection 5.5.1, since they negate the use of OpenMP dynamic scheduling given that the workload is represented as edges and not vertices. The assigned chunks, therefore, represent workloads on a per-vertex basis.

5.6 Experimental environment

This section describes the conditions and configurations in which the experiments presented in this chapter were conducted.

Table 5.1: Order and size of graphs selected for experiments

Name	Graph order	Graph size
DBLP	317,080	1,049,866
Live Journal	4,036,538	34,681,189
Orkut	3,072,441	117,185,083
Friendster	65,608,366	1,806,067,135

5.6.1 Computing environment

Experiments are run on a standard compute node of Cirrus, whose technical specifications are provided in Subsection 4.6.1.2. What is referred to as the number of threads in the rest of this chapter implies a 1:1 mapping between the number of threads and the number of cores.

The compilation is achieved by using the *gcc* compiler version 8.2.0 with OpenMP version 4.5. Compilation flags passed enable the support for C11 standard (`-std=c11`) and level 3 optimisations (`-O3`). All results reported averaging over 5 number of runs.

5.6.2 Graph configurations

Table 5.1 lists the graphs processed in the experiments presented in this chapter. All four are real-world graphs publicly available in the Stanford Network Analysis Project [77] online collection. The smallest graph, the Database and Logic Programming Bibliography graph[1] (DBLP), represents the eponymous computer science bibliography while LiveJournal[2], Orkut[3] and Friendster[4] are network graphs about blogging, social and gaming respectively. These graphs cover all orders of magnitude from a million to a billion edges and are undirected, meaning that the total number of directed edges is twice the amount presented.

5.6.3 Benchmarks

The experiments presented in this chapter are conducted on the three benchmarks introduced in Subsection 2.3.2, which are commonly used by the vertex-centric community, namely PageRank, Connected Components and Single-Source Shortest Paths.

[1]https://snap.stanford.edu/data/com-DBLP.html
[2]https://snap.stanford.edu/data/com-LiveJournal.html
[3]https://snap.stanford.edu/data/com-Orkut.html
[4]https://snap.stanford.edu/data/com-Friendster.html

As discussed in Section 5.1, optimisations of Sections 5.3, 5.4 and 5.5 are designed in a manner that requires no modifications in user code. Therefore, the *iPregel* benchmark implementations provided in Chapter 4 remain unchanged in the experiments conducted in this chapter.

5.7 Results

Results presented in this section are obtained from experiments designed to evaluate two aspects of the parallel performance of techniques developed:

- **graph scalability**: where the number of threads is fixed in order to isolate the impact of each optimisation against the graph size.

- **thread scalability**: where the graph size is fixed in order to isolate the impact of optimisations against the number of threads used.

5.7.1 Graph scalability

5.7.1.1 Individual optimisations

The experiments presented in this subsubsection consist, for each benchmark, in applying every optimisation individually and calculating the speedup obtained compared to the baseline version, using 32 threads.

The only optimisation not applicable to all benchmarks is the hybrid combiner presented in Section 5.3 because it focuses on improving the usage of locks during combination. Both PageRank and CC rely on lock-free algorithms, making them unsuitable for that optimisation. Nonetheless, the results presented in Table 5.2 show that the hybrid combiner improves the performance of SSSP on all graphs. It also proves to be the optimisation yielding both the biggest speedup overall, up to 4.07 on Friendster, and on average, with a geometrical mean of 1.81. Furthermore, as the size of the graph increases, so does the speedup. The reason for this is that the number of combinations depends upon the number of edges, and because of this the benefit of improving the combination, therefore grows with the number of combinations generated. Overall, the constant performance gain observed from the hybrid combiner demonstrates that when fine-grain synchronisations cannot be achieved using exclusively a lock-free design, partially leveraging lock-free interactions and carefully designing them to work with lock-based interactions can be a beneficial alternative.

Table 5.2: Speedups obtained from each optimisation applied independently, compared to the respective baseline, on each benchmark, using 32 threads, on all graphs ordered by ascending number of edges.

	DBLP	LiveJournal	Orkut	Friendster
PR				
Baseline	1.00	1.00	1.00	1.00
Hybrid combiner	n/a	n/a	n/a	n/a
Externalised structure	1.31	1.27	1.51	1.13
Edge-centric workload	1.01	2.31	1.67	1.36
Dynamic scheduling	1.23	2.31	1.99	1.44
CC				
Baseline	1.00	1.00	1.00	1.00
Hybrid combiner	n/a	n/a	n/a	n/a
Externalised structure	1.58	1.66	1.47	1.65
Edge-centric workload	0.56	1.12	1.27	1.41
Dynamic scheduling	1.23	1.67	1.69	1.20
SSSP				
Baseline	1.00	1.00	1.00	1.00
Hybrid combiner	1.01	1.12	2.35	4.07
Externalised structure	1.08	1.01	1.07	1.10
Edge-centric workload	0.91	0.87	1.28	1.29
Dynamic scheduling	1.11	1.33	1.55	1.69

Similarly, the externalised structure optimisation is beneficial for all graph-benchmarks tested, with a speedup of 1.30 on average. The results in Table 5.2 show that externalising vertex attributes generates the best speedups for CC and the worst for SSSP. The explanation is twofold: firstly, PageRank and CC benefit more because they rely on *iPregel* versions that use pull-based communications, which are lock-free by design. As a consequence, the memory accesses performed during the communications are not interleaved with lock acquisition or release, which reduces further the number of vertex attributes that are frequently accessed. Secondly, PageRank and CC rely on different algorithms, the one underpinning PageRank has one loop that can leverage structure externalisation while the one for CC has two. The overall benefit obtained for CC is therefore greater since it can leverage this optimisation in two parts of the code. Overall, structure externalisation, therefore, demonstrates that heavily irregular memory access patterns may exhibit certain regular aspects which can be leveraged to deliver performance improvements.

The timings reported in Table 5.2 also indicate that shifting to the edge-centric workload proves to be beneficial in 75% of the experiments, resulting in a speedup of 1.19 on average. The extremes are observed for PageRank on Live Journal with a speedup of 2.31 and Connected Components on DBLP with only 0.56. In fact, the edge-centric workload approach performs better on PageRank than on any of the two other benchmarks. The reason for this is that CC and SSSP rely on an *iPregel* implementation leveraging the `selection bypass` optimisation introduced in [1], which helps manage the variable number of active vertices but requires the edge-centric workload distribution to be recalculated at every superstep, therefore increasing the total overhead.

The OpenMP `dynamic` scheduling is the fourth optimisation explored in these experiments. It contrasts with the edge-centric optimisation both because it still represents the workload in terms of vertices and because it no longer statically allocates them to threads but uses a first-come-first-served basis instead. The results reported in Table 5.2 are obtained with an empirically determined chunk size of 256. Unlike the edge-centric optimisation, `dynamic` scheduling improves performance in all experiments, resulting in speedups between 1.11 and 2.31. With an average speedup of 1.50, the first-come-first-served dispatch pattern proves to be a valuable part of an efficient load-balancing strategy.

5.7.1.2 Aggregated optimisations

The optimisations presented in Sections 5.3, 5.4 and 5.5 and evaluated individually in Subsubsection 5.7.1.1 are not mutually exclusive, except the edge-centric workload and

Table 5.3: Final speedups observed compared to the respective baseline, after having applied all beneficial optimisations on each benchmark, using 32 threads, across all graphs

	DBLP	LiveJournal	Orkut	Friendster
PR	1.61	3.14	3.07	1.63
CC	2.05	2.96	2.41	2.12
SSSP	1.09	1.75	3.18	5.73

the dynamic scheduling, and therefore can be combined[5]. As such, an *all-optimisations* version has been developed for each benchmark by combining all optimisations applicable to that benchmark. For PageRank and Connected Components since the hybrid combiner is not applicable, the optimised version consists of the externalised structure with dynamic scheduling. For SSSP however, the optimised version also includes the hybrid combiner. The speedups obtained by the all-optimisations versions against the baseline, for each benchmark, are reported in Table 5.3.

For PageRank and the Connected Components, the speedup patterns exhibited are identical, where the smallest and biggest graphs, DBLP and Friendster, benefit the least but exhibit similar speedups. Live Journal and Orkut provide better speedups, with Live Journal benefiting the most. This correlates to the patterns observed in Table 5.2. In the case of PageRank, the speedups obtained range from 1.61 up to 3.14, meaning that at least 38% of the runtime is saved when using the optimised version. For Connected Components, the speedup range is smaller, reaching only a maximum of 2.96 but never going below 2.05. In other words, at worst, the optimised version halves the runtime of the baseline version.

Table 5.3 demonstrates that, in the case of SSSP, the speedup pattern is different, where the larger the graph, the higher the speedup. This is because of the hybrid combiner, which provides such a speedup pattern, in addition to the dynamic scheduling also exhibiting that pattern for SSSP in Table 5.2. Starting at 1.09 on the smallest graph, the speedup obtained increases until reaching 5.73 on the largest graph, therefore saving up to 83% of the runtime.

Overall, when fixing the number of threads at 32, the optimised versions prove to be beneficial for all benchmarks on all graphs tested. On average, the optimised versions reduce the runtime by almost two-thirds (59%), with extreme cases observed at 8% and 83%.

[5]when the edge-centric workload and dynamic scheduling conflict, the latter is preserved due to the better performance showcased in Table 5.2.

5.7.1.3 Predictability

The expected aggregated speedup of the selected optimisation techniques for each benchmark/graph pair, as presented in Table 5.2, has been calculated and is available in Appendix G.

After calculating the aggregated speedups for each benchmark/graph pair, we conducted further analysis to assess the predictability of the measured aggregated speedup based on the combined individual speedups. To do so, we calculated the ratio between the measured aggregated speedup and the estimated aggregated speedup, which was inverted if it was strictly greater than one. The ratio obtained serves as an indicator of the predictability of the measured aggregated speedup compared to the estimated one. The results of this analysis are presented in Table 5.4.

	DBLP	LiveJournal	Orkut	Friendster
PR	100%	94%	98%	100%
CC	94%	93%	97%	93%
SSSP	90%	85%	82%	76%

Table 5.4: Ratio obtained between the speedup measured and the speedup expected, across all three benchmarks and all four graphs.

The combination of structure externalisation and dynamic scheduling policy in PageRank and Connected Components yields a predictability rate above 93%, with two instances of PageRank achieving 100%. This indicates that the measured aggregated speedup closely matches the one obtained by aggregating individual speedups. However, in the case of SSSP, the predictability rate starts at 90% for DBLP and steadily decreases to 76% for Friendster, implying that the expected speedup is higher than the one measured. This disparity may be attributed to the presence of the hybrid combiner, which sets SSSP apart from the other benchmarks. A potential explanation could be that the specific implementation of the hybrid combiner used in the study is not well-suited to work effectively with the other techniques as these optimisation techniques were designed separately. Further investigation and experimentation would be needed to determine the exact cause of the lack of synergy observed.

5.7.2 Thread scalability

This section contains experiments running the same optimised versions presented in Table 5.3 but varying the number of threads and fixing the graph size. The biggest graph, Friendster, was selected to minimise the timing volatility that would be due to small runtimes. The results from this experiment are reported in Figure 5.3.

Across all benchmarks, the optimised versions exhibit two characteristics. Firstly, their sequential performance is better than that of the baseline, up to 4.36 times faster in the case of SSSP, due to the benefits of structure externalisation and hybrid combination. Secondly, they demonstrate improved thread scalability, which increases the parallel efficiency observed at 32 threads by 17% on average due to structure externalisation and dynamic load balancing. In the case of PageRank for instance, switching from the baseline to the optimised version improves the parallel efficiency from 43% to 64%. The worst case observed is SSSP, with a parallel efficiency at 32 threads rising by 13%, from 43% to 56%.

Using the SSSP benchmark as an example, the performance gains that have been obtained mean that at only 4 threads the optimised version matches the baseline at 32 threads. Similarly, for PageRank, the 32-thread performance of the baseline is matched by the optimised version using running over only 16 threads.

The experimentation results presented in this section demonstrate that the optimisations explored yield a performance benefit in 37 out of the 40 graph-benchmarks pairs tested. The combined optimisations proved to be beneficial in all the cases, whether they are assessed from a graph scalability or thread scalability perspective. For the latter, as shown in Figure 5.4, at 32 threads, the speedups obtained across all three benchmarks have a geometric means of 2.9 times. For SSSP, which is the benchmark with the highest load-imbalance, therefore benefiting the most from the optimisations developed in this work, the speedup reaches 5.7 times.

5.8 Conclusions and future work

This chapter has explored techniques to address the irregular challenges inherent to vertex-centric programs. The first, presented in Section 5.3, was fine-grain synchronisations that underpin message combinations. To that end, a hybrid combiner was developed that couples compare-and-swap and lock-based operations together. Whilst being transparent to the user, results presented in Table 5.2 demonstrate that the hybrid combiner can reduce the runtime by up to 75% and that the performance gain obtained increases with the graph size.

Unpredictable memory access patterns were the second irregularity investigated in this chapter. These are unpredictable, both in terms of quantity and location, and it was found that although one cannot know which vertex's structure will be accessed next, it is predictable which attribute of the structure will be accessed. This characteristic was exploited to redesign vertex structures for cache efficiency, decreasing the runtime by up

to 40% as shown in Table 5.2. The temporality of these accesses was also considered to explore software prefetching, despite yielding no performance benefit in this case due to the bandwidth-intensive nature of vertex-centric programs.

The third challenge targeted in this chapter was the load imbalance, as presented in Section 5.5. The approach developed in this research was to evaluate the workload by representing it with an edge-centric metric while preserving the user interface. Although this shift is beneficial in 75% of the tests and provides a runtime reduction of up to 57%, Table 5.2 also shows that it degrades performance in 25% of the cases. By contrast, the first-come-first-serve dispatch from the OpenMP `dynamic` scheduling never resulted in performance degradation while still providing a maximum runtime gain equal to that of edge-centric. This, therefore, demonstrates that a better workload proxy is only a part of a well-rounded load-balancing strategy.

Overall, the experiments conducted in this chapter demonstrate that the techniques that have been developed deliver performance benefits in 37 out of the 40 graph-benchmark pairs tested. When successful optimisation techniques are combined, performance benefits are delivered for all graph-benchmark pairs tested, whether they are evaluated from a graph scalability or thread scalability perspective, as shown in Table 5.3 and Figure 5.3 respectively. This demonstrates that although the vertex-centric model exhibits many sources of irregularity, they can be efficiently addressed, effectively reducing the runtime by up to 83%.

Future directions for this work include the integration of work-stealing in the edge-centric workload, for example by designing an affinity schedule tailored for edge-centric. Another direction could be that of incrementalisation [36], which is an optimisation area under-explored in vertex-centric but which could provide a new level of performance.

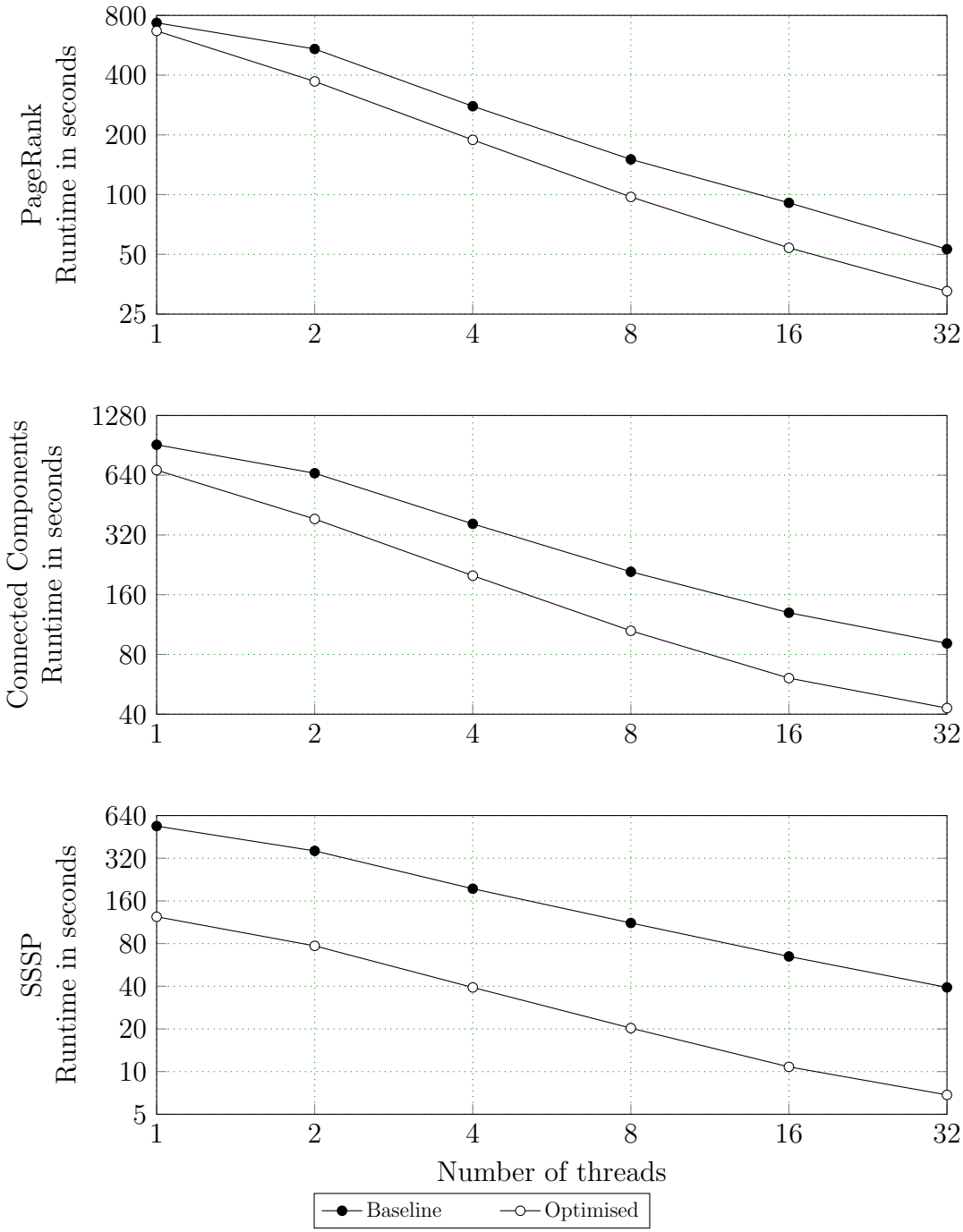

Figure 5.3: Variation of the runtime (in seconds) of the baseline version and the all-optimisations version for each benchmark on the Friendster graph, against the number of threads, using logarithmic (base 2) scales.

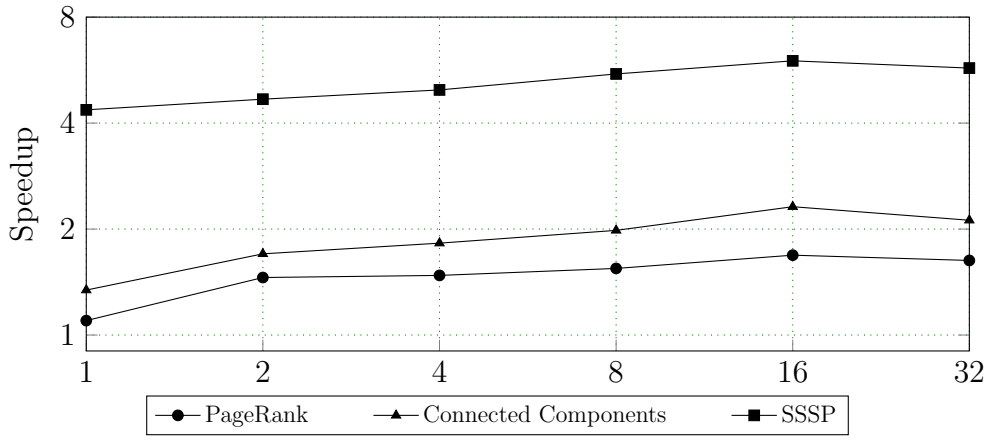

Figure 5.4: Variation of the speedup of the optimised version compared to the baseline version for each benchmark on the Friendster graph, against the number of threads, using logarithmic (base 2) scales.

Chapter 6

Leveraging non-volatile memory

6.1 Introduction

Chapters 4 and 5 demonstrated the effectiveness and feasibility of vertex-centric optimisation techniques that do not sacrifice programmability, in the context of shared-memory parallelism. However, a limitation of shared-memory systems is that applications are restricted to the amount of memory available on a single node. This becomes a serious challenge when considering large-scale graph processing, which is an important activity underlying many technologies. Taking the internet as an example, the fact that so many patterns of web-based interaction, from likes and friends on social networking sites to click-throughs can be represented as a graph data structure means that companies generate vast value from analysing such structures. Moreover, many communities including biological research, transportation planners and communication specialists also derive significant benefits from graph processing. However, with the explosion of data, which is only set to continue, graph sizes are growing exponentially and an important question is how the community can support the processing of these next-generation graphs in the coming decades.

In graph processing, terabytes of memory can be required to hold large graphs, and this is orders of magnitude larger than what can be reasonably held within the DRAM of a single node. This memory limitation is one of the key motivations underpinning the popularity of distributed-memory graph processing [68], and whilst *iPregel* was demonstrated to be significantly more memory efficient than other vertex-centric frameworks, there still reaches a point where it will exhaust the DRAM of a single node. In the distributed approach one is typically scaling their graph processing workloads across nodes due to memory limits rather than being driven by computational concerns, but such an approach can result in numerous disadvantages. The first of which is the need to,

often entirely, rewrite the shared-memory implementation, commonly into some form of message-passing abstraction which requires communications to be explicitly programmed. Secondly, such inter-node communications are likely to result in significant communication overhead if they are unstructured, which is the case for graph processing. Furthermore, graph processing is prone to heavy load imbalance due to the power-law distribution underpinning many graphs, and this forms a major obstacle to efficient distributed graph partitioning. An alternative approach especially popular in graph processing is that of out-of-core solutions [24], where the file system is used as a backing store for data and the DRAM as effectively a manual caching mechanism, fetching the data into DRAM as it is required and flushing unneeded data back to disk. Due to disks being orders of magnitude slower than DRAM, the performance of out-of-core solutions greatly depends on the careful scheduling of those operations. Unfortunately, predicting such behaviour with an irregular, variable, and imbalanced workload is not trivial [3]. In short, graph processing is a prime example of an application where one would ideally stay within the memory space of a single node for as long as possible.

One possible solution to this challenge is the use of commodity NVRAM which has emerged onto the market through Intel's Optane Data Centre Persistent Memory Module (DCPMM). Whilst this is slightly slower than DRAM, although much faster than disk, the significantly increased per-DIMM capacity compared to DDR4 DRAM, means that byte-addressable NVRAM can provide much larger RAM-style memory pools than DRAM alone. Furthermore, NVRAM can act as an extension to DRAM in a transparent manner, resulting in applications being able to leverage this technology without any code changes required. Whilst DCPMM is Intel's specific NVRAM implementation, in this chapter these terms are used interchangeably. Therefore, for large-scale graph processing, the use of NVRAM could be an inflection point, enabling much larger graphs to be processed without incurring the performance overheads of moving to distributed-memory or out-of-core solutions. This chapter explores the use of Intel's DCPMM in the context of vertex-centric graph processing, to understand the role that this could play and the most appropriate techniques to obtain optimal performance. The contributions of the research reported in this chapter are:

- Experimenting with and analysing the scalability of shared-memory vertex-centric graph processing using NVRAM. Resulting in, as far as it can be determined, a new world record for the size of a graph processed within a single node without the use of out-of-core computation.

- Evaluating the need for manual tuning of existing codes to most efficiently exploit NVRAM.

- Quantifying the impact of NVRAM data placement based on the type of memory access performed.
- Discussing the price and power properties of using NVRAM for large-scale graph processing compared to alternate distributed approaches.

The rest of this chapter is organised as follows: Section 6.2 presents related work and introduces the persistent memory technology and vertex-centric before Section 6.3 focuses on the Intel Optane DC Persistent Memory Module. Section 6.4 describes the environment in which experiments are conducted, followed by Section 6.5 which analyses the results obtained before we draw conclusions and discuss potential future work directions in Section 6.6.

6.2 Related work

Interest in non-volatile memory technologies has grown over the past decade [80], with hardware advances in the last two years now resulting in this technology becoming a realistic proposition for use in the data-centre and HPC machines. One such recent NVRAM technology is Intel's Optane DC Persistent Memory (DCPMM) [81]. Released in April 2019, in addition to featuring byte-addressability and non-volatility, the product is provided in a standard DRAM DIMM form factor and at a significantly lower cost per byte than previous DRAM and NVRAM solutions. The byte-addressability means that the CPU can access any location in the DCPMM, effectively meaning that DCPMM can be used as either an extra storage disk or an additional pool of RAM. The focus of this chapter is most interested in the second benefit, where the between five and ten times increase in per-DIMM capacity when compared to DDR4 DRAM, results in the ability to provide very large memory spaces. Therefore, whilst DCPMM is slower than DRAM, it makes it possible to equip nodes with an additional layer of memory hierarchy of TBs in size, at much lower energy and purchase cost than if this was all DRAM. DCPMM's read bandwidth is quoted as around 2.4 times lower than DDR4 DRAM and write bandwidth around 6 times lower than DDR4 DRAM [82], however, this is still far faster than disk and can often be ameliorated in an application either by using the node's DRAM as an additional layer of cache (which is an automatic feature of the technology) or by the programmer explicitly controlling data placement.

Previous studies have been conducted around the use of NVRAM for a variety of applications, including [83] which specifically focuses on the use of DCPMM for scientific codes. An important result has been to show that applications which scale poorly in

the distributed-memory environment, can exploit NVRAM's large memory space to significantly increase the local problem size (effectively the data which can fit into a node's memory space) and ultimately improve performance. This has been highlighted as a key facet of the technology, extending the memory capacity of a node to enable applications most suited to shared-memory operation to reach a scale hitherto unobtainable with DRAM exclusively.

In recent years graph size has grown exponentially to reach today's scale which routinely involves hundreds of billions of edges, and even over a trillion for the largest graphs reported [19, 27, 58]. The amount of memory required to process such graphs increases with graph size and now stands at TBs for the largest graphs. The actual computation required for processing the graph is typically fairly low, with the codes themselves limited by the amount of memory that can be provided. Whilst the ideal is to stay within a single node, when processing the largest graphs the memory required is beyond what the vast majority of machines can hold or reasonably affordable to provide. Traditionally, there have been two possible approaches to tackle processing such large graphs. The first is that of distributed parallelism, often via MPI, but graph processing applications tend to be communication-intensive, resulting in poor inter-node scaling. Furthermore, the high load imbalance frequently found in graphs, especially within social networks, greatly increases the complexity of developing efficient distributed-memory solutions. This was demonstrated in [5], where a 70 trillion edge graph (the overall world record for graph size in a distributed-memory environment as of 2023) was processed on 38,656 compute nodes of Sunway TaihuLight (with each node containing 260 CPU cores). This required over a million CPU cores to process the graph, and in their scaling experiments, they highlighted that performance was limited by the bisection bandwidth due to static routing in InfiniBand, and the increasing volume of the graph cut in their distributed algorithm. The second possible solution is the use of out-of-core techniques, where significant chunks of a large graph are held on disk (typically SSDs) and the DRAM is effectively used as a cache managed explicitly by the programmer. However, this approach also tends to perform poorly, because of the relatively long latencies and low bandwidth of disk accesses.

This is where NVRAM can be of great benefit for graph processing, enabling one to process much larger graphs within a single node before being forced to move to distributed-memory, based on a technology which is hundreds of times faster than disk for access [83]. Furthermore, graph processing represents a workload with a highly irregular memory access pattern, which in itself is an important application pattern to explore within the context of how best one can leverage NVRAM most effectively, with

lessons learnt applying more widely across other codes which also exhibit similar irregular memory access patterns. There have been a small number of previous studies of NVRAM with graph processing, for instance [84] where the authors compared a number of existing graph frameworks on NVRAM without optimising them specifically for this technology, and [85] where the authors developed their own placement algorithm exploiting the asymmetry between NVRAM read and write operation performance. Both of the studies concluded that NVRAM is a promising technology for graph processing. The research reported in this chapter expands upon this existing work by examining the use of NVRAM to store significantly larger graphs. The largest graph examined in this study contains 750 billion edges, compared to 128 billion and 225 billion edges in the previous studies.

6.3 Persistent memory modes

The Intel Optane DC persistent memory modules can be used in different modes, offering different levels of granularity in the control of data placement. These modes can be activated by rebooting the node, which usually takes approximately 20 minutes, making it relatively easy to switch between DCPMM modes without hindering the overall throughput of jobs on a cluster.

6.3.1 Memory mode

The first mode presented in this chapter is referred to as *memory mode*, which is convenient due to it being entirely transparent to applications. This requires no application modification because the NVRAM provided by the DCPMM becomes the main memory space whilst the DRAM effectively becomes the last level cache. By default, all allocations (both static and dynamic) take place within the DCPMM.

6.3.2 App-direct mode

The *app-direct mode* is the second mode presented in this chapter and, unlike memory mode, does not provide automatic access to the DCPMM. Instead, existing DRAM remains main memory while the NVRAM can also be accessed via explicit load and store operations. Allocating memory on the DCPMM can be achieved by mounting a file system upon it, and using a special malloc interface from the `libvmem` library which is part of the Intel's Persistent Memory Development Kit (PMDK) [86].

DCPMM's app-direct mode can also be used in conjunction with the `libvmmalloc`

Table 6.1: Hardware specification of a NEXTGenIO node.

Metric	Value
Processor	2 × Xeon Platinum 8260M 24-core @ 2.4GHz
Volatile memory (DRAM)	192GB (12 × 16GB)
Non-volatile memory (NVRAM)	3,072GB (12 × 256GB)

library, which intercepts all dynamic allocation calls including `malloc`. Without such a library these dynamic allocations would have resulted in data being allocated in DRAM (as described in Subsection 6.3.2) and the library instead allocates them within DCPMM. Using this library, dynamic allocations now take place on the DCPMM while static allocations continue to be placed in DRAM. Furthermore, unlike the general app-direct mode and explicit use of PMDK, this mode does not require application rewriting beyond the inclusion of the `libvmmalloc.h` header file.

6.4 Experimental environment

This section describes the conditions in which our experiments were run, from the hardware and software used, to the graphs and applications selected.

6.4.1 Hardware and software

The experiments presented in this chapter have been run on the cluster built as part of the NEXTGenIO project whose per-node specifications are given in Table 6.1. The cluster contains 34 identical nodes totalling over 100TB of NVRAM and 6.5TB of DRAM. Given the shared-memory nature of the work presented in this chapter, only one node of this cluster was used at any given time during the experiments discussed in Section 6.5.

The *iPregel* framework was compiled with *gcc* 8.3.0, using level 2 optimisations (`-O2`), with support for OpenMP version 4.5 enabled. OpenMP threads are placed on physical cores and pinned to them in order to prevent thread migration. Also, threads are placed on consecutive physical cores; meaning that the first 24 OpenMP threads are placed on the same NUMA region. The libraries `libvmem` and `libvmmalloc` have been used and these can be found in Intel's Persistent Memory Development Kit [86].

Table 6.2: Graphs selected

Name	Number of vertices	Number of directed edges
S-250 / C-250	250,000,000	250,000,000,000
S-750 / C-750	750,000,000	750,000,000,000
Kronecker 25 500	33,554,432	33,554,432,000
Kronecker 28 500	268,435,456	268,435,456,000
Kronecker 33 16	8,589,934,592	274,877,906,944

6.4.2 Graphs selected

Table 6.2 lists the graphs that have been used in the experiments of Section 6.5. Of the five graphs, three of them have been created using a Kronecker graph generator, where the name of these Kronecker graphs contains the parameters to reproduce them. The first number represents the logarithm base 2 of the number of vertices, and the second number is the logarithm base 2 of the average out-degree. The graphs generated vary in sparsity, with an average degree of up to 500 which mimics those typically found in large social network graphs [27].

The two other graphs have been generated using a bespoke graph generator, which provides finer control over vertex adjacency lists. Each graph was generated in two forms, consecutive and scattered, where both versions comprise the same number of vertices, edges and degrees. The difference is in the locality of each vertex's neighbours, where the consecutive version results in the neighbour list of each vertex containing consecutive vertex identifiers. For instance, given vertex i, in this configuration, its neighbours would be $i+1, i+2, ..., i+n$, where n is the number of neighbours. By contrast, the scattered version inserts a gap between any two consecutive neighbours, such that the identifiers of two consecutive neighbours are widely separated. For example, in this configuration given vertex i, its neighbours would be $i+a, i+2a, ..., i+na$, with n the number of neighbours and a the *scattering distance*. These two configurations have been designed to represent extreme cases of memory locality, both consecutive and widely scattered accesses, which enables exploration of the impact of cache and page friendliness in the context of the NVRAM. These graphs are denoted as C-V for the consecutive version, and S-V for the scattered version, where V is the number of vertices in millions. The degree remains 500 in all cases and the scattering distance a is set to 100,000 for the scattered versions. It can be seen that four out of our five graphs are larger than the largest experiments conducted in both previous studies of graph processing on NVRAM [84, 85].

6.4.3 Benchmarks selected

The results presented in this chapter are obtained by running 10 iterations of the vertex-centric implementation of PageRank, whose code is illustrated in Figure 4.4. PageRank is at the core of vertex-centric programming and has become a de-facto benchmark in the graph community. Whilst other commonly used benchmarks, CC or SSSP, are used elsewhere in this thesis, PageRank provides a stable workload across iterations. Crucially, this means that it minimises load imbalance in the sense that every vertex participates towards the calculation at every superstep, whereas other graph applications deactivate vertices as the calculation progresses. This enables experiments to remain focussed on evaluating the performance of the NVRAM hardware and software, without being potentially biased by application or configuration-specific logical behaviour. Moreover, PageRank places the most pressure on the memory subsystem as every vertex broadcasts a message to all its neighbours at every superstep. Therefore the number of messages generated at every single superstep is equal to the total number of edges, placing a high degree of pressure on the memory system thus making it a challenging test of NVRAM performance. As such, whilst it might seem somewhat narrow to focus only on one specific graph benchmark, no additional applications were selected in our experiments because PageRank inherently exposes the characteristics that most accurately explore the role of NVRAM, with the conclusions then applicable to a wide variety of other graph applications.

The experiments presented in this chapter were conducted on two versions of the *iPregel* vertex-centric framework. These are `push` and `pull`, where the versions are alternative implementations of *iPregel*, triggering specific optimisations by redesigning certain parts of the vertex-centric features and tuning them for specific situations. As described in Subsubsection 4.3.4.3, the push version of *iPregel* consists of each sender manually writing into the recipient's memory, where the thread that processes a vertex will write into the memory of each neighbouring vertex, typically held at random locations in memory. Of most interest here, this version generates memory writes at multiple memory locations, in addition to the locks required to prevent potential data-race. By contrast, the pull version consists of the recipient fetching messages from senders. The thread processing a vertex will therefore read from the memory locations of the sender vertices, before writing into the recipient vertex only. In addition to being lock-free, the pull version therefore generates writes that take place at a single memory location. These two versions thus make for two configurations that stress the memory in different ways and provide additional information to aid in analysing the performance of NVRAM under pressure.

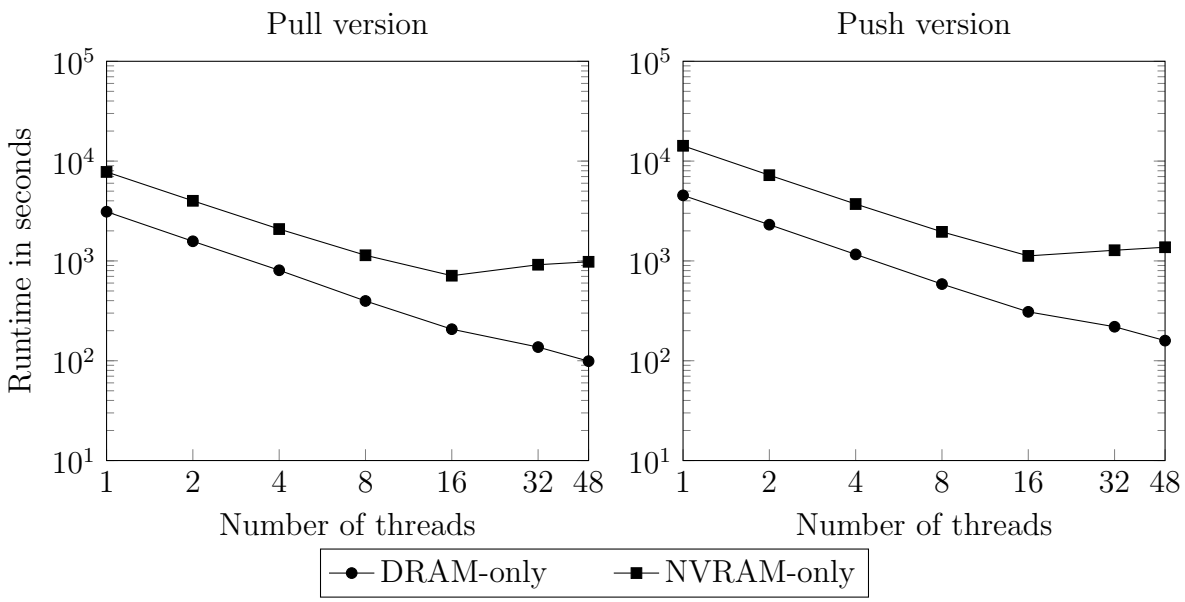

Figure 6.1: Variation of the *iPregel* runtime (in seconds) against the number of threads, for the Kronecker 25 500 graph using different graph memory placements.

6.5 Results

This section presents and analyses the results collected during our experimentation. Multiple data placement configurations were explored to assess the different performance overheads related to the use of NVRAM, and by leveraging the memory modes presented in Subsection 6.3.1, it is possible to control the placement of vertices and edges.

6.5.1 Experiment 1: storing all data in DRAM only

The first experiment presented in this chapter compares processing a graph which is stored exclusively in DRAM, against one exclusively in NVRAM. By comparing the performance, any overhead imposed by the use of NVRAM can be identified.

The app-direct mode is used to place a graph entirely in DRAM, where dynamic allocations are by default placed on the DRAM. By contrast, to place the graph in NVRAM the `libvmmalloc` library, described briefly in Subsection 6.3.2, is used. When using this library dynamic allocations are automatically intercepted and their NVRAM-equivalent is instead issued.

The graph selected is the Kronecker 25 500, which is a graph small enough to fit within the 192GB of DRAM available on a single node. Nonetheless, it remains a graph that has over 30 billion edges, which is larger than most graphs processed by shared-memory frameworks or publicly available [75, 76]. The results gathered from this experiment are illustrated in Figure 6.1, where it can be seen that there is always an overhead observable

between the DRAM and NVRAM placement, as expected. Irrespective of the parallel configuration, up to and including 16 OpenMP threads, NVRAM placement of the graph is approximately 2.5 times slower than the DRAM counterpart for the pull version, and 3 times slower for the push version.

There is a noticeable performance drop for the NVRAM-only placement at 32 and 48 threads, and this is explained by how NVRAM-only placement is implemented. Using this approach, dynamic memory allocations are supplied from a memory pool built upon a memory-mapped file. This memory-mapped file can only be created on either the first or second socket, meaning that only the NUMA region local to that socket will be local to that memory pool. A socket contains 24 physical cores on this NEXTGenIO cluster and utilising fewer than 24 OpenMP threads, given the OpenMP placement configuration adopted here, results in those OpenMP threads being pinned to physical cores on the socket which is local to the memory-mapped file. However, 32 and 48 OpenMP thread configurations result in threads also being mapped to physical cores of the other socket, and hence accessing the memory-mapped file in a cross-NUMA fashion which impacts performance. Until this cross-NUMA configuration is reached, the use of NVRAM memory mode does not hinder parallel scaling, either for the push, or the pull versions.

6.5.2 Experiment 2: increasing the size of the graphs

Unlike the experiment presented in Subsection 6.5.1, the next experiment aimed to explore the use of DRAM and NVRAM working together. To achieve this, DCPMM's `memory mode`, as described in Subsection 6.3.1, was used which automatically places data in NVRAM and uses DRAM as the last-level cache.

The three Kronecker graphs (see Table 6.2) are used for this experiment, as they are designed to gradually increase the pressure on the non-volatile memory as their size grows. The first graph is the Kronecker 25 500, the 30 billion edge graph used in experiment one and small enough to fit entirely in DRAM. The second graph selected is Kronecker 28 500, and with 270 billion edges requires more than 5 times the memory available in DRAM. The third graph is Kronecker 33 16 which also contains approximately 270 billion edges, however, this graph holds 30 times more vertices, exceeding 2^{32}. Such a number of vertices requires vertex identifiers to be encoded using 64-bit integers instead of 32-bit, effectively doubling the amount of memory required to store the edges. Moreover, the number of edges per vertex on this last graph is 30 times lower than that on the Kronecker 28 500 whose vertices are more densely interconnected, enabling an evaluation of the performance of automatic caching in DRAM by using very different graphs.

The timings collected during this experiment are reported in Figure 6.2 and, as to be

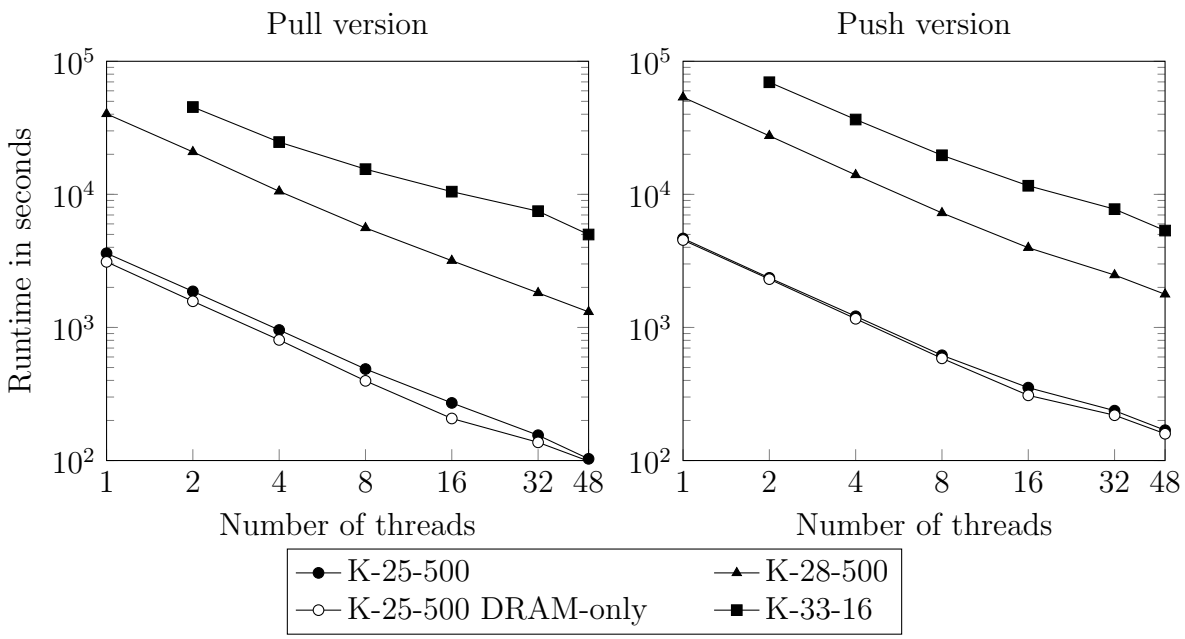

Figure 6.2: Variation of the *iPregel* runtime (in seconds) against the number of threads used, for different graph configurations, using NVRAM memory mode (including the K-25-500 DRAM-only data for reference).

expected, runtime increases as the graph size grows. The Kronecker 25 500, which can fit entirely in DRAM, only requires a single movement of data from NVRAM to cache this in DRAM. The two other graphs however cannot fit in DRAM alone and therefore require multiple data movements between NVRAM and DRAM as data is evicted from this last level cache.

It can be observed that whilst both the Kronecker 33 16 and Kronecker 28 500 graphs contain a similar number of edges, the processing of the former performs worse than the latter. The crucial difference here is in the number of vertices, with Kronecker 33 16 containing 30 times more vertices than Kronecker 28 500. Storing the 280 million vertices of the Kronecker 28 500 graph requires approximately 10GB in *iPregel*, which can fit in DRAM. By contrast, storing 30 times more vertices consumes over 300GB of memory, exceeding the total amount of DRAM available. As a result, not all vertices can be held in DRAM at once and as a superstep progresses, vertices must be evicted from DRAM to NVRAM.

This experiment has provided an evaluation of the the performance of using DRAM and NVRAM together. However, whilst DCPMM's memory mode enables the use of a much larger memory pool without requiring application rewriting, the initial placement of all data on NVRAM regardless of their access pattern is likely sub-optimal. This can have serious implications for performance when edges evict vertices from the DRAM

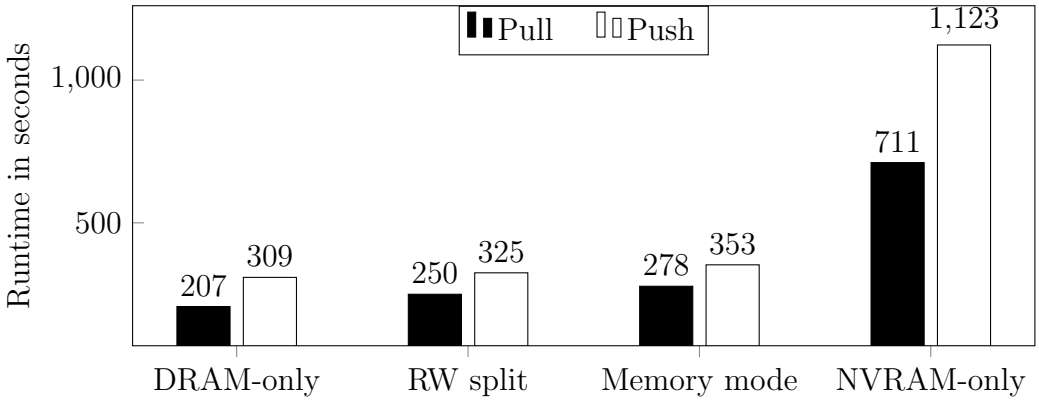

Figure 6.3: Variation of the *iPregel* runtime (in seconds) on the Kronecker 25 500, using multiple data placement configurations, for both push and pull versions at 16 threads

cache since the NVRAM overhead of the latter is 3 to 4 times bigger.

6.5.3 Experiment 3: exploring the difference in performance between read and write NVRAM operations

The second experiment does not take into account the esoteric property of NVRAM, where the overhead involved in read and write operations is asymmetric (write operations are over twice as slow as read). Therefore, a hypothesis was that to most optimally leverage NVRAM, one should tune their application to fit these differences as appropriate.

To minimise the penalty of NVRAM's write overhead, data should be placed on DRAM or NVRAM based on its access pattern. It follows that DRAM should therefore be privileged for data that is read-write, while NVRAM should be ideally kept for read-only data. In the case of *iPregel* for instance, vertices are writable while edges are read-only. An experiment was therefore run to exploit this property, where vertices are placed in DRAM and edges in NVRAM. Furthermore since NVRAM modules are plugged into DRAM slots, they are subject to Non-Uniform Memory Access (NUMA) effects. Therefore, to maximise performance, the placement of edges on NVRAM should be NUMA-aware, resulting in placing edges on the NVRAM NUMA region corresponding to that of the vertex from which they are outgoing.

App-direct mode is used to manually place data on the DRAM or NVRAM, where a file system is mounted on each NVRAM NUMA region which is then accessed via PMDK's `libvmmem` library. The first step involves allocating a memory space on the NVRAM, from which a pointer is returned. Subsequently, this pointer is then passed to a decorated set of functions equivalent to classic malloc functions, which perform the actual allocation on the NVRAM area pointed to.

Figure 6.3 reports the runtimes obtained by applying the read/write split technique on the Kronecker 25 500 graph (30 billion edges). Performance is compared against results collected in previous experiments (DRAM-only, Memory mode, and NVRAM-only). The performance observed is bounded by the DRAM-only and NVRAM-only configurations, where for both the push and pull versions the DRAM-only configuration remains the fastest, and NVRAM-only the slowest. It can be seen that the RW-split and memory mode experiments provide similar performance, albeit with the memory mode approach being slightly slower in both cases. Therefore, pinning verticies in DRAM, compared to memory mode that may evict them and flush them back to NVRAM, does provide a marginal performance improvement. It should be highlighted that as this graph fits into DRAM, the data movements performed by memory mode are simpler.

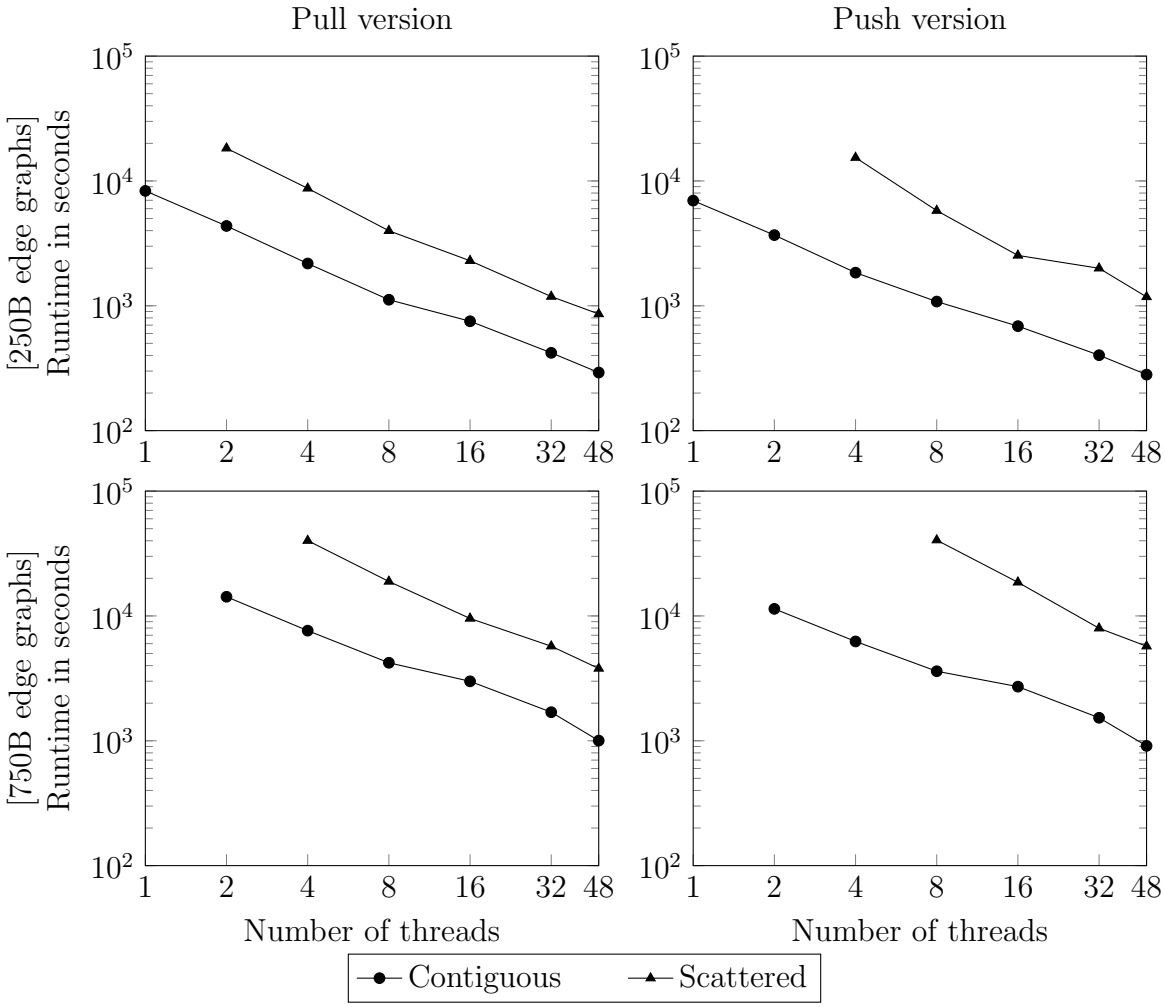

Figure 6.4: Variation of the *iPregel* runtime (in seconds) against the number of threads used, on the contiguous and scattered versions of the 250 and 750 billion edge graphs. (Missing results are due to excessive runtime)

6.5.4 Experiment 4: the impact of data locality and paging

A fourth experiment was designed which evaluates the impact of memory locality which involves two versions of each graph that share the same size both in terms of vertices and edges but differ in terms of how they are connected. This is an important property as it influences the efficiency of caching during graph processing.

As described in Section 6.4, two graphs have been designed, each with a contiguous and scattered version. Contiguous versions contain vertices whose neighbours are consecutive identifiers, whereas the scattered versions contain vertices with neighbours that are wide apart from each other. The NVRAM memory mode presented in Subsection 6.3.1 was selected so that its ability to automatically page data between NVRAM and DRAM can be explored. Furthermore, the graphs generated consisting of 250 and 750 billion edges respectively, enabling an evaluation of the performance of NVRAM under significant memory access load.

Figure 6.4 depicts the performance observed when processing synthetic graphs C/S-250 and C/S-750 respectively. Whilst, as would be expected, absolute runtime is less for the 250 billion edge graphs, scalability remains similar when tripling the graph size to 750 billion edges. The graphs with scattered memory access patterns exhibit poorer performance than those with consecutive accesses. This is to be expected and due to the extra paging operations required when recipient vertices are not already in DRAM, which is much more likely to occur on the scattered configurations. This overhead varies between 3.19 and 6.85 times, with 4.99 times being the average, depending on the graph size and *iPregel* version used, albeit then remaining constant as the number of threads increases.

The performance observed on NVRAM for these graphs is reported in Table 6.3 using the metric billion edges traversed per second (Giga-Traversed Edges Per Second or GTEPS). As already observed, scattered neighbours result in a significant performance overhead compared to the contiguous graph versions, and as far as can be reasonably deduced, the 750 billion edge graph runs are a new world record for the size of the graph processed within a single node without the use of out-of-core computation. For comparison, PageRank with similarly sized synthetic large graphs when processed using the latest out-of-core solutions typically ranges between 0.07 and 0.83 GTEPS [5] and distributed *GraM* over 64 servers results in 8.6 GTEPS [5], as shown in Table 6.3. *GraM* was the fastest reported PageRank implementation for large graphs until the ShenTu implementation which significantly outperforms all of these approaches by two to three orders of magnitude over 38,656 compute nodes. Whilst the performance of ShenTu is impressive, realistically even for large companies such as Facebook or Google, such a

Table 6.3: Maximum number of billions of edges traversed per second (in GTEPS) by the pull and push versions of *iPregel*, on all graphs considered in this section, running over 48 threads

Graph	Pull version	Push version
C-250	10.12	9.56
S-250	3.26	2.13
C-750	8.61	8.63
S-750	2.35	1.35
Kronecker 25 500	3.35	2.10
Kronecker 28 500	2.31	1.55
Kronecker 33 16	0.61	0.54

specialist and expensive system is likely a difficult proposition.

6.5.5 Performance summary

Across the four experimental configurations tested, certain performance patterns and overheads were observed. Table 6.3 reports the performance in billion edges traversed per second (GTEPS) resulting from experiments conducted in this section.

As expected, the highest performance is delivered for the contiguous version of the graph comprising 250 billion edges (C-250). This is due to the optimal data locality and load balancing, which enables both *iPregel* versions to achieve approximately 10 GTEPS. Between 85 to 90 percent of this performance was preserved when moving to the C-750 graph which contains three times as many edges, resulting in approximately 8.6 GTEPS for both the pull and push *iPregel* versions.

The performance observed on their scattered counterparts however is noticeably lower due to poor data locality. Performance achieved for the scattered graphs is 3.1 to 3.6 times lower than those seen for contiguous graphs with the pull version, and 4.5 to 6.4 times lower for the push version. This difference is explained by the access pattern paired with the asymmetric read-write overhead of NVRAM. The pull version of *iPregel* fetches broadcast messages from neighbours (read) and combines them on the vertex being processed (write). Thus all writes are located on the vertex which is being processed by the executing thread. Conversely, with the push version of *iPregel*, when a thread processes a vertex then as messages are produced by that vertex these are immediately placed into the recipient's mailbox (or combined with an already present message). Therefore, writes are located on every neighbour of the vertex being processed, and no longer local but are instead remote. Although both *iPregel* versions encounter scattered memory accesses

when processing the S-250 and S-750 graphs, only for the push version does the location of write accesses change. Therefore this version of *iPregel* is more impacted by the write-specific additional overhead imposed by NVRAM.

When considering the Kronecker graphs the pattern of performance for each *iPregel* version is initially similar. When moving from the K 25 500 graph to the K 28 500 graph, the latter comprising 8 times as many vertices and edges, this results in a performance reduction of approximately 30% for both *iPregel* versions. However, the performance reduces significantly, by approximately 70%, when moving from the K 28 500 to the K 33 16 graph. This performance decrease is explained by the increased number of vertices, over 8 billion, for the K 33 16 graph. Whilst such an increase enlarges the workload, the main issue is that at this scale a change in the type used to identify vertices is required. Eight billion is beyond what can be encoded by a 32-bit unsigned integer type and therefore this number of vertices requires the use of a 64-bit type instead. Whilst the consequence for the vertex data structure is negligible, requiring only an additional 4 bytes per vertex structure, it is far more significant for the edges. Such an increase in datatype size results in each edge requiring double the amount of memory, resulting in longer loading times as well as worse data locality since the cache now contains 50% fewer edges.

It is also instructive to compare the performance of our approach on NVRAM reported in Table 6.3 with other popular graph processing frameworks which do not use NVRAM. Table 6.4 is reproduced from [5] and illustrates the performance achieved by these other graph processing frameworks with similar-sized graphs. Whilst the exact configuration of these graphs is not made explicit in [5], and likely a number of different configurations are used between frameworks, several comparisons can still be made. Firstly frameworks including G-Store, Graphene, and Mosaic utilise an out-of-core approach for processing these large graphs within a single node. Irrespective, this results in very poor performance which is typically significantly lower than all performance figures reported when running *iPregel* on NVRAM, apart from when the Kronecker 33 16 graph is processed due to the issues highlighted in this section. Whilst each of these frameworks is utilising a server with SSDs rather than spinning hard disks, clearly, the performance delivered by such hardware for large-scale graph processing falls significantly short of that delivered by NVRAM.

Whilst the parallelised in-memory frameworks *Giraph* and *GraM* deliver much greater performance than the out-of-core solutions in Table 6.3, the results obtained on NVRAM demonstrate that a single-node NVRAM approach to graph processing is still competitive for many graph types. Moreover, *Giraph* and *GraM* deliver only 0.028 GTEPS and 0.134

Table 6.4: Performance (in GTEPS) of other graph processing frameworks running with similar-sized graphs to ours, data reported in [5] and reproduced here for comparison against NVRAM results

Framework	In-memory or out-of-core	Configuration	Performance (GTEPS)
Giraph	In-memory	200 nodes	5.6
GraM	In-memory	64 nodes	8.6
ShenTu	In-memory	1024 nodes	72.8
Chaos	Out-of-core	32 nodes (480GB SSD each)	0.07
G-Store	Out-of-core	1 node (8x512GB SSDs)	0.23
Graphene	Out-of-core	1 node (16x500GB SSDs)	0.83
Mosaic	Out-of-core	1 node (6 NVMe SSDs)	0.82

GTEPS per node respectively. Performance wise, the framework that stands out in Table 6.3 is ShenTu which, for a similar-sized graph achieved 72.8 GTEPS over 1024 nodes. However, it should be noted that this is equivalent to only 0.07 GTEPS per node.

6.5.6 Additional metrics

In addition to raw performance, other metrics such as purchase cost and energy efficiency are also important when one is considering a specific hardware solution such as NVRAM. A node from the NEXTGenIO cluster contains hardware totalling approximately £20,000. Comparatively, considering the Sunway TaihuLight supercomputer as an example, 96 nodes are needed to obtain the same amount of memory. With the total cost of the 40,960-node supercomputer estimated at 273 million dollars [87], a per-node price of 6,665 dollars is estimated (approximately £5,000). Therefore, processing the graphs used in this chapter, which occupy 3,072GB of memory, either requires a single £20,000 NEXTGenIO node, or 96 × £5,000 Sunway TaihuLight nodes totalling £480,000. From a cost perspective alone there would be an approximate saving of £460,000 by adopting a non-volatile memory approach. Of course in such a scenario, the amount of computational processing power that can be leveraged by 96 nodes of the Sunway TaihuLight supercomputer far exceeds that of a single NEXTGenIO node; however, as described in Section 6.2, typically vertex-centric graph processing is not computationally bound and there is a communication overhead involved in a distributed-memory approach.

Energy and power usage is another metric that has become increasingly important over recent years. The NEXTGenIO and Sunway TaihuLight nodes share a power consumption

at approximately 400W per node, 375W for TaihuLight[1] and 430W for NEXTGenIO [83]. Therefore, in terms of power draw, which can be a major limit for data-centre machine rooms, the graph processing workload running over NEXTGenIO will draw approximately 84 times less power when at full load at any one point in time. It is also instructive to compare the overall energy consumption in terms of energy-to-solution, which also requires taking into account the execution time. As NEXTGenIO and Sunway TaihuLight nodes share the same power consumption this means that such calculations to be simplified to runtime alone. When processing one iteration of PageRank on Kronecker 34 16, which is twice as many vertices and edges as the Kronecker 33 16, the ShenTu framework using 1024 nodes from the Sunway TaihuLight supercomputer reaches 72.8 GTEPS, equivalent to 0.07 GTEPS per node. By contrast, when processing one iteration of PageRank on Kronecker 33 16, *iPregel* using a single node from the NEXTGenIO cluster reaches 0.60 GTEPS. This is 8.5 times more GTEPS, using 15% more energy, for a graph twice as small. Unless the runtime grows as a cubic of the graph size, the NEXTGenIO node proves to be more efficient when considering the energy-to-solution.

6.6 Conclusion and further work

This chapter has discussed and analysed multiple experiments covering numerous scenarios in testing the performance of NVRAM for processing large-scale graphs using the vertex-centric methodology. These experiments have been designed to evaluate the performance of NVRAM when used in isolation, or in conjunction with DRAM either implicitly or directly by the programmer. Two versions of the *iPregel* vertex-centric framework which exhibit different memory access patterns have been used as a vehicle to drive the experiments, with 5 different graphs including 2 specifically designed to provide fine control over data locality.

Whilst the focus of this chapter has been on the vertex-centric methodology, the results and conclusions drawn can be more widely applied to both other graph processing technologies and codes with similar irregular memory access patterns. It was found that NVRAM permits, without code rewriting, a shared-memory framework such as *iPregel* to seamlessly scale to a graph of 750 billion edges, equivalent to 75% of the Facebook graph [27]. As far as can be reasonably determined, this is a new world record for the largest graph ever processed by a shared-memory system without the use of out-of-core computation and makes NVRAM a crucial enabler in reaching new horizons in shared-memory graph processing within reasonable purchase cost and energy usage. Therefore

[1]Calculated from the results submitted to Top500, which specify an average power consumption of 15,371kW across 160 supernodes, each containing 256 nodes.

for technology companies, such as Facebook and Google, who have large graph processing requirements and vast data centres, NVRAM is a technology that should be considered a serious contender as having a role in their overarching hardware strategy. This is not least because such a step change in single-node graph processing capability can be delivered by NVRAM at a reasonable price and power usage point compared with other options.

The impact of data placement on performance was observed and demonstrated that, if one is willing to invest time in tuning their code for NVRAM, then manually placing data based upon the access pattern provides better performance than that of automatic placement. Nonetheless, the multiple modes available to make use of NVRAM allows non-experts to leverage this technology without having to rewrite their application or their framework, and experts to rewrite parts of their software to make the most of this technology.

When considering future work, in addition to being an enabler for shared-memory graph processing systems, NVRAM could be beneficial in distributed-memory graph processing systems too. In contrast to the Sunway TaihuLight supercomputer, where the full Sogou graph (270 billion vertices and 12 trillion edges) required at least 10,000 nodes [5], holding the Sogou graph in NVRAM memory could be achieved with fewer than 50 nodes based upon the NEXTGenIO cluster specifications. Furthermore, it was demonstrated in Section 6.5.5 that on a node-by-node basis the performance of existing distributed in-memory graph processing frameworks tends to be poor. As a next step, it would be instructive to explore how the emergence of clusters comprising large-memory nodes could influence the design of distributed-memory graph systems and algorithms. This will potentially deliver the capability to process extremely large graph by combining the single-node performance enabled by NVRAM with the memory capabilities of multi-node NVRAM.

Chapter 7

A distributed-memory implementation of iPregel

7.1 Introduction

The competitiveness of non-volatile memory (NVRAM) for processing large-scale graphs was demonstrated in Chapter 6. Furthermore, NVRAM also proved to be an enabler for shared-memory frameworks, by significantly extending the boundary of their fundamental limitation, memory. Nonetheless, this fundamental limitation still holds and NVRAM just delays the exhausting of memory. The amount of memory available in a single node is still a blocker for processing very large-scale graphs..

As of 2022, traditional HPC cluster nodes commonly contain between 256GB and 512GB of RAM, thus rapidly totalling terabytes of RAM with only a few nodes. The distributed-memory architecture, therefore, remains an important target, and ultimately the only realistic in-memory approach for graphs requiring more memory than what a single node can provide. However, the vertex-centric programming model comprises numerous challenges, as illustrated throughout previous chapters. Such challenges are further aggravated when moving to distributed-memory solutions, as they can also impact network communications, which are already a major limitation of distributed-memory graph processing approaches.

Existing distributed-memory vertex-centric frameworks have relied upon numerous message wrapping and buffer exchange approaches to achieve the communication phase needed at every superstep. Whilst these aim to lower the overhead of communications, such designs incur additional computations and involve memory access patterns which typically offset the benefits of distributed memory parallelism, especially on a small number of nodes [12].

However, the majority of real-world graphs [68] are reported as being orders of magnitude smaller than the record-breaking graphs presented in literature [5]. Although research can aim at developing solutions to prepare for long-term challenges, it is not incompatible with also tackling present and short to medium term ones. In the vast majority of cases, graphs will not be processed with tens of thousands or millions of cores, either due to the lack of budget or the graph size not requiring this amount of resources. For these real-world, practical, cases, a distributed-memory implementation that is efficient, and scales to modest node counts is highly desirable and will be directly usable for state of the art graph processing.

To that end, the shared-memory framework *iPregel* was ported to distributed-memory and in doing so this research:

- Proposes a new buffer design which, on the sender-side, avoids the appending of recipient vertex identifiers along with messages and the use of computationally intensive hash-based indexing. By doing so this avoids the need for a pre-exchange sort, reducing the message-sending algorithmic complexity to linear. Furthermore, on the receiver side, this technique removes the need for both message unwrapping and dispatch, as well as more generally enabling support for network offloading optimisations.

- Leverages MPI-3 shared-memory to implement an intra-node buffer sharing optimisation that reduces the buffer memory footprint from a replicated cost per MPI process down to per node. This decreases the pressure on the network bandwidth and improves MPI reduction parallelisation.

- Develops an interval-based message processing technique where the maximum amount of memory occupied by the framework's buffers can be specified. This allows the framework to adapt to memory-constrained environments, for example existing compute resources available for graph processing, that would have otherwise resulted in an out-of-memory failure.

Section 7.2 explores the evolution of distributed-memory vertex-centric frameworks, along with the different directions investigated and the challenges that arose. Section 7.3 then introduces the framework developed in this research, *DiP*, from its interface and architecture to buffer design. Then, optimisations developed for the buffer design are presented and explained in Section 7.4. The experimental environment is outlines in Section 7.5, before the results obtained for *DiP* are discussed and analysed in Section 7.6. Finally, this chapter concludes in Section 7.7 summarising the findings of this investigation, as well as future research directions that would be interesting to pursue.

7.2 Related work

Over the years, several distributed-memory vertex-centric frameworks have been developed, starting with the original *Pregel* framework back in 2010. In vertex-centric programming, distributed-memory frameworks must handle a phase that shared-memory frameworks do not, which is the carrying of messages across distributed-memory workers. Shared-memory accesses can no longer be systematically employed because, by definition, such distributed-memory workers may reside on physically distinct nodes. Since network communications are known to be the major limitation of distributed-memory solutions, attention must be given to the design of this extra phase and several approaches have been developed.

The first approach, adopted by *Pregel+* [12] and *GraphD* [25], consists of appending messages to a single queue. Before being added to the queue each message is prepended with the vertex identifier of the recipient, and messages are therefore queued in an unordered fashion. However, applying a sender-side combination would be particularly inefficient in this configuration because, for each recipient, the entire queue must be parsed to locate corresponding messages and combine them. This step would thus be of complexity $O(n^2)$ and, to avoid this, a sort is applied at the end of the vertex processing phase. The C++ Standard Template Library (STL) sort is used for *Pregel+*, with both algorithms being of complexity $O(n \times log_2(n))$. The combination pass with a sorted queue completes in linear complexity, reducing the total algorithmic complexity to $O(n \times log_2(n) + n)$.

In *HAMA* [88], a similar design is used, except that the queue is built as a hashmap indexed on the vertex identifier. Messages no longer need to be individually appended to the vertex identifier, as they are now grouped per recipient vertex. The major benefit of this approach is that it avoids needing to sort the queue before combining, resulting in an algorithmic complexity of $O(n)$. However, the constant in the $O(1)$ complexity of hashmaps indexing must not be ignored, as it may represent a non-negligible overhead in the entire runtime across large graphs.

The major drawback of these with these approaches is that the sender-side combination is applied only after all messages are queued. Consequently, on each distributed-memory worker, the queue can grow to a maximum size equal to the number of edges to which its local vertices are connected. As graphs typically contain order(s) of magnitude more edges than vertices, this limits the overall size of the graph, as large-scale graphs with many edges will exhaust available memory resulting in an out-of-memory error.

Although implementations could be modified to undertake combinations immediately upon message queuing, this would still involve a second phase on the receiver side. This

is because, regardless of whether they are sorted on the recipient vertex identifier, messages queued are not guaranteed to be consecutive. Therefore, the distributed-memory worker on the receiving end must dispatch each message to its recipient. During this step, for each queue entry, the message along with its vertex identifier are extracted and dispatched into the corresponding recipient vertex mailbox, as well as combining it with the existing message, if any. Although this single-pass step is of linear complexity $O(n)$, such random memory accesses are likely to result in significant overhead. Moreover, the network communications are implemented as individual send-receive pairs, resulting in the underlying technology being unaware of any reduction taking place and therefore unable to leverage potential optimisations. Such optimisations, commonly provided by supercomputers and advantageous to HPC workloads, consist of performing certain operations directly on network hardware such as the processing of MPI collectives on network switches [89], or MPI non-blocking all-to-all on smart network interface cards [90].

As seen in the approaches described above, the design of the queue, or buffer, used in the message exchange has consequences beyond network communications including requiring a sender-side sort to an extra receiver-side message dispatch. Based on this, the hypothesis is that a buffer design which reduces the number of additional steps as well, as better expressing the combination intent during the buffer exchange to the underlying distributed-memory technology, would be beneficial.

7.3 Overview of DiP

The *DiP* framework is the approach developed in this research for tackling the challenges presented in Section 7.2. Standing for Distributed iPregel, it combines the overarching philosophy and design concepts of *iPregel*, which proved to be successful in shared-memory parallelism, with techniques crucial for distributed-memory performance.

7.3.1 Interface

iPregel has served this research by being a vehicle for testing and developing techniques for efficient shared-memory vertex-centric graph processing. However, whilst the framework has changed internally due to these techniques, the API exposed to the programmer has remained constant. This demonstrates the benefit of adopting a clear, and simple API, and the interface adopted for *DiP* has therefore been strongly inspired by *iPregel*.

Similarly to its shared-memory counterpart, the *DiP* framework requires two functions to be defined by the user: `dip_compute` and `dip_combine`, listed in Figure 7.1. Their semantics are identical to that in *iPregel*, thus providing a smooth transition from *iPregel*

```
1  /**
2   * @brief This function performs the actual superstep calculations of a
3   * vertex.
4   * @param[inout] me The vertex to process.
5   * @pre The vertex pointer given points to an allocated memory area
6   * containing a vertex.
7   * @pre This function must be defined by the user.
8   * @post The vertex specified has finished its work for the current
9   * superstep.
10  **/
11 void dip_compute(dip_vertex* me);
12
13 /**
14  * @brief This function combines two messages into one.
15  * @param[inout] old The existing message in the vertex mailbox.
16  * @param[in] new The message that arrived for receipt.
17  * @pre The variable old points to an allocated memory area containing a
18  * message.
19  * @pre The variable new points to an allocated memory area containing a
20  * message.
21  * @pre The operation defining the combination is associative and
22  * commutative.
23  * @pre This function must be defined by the user.
24  * @post The variable old contains the combined value of old and new.
25  **/
26 void dip_combine(DIP_MESSAGE_TYPE* old, DIP_MESSAGE_TYPE new);
```

Figure 7.1: User-defined functions of *DiP*

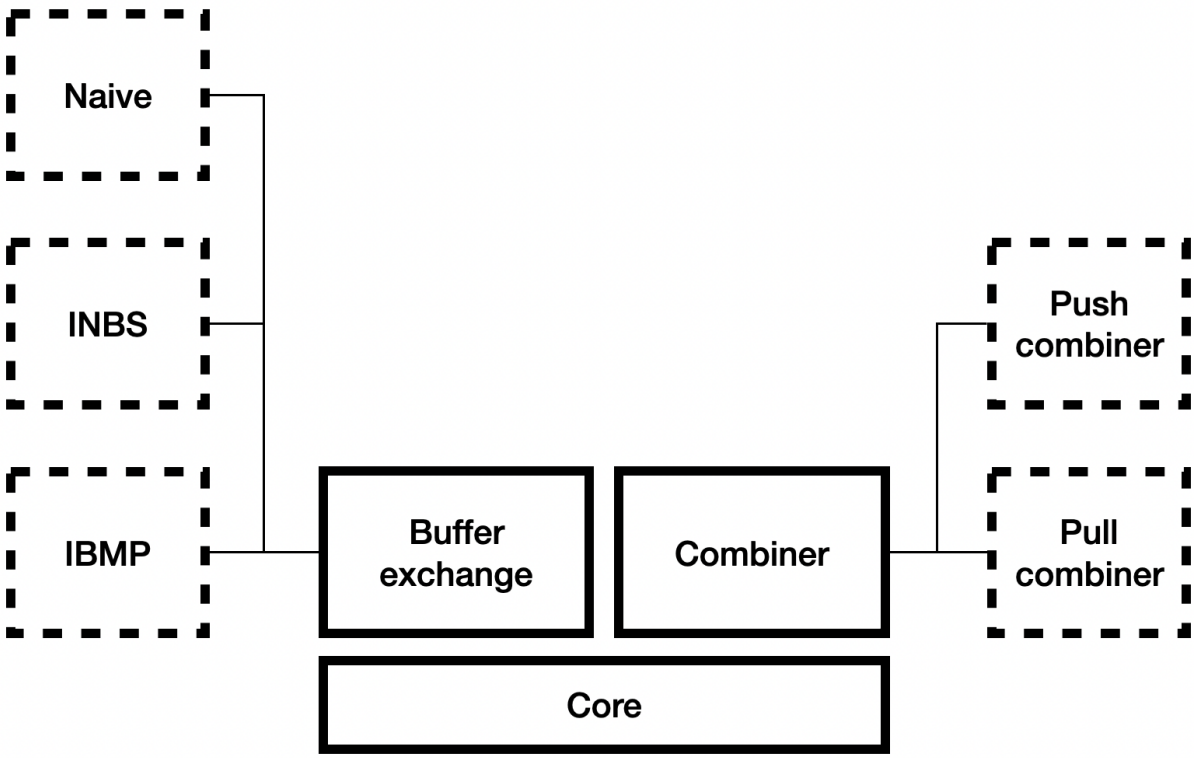

Figure 7.2: Structure of the *DiP* framework

to its distributed-memory counterpart. Unlike *iPregel* however, there is an optional compilation flag that directs whether the combination function expressed in `dip_combine` matches one of the predefined MPI reduction operations. If this is present then *DiP* will leverage the underlying highly-optimised MPI predefined reduction operation instead of that defined by the user.

In addition to the user-defined functions listed in Figure 7.1, there are also functions provided by the *DiP* framework for tracking supersteps, or getting the graph size and order. Some of these are described in Appendix H. Despite the different methods of parallel computing, this set of functions maps to that of *iPregel*, permitting a smooth porting of *iPregel* applications to *DiP*.

7.3.2 Architecture

Similarly to its shared-memory counterpart, the *DiP* framework is developed in C and parallelised within a node using the shared-memory OpenMP [45]. To go beyond single-node development, the distributed-memory aspects were written following the Message-Passing Interface (MPI) standard [46].

One of the *iPregel* characteristics is modularity and, as shown in [1–3], this proved

to be a valuable feature throughout the evolution of the *iPregel* framework. Modularity therefore played a major role in the design of the *DiP* framework, and, as shown in Figure 7.2, the *DiP* framework contains multiple implementations of modules. Similarly to *iPregel*, these modules operate independently, where any combination is possible. The combiner module is the distributed-memory implementation of *iPregel*'s combiner, containing both push and pull versions whose semantics are identical to that in *iPregel*. The push combiner consists of putting generated messages into the recipient's mailbox and combining them with the existing message. The pull combiner leverages a synchronisation-free approach, as was demonstrated to be beneficial for the shared-memory implementation, where every MPI process issues writes only to its locally-held vertex mailboxes.

iPregel's vertex addressing module described in Subsubsection 4.3.4.4 does not exist in *DiP* because the direct mapping technique became the choice adopted in *DiP* for vertex addressing. During the investigation into the NVRAM technology presented in Chapter 6, the graphs used in the evaluation of this research grew beyond what can be publicly found. Consequently, synthetic graphs are generated which are typically built on consecutive 0-index vertex identifiers, making direct mapping the most appropriate addressing technique.

The buffer exchange module is specific to *DiP* and provides different means of undertaking data communications during the buffer exchange phase. This will be discussed further in Subsection 7.3.3.

7.3.3 Buffer design

In the *iPregel* framework every vertex can reach any other vertex using direct memory accesses, regardless of which threads were holding the sender and recipient vertices. In *DiP*, however, this no longer applies as the distributed nature means that neighbours of a given vertex may be held by an MPI process residing on a different node. Consequently, although shared-memory accesses are still a viable solution for node-local neighbours, a mechanism must be provided for node-remote neighbours.

The design presented in this section relies on replicated buffers, which comprise a pair of buffers allocated on every MPI process. To leverage the direct mapping vertex addressing discussed in Subsubsection 4.3.4.4, each of the two buffers contains a number of elements equal to the graph order. The first buffer, known as the *flag-buffer*, contains boolean elements that indicate whether the corresponding vertex takes part in inter-vertex communications. Elements in the flag-buffer are initialised, and reset at the end of every superstep, to `false`. The second buffer, the *value-buffer*, holds elements of

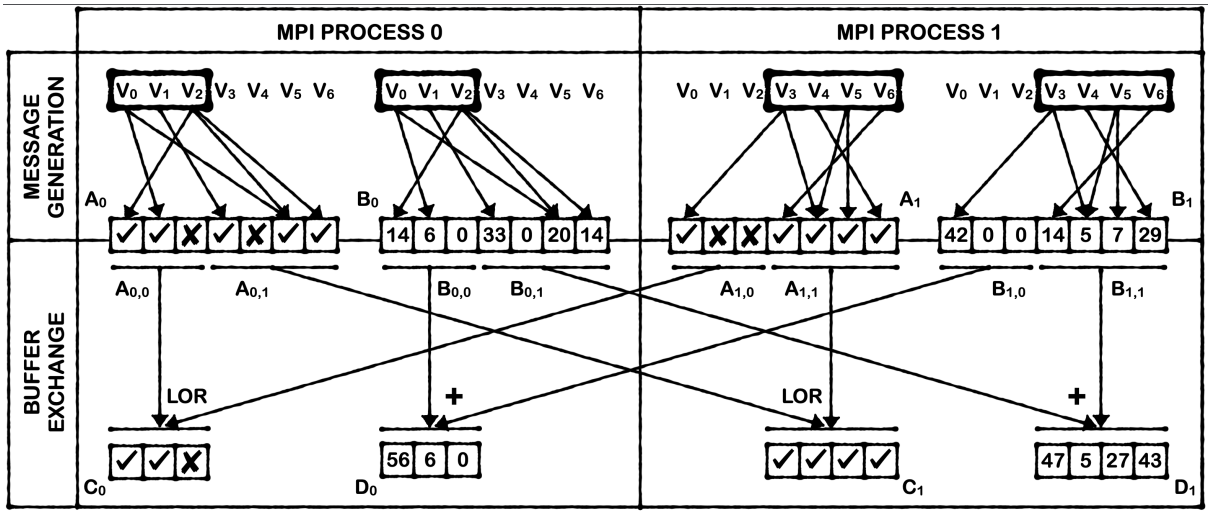

Figure 7.3: Workflow of the buffer exchange for the push version in the *DiP* framework, assuming a sum combination operation.

the message type `DIP_MESSAGE_TYPE` defined by the user. These elements contain the actual message content exchanged with the vertex, and the default value, which is also the value to which they are reset at the end of every superstep, is the neutral value of the message combination operation defined by the user. The value contained in an element of the value-buffer is therefore significant only if the corresponding element in the flag-buffer contains the value `true`.

The underlying buffer exchange may be designed differently therefore based upon the version of the combiner that has been used.

7.3.3.1 Push version

An example of a message exchange for the push version is shown in Figure 7.3. For simplification purposes, the configuration illustrated contains only two MPI processes and a graph with seven vertices. As explained above, each MPI process holds two buffers containing one element per vertex in the entire graph. Buffers A are the flag-buffers, while buffers B are the value-buffers. Additionally, to distinguish between buffer A on distinct MPI processes, the MPI process rank is appended to the buffer name such that A_0 is the buffer A held on MPI process 0 and A_1 is the buffer A held on MPI process 1.

As shown in Figure 7.3, the first phase consists of storing the messages generated by the vertices processed. For instance, in the example given, vertex 0 on MPI process 0 generates messages to vertices 1 and 5. Therefore, the corresponding elements in the A_0 flag-buffer are set to `true`. In practice no data-race protection is needed for concurrent writes attempting to store the same value regardless because the assumption is that such

writes in hardware are atomic. However, it should be noted that this is not legal OpenMP.

The value-buffers (B_0 and B_1) contain the actual message content to transmit. The default value of those buffer's elements is selected to be neutral to the combination operation specified by the user, zero here since the operation is a sum. As a consequence a compare-and-swap combination can be performed systematically, removing the need for data-race protections on these buffers too.

Out of all the buffers held by a process, the recipient is ultimately only interested in the final value of the elements corresponding to the vertices it processes. This is made visible by appending the buffer name to the rank of the destination MPI process. For instance, buffer $A_{0,0}$ is the subset of buffer A_0 for MPI process 0, while buffer $A_{0,1}$ is the one for MPI process 1. The buffer exchange phase, therefore, consists of combining the buffers so that each MPI process receives the final value of elements that reside in its subset. In this example, obtaining the combined results of flag-buffers $A_{0,0}$ and $A_{1,0}$, and that of value-buffers $B_{0,0}$ and $B_{1,0}$, on MPI process 0, and similarly for MPI process 1 with flag-buffers $A_{0,1}$ and $A_{1,1}$, and value-buffers $B_{0,1}$ and $B_{1,1}$. The resulting data is stored in buffers C and D respectively, which are the memory locations of a vertex's mailbox flags and values.

In MPI, this communication pattern can be achieved with two series of MPI reductions; a logical `or` reduction on the flags and then the user-defined combination operation operating upon the values. Each MPI process, in turn, becomes the root MPI process for both reductions, sending the interval of the flag and value buffers it holds. Non-blocking reductions are used to ensure that the MPI reductions that are issued are executed concurrently.

In this implementation, the structure externalisation technique leveraged in [3] was reused to extract both the vertex received message and the received message flag from the vertex structure. However, they have been grouped into two separate arrays so that all vertex mailbox flags are contiguous in memory, likewise for vertex mailbox values, while preserving the direct mapping property. This design, therefore, removes the need for a message dispatch phase as MPI reductions can store the results directly into the recipient vertex mailboxes via a single contiguous memory move.

7.3.3.2 Pull version

An example for the opposite approach, the pull version, is provided in Figure 7.4.

In the pull version, introduced in Subsubsection 4.3.4.3, communications consist of vertices fetching messages, if any, from the senders' outbox.

In the first phase, referred to as flag for broadcast, an element filled with true in buffer

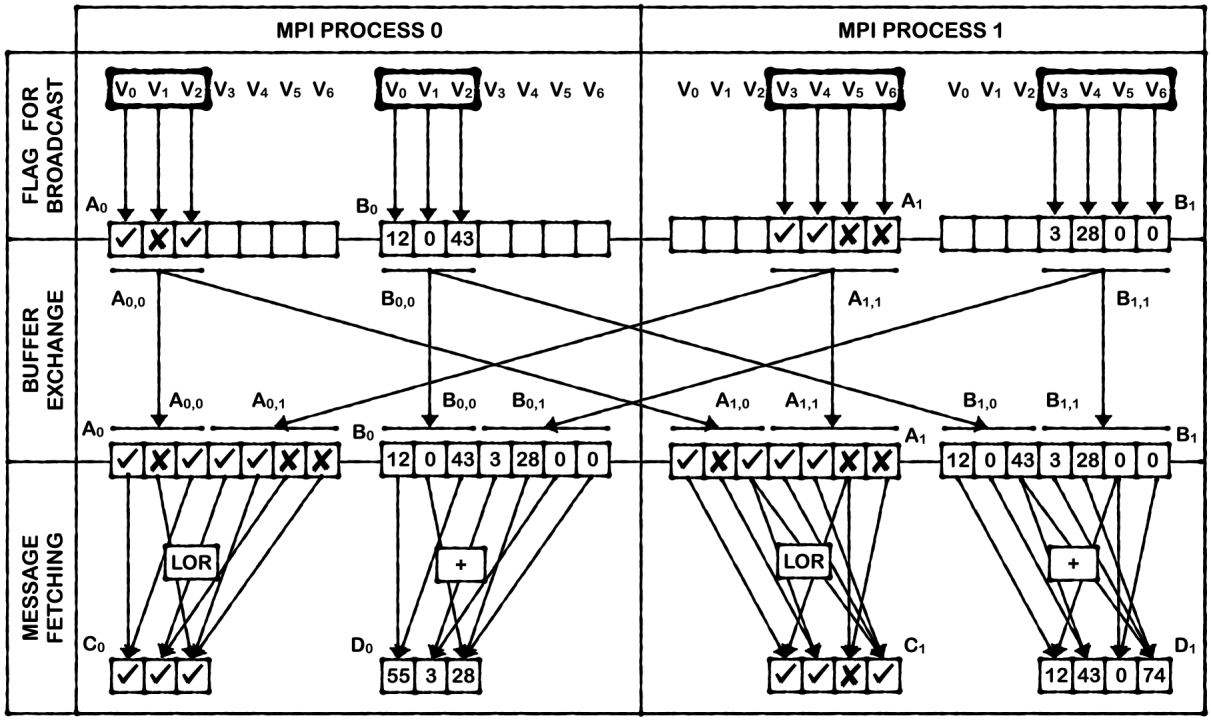

Figure 7.4: Workflow of the buffer exchange for the pull version in the *DiP* framework, assuming a sum combination operation.

A signifies that the corresponding vertex has a message to broadcast. For instance, in Figure 7.4 vertices 0 and 2 processed by MPI process 0 generate a message to broadcast. This contrasts with the push version, in which setting these two A_0 elements to true would have meant that vertices 0 and 2 required a message to be sent to them. As a consequence, each MPI process only fills its subset of the buffer during this phase.

During the second phase, where buffers are exchanged, each MPI process sends its subset of the buffer to all other MPI processes, which informs these other MPI processes about potential messages broadcast from the vertices it is processing. In MPI this can be achieved with two series of MPI broadcasts, one for the flag-buffer and one for the value-buffer (note that the same result can also be obtained using MPI_Gather). Since a subset of the buffer is sent to multiple MPI processes, the notation selected differs from that used by the push version. $A_{0,1}$ is the subset of buffer A held on MPI process 0 that is updated with data coming from MPI process 1.

The third and final phase occurs when each MPI process has gathered all messages broadcast across the entire graph. This phase consists of each MPI process iterating through each of their vertex neighbours and fetching the message broadcast, if any. Similar to the push version, flags are combined with a logical or and values combined using the user-defined combination operation (a sum in the example).

7.3.3.3 Benefits

The buffer design, which underlies both the push and pull approaches, presented in this subsection offer multiple advantages; for instance, it does not rely on a dynamically resized structure, which implies that any out-of-memory failure will arise during the memory allocations issued at the beginning of the application. This contrasts with other approaches where the memory usage may not exhaust memory resources until an arbitrary point in time in the execution, such as the last superstep for instance. This predictability in memory usage may prove to be valuable in long, and thus expensive, runs. The benefits of this buffer design are also observed in both sender and receiver distributed-memory workers.

Sender side By enhancing the element indexes to also encode vertex identifiers, the sender does not need to append the vertex identifier to messages. Instead, the index used to access any element in the flag and value buffers already represents the vertex identifier of the sender.

Furthermore, locating an element that corresponds to the recipient vertex in the flag or value buffer is straightforward due to the direct mapping characteristic of this buffer design. As recipient vertex mailbox information resides at the element whose index is equal to the recipient vertex identifier, this buffer design removes the need of using a hash function whose computational intensity is not negligible.

Direct mapping also allows messages generated by vertices to be combined immediately with any preexisting message for each recipient vertex identifier. This provides two benefits: firstly it removes the need for a sort later in the process, thus sparing the $O(n \times log_2(n))$ observed on other distributed-memory frameworks. Secondly this caps the maximum buffer size to be proportional to the number of vertices, not the number of edges.

On the sender side, the resulting algorithmic complexity of the message generation phase using the approach developed in this research is $O(n)$, which is reduced from the $O(n \times log_2(n))$ provided by distributed-memory vertex-centric frameworks.

Receiver side Benefits of this buffer design approach are also observed on the receiver side. Firstly, the buffers received do not require unwrapping to access the message content as the value buffer contains only message values. Secondly, the direct mapping characteristic removes the need for a message dispatch mechanism, along with intermediate copies. Paired with the vertex structure externalisation to place vertex mailboxes consecutively in memory, the flag and value buffers received can be stored directly in the final recipient

vertex's mailbox by executing a single memory move.

Finally, leveraging the underlying MPI technology to embed the combination into the collective operation provides the possibility of leveraging HPC optimisation techniques such as network offloading.

7.3.3.4 Limitations

The fundamental limitation of the buffer design developed in this thesis is the memory footprint of each MPI process when scaling. Equation 7.1 describes the amount of memory required per MPI process:

$$M_B = |V| * (S_F + S_M) \tag{7.1}$$

where M_B is the total amount of memory needed to store both flag and value buffers, $|V|$ is the graph order, S_F is the size of a flag and S_M is the size of the message payload. This equation illustrates that the memory cost associated with this buffer design is replicated across MPI processes, hindering both intra-node and inter-node MPI process scalability as well as increasing the amount of data exchanged by network communications.

7.3.4 64-bit collectives

In MPI, counts and displacements passed to routines are of type `int`, which is typically encoded using 32-bit signed integers. However, the size of buffers considered in this chapter results in counts and displacements that have values beyond what can be encoded within a 32-bit signed integer. Therefore, an extended version of the MPI routines used in this research was developed, where the count and displacement types are encoded using unsigned 64-bit integers. This change increases the maximum value encodable from $2^{31}-1$ to $2^{64}-1$, moving the upper limit from a couple of billion elements to trillions of trillions.

Nonetheless, the backend of this 64-bit library relies on the native 32-bit MPI routines. The technique employed, therefore, consists of iterating through the buffers and issuing multiple rounds of 32-bit MPI routines. For non-blocking routines, a new `MPI_Request` type is used to store all underlying 32-bit non-blocking requests obtained from the multiple rounds of native MPI non-blocking routines. Functions equivalent to `MPI_Wait` and `MPI_Waitall` were also developed to provide 64-bit counterparts of this synchronisation mechanism.

7.4 Optimisations

The original buffer design presented in this chapter provides multiple benefits, as described in Subsubection 7.3.3.3. However, the large memory footprint is a major disadvantage as this limits MPI process scaling, both intra-node and inter-node, and techniques presented in this section aim at mitigating.

7.4.1 Single MPI process

One potential solution would be to use a single MPI process per node, where multithreading covers the entirety of the node, including all NUMA regions. The frequent random accesses to buffers would therefore trigger cross-NUMA accesses, potentially resulting in a non-negligible overhead. By contrast, mapping one MPI process per NUMA region ensures that intra-MPI (i.e: OpenMP) memory accesses are kept within that NUMA region, effectively delaying the cross-NUMA interactions to the buffer exchange phase. Therefore the consideration is evaluating situations in which the importance of frequent OpenMP cross-NUMA accesses outweigh the overhead in exchanging MPI messages. The first factor to consider is the average degree, and this is because the more vertices that are present, the larger the buffers needed to be, and the more edges and more memory accesses. This is because a higher average degree increases the amount of data to be transferred by MPI whilst decreasing the number of memory accesses (thus potential cross-NUMA accesses) for OpenMP. Conversely, a lower average degree will favour an MPI process per NUMA region. This also depends on the nature of the memory accesses because, in the pull version, MPI processes write into the exchange buffer only at the indexes corresponding to the vertices it processes. Therefore, writes are issued to two disjoint sets of buffer entries. After the buffers are exchanged, however, the pull version needs to fetch messages, which results in memory accesses occurring over both NUMA regions. However, these are limited to read accesses, which might make a difference, for instance those systems involving NVRAM. By contrast, with the push version, before buffers are exchanged an MPI process can write to any location in that buffer, not allowing them to be split in a NUMA-friendly way for writes. Consequently this technique of a single MPI process is more likely to be beneficial for the pull version.

7.4.2 Intra-node buffer sharing (INBS)

As mentioned in Subsubsection 7.3.3.4, the memory footprint of the buffer design presented in this chapter is replicated on each MPI process. Therefore, on any node, the amount of memory needed to store the buffers is multiplied by the number of MPI processes resid-

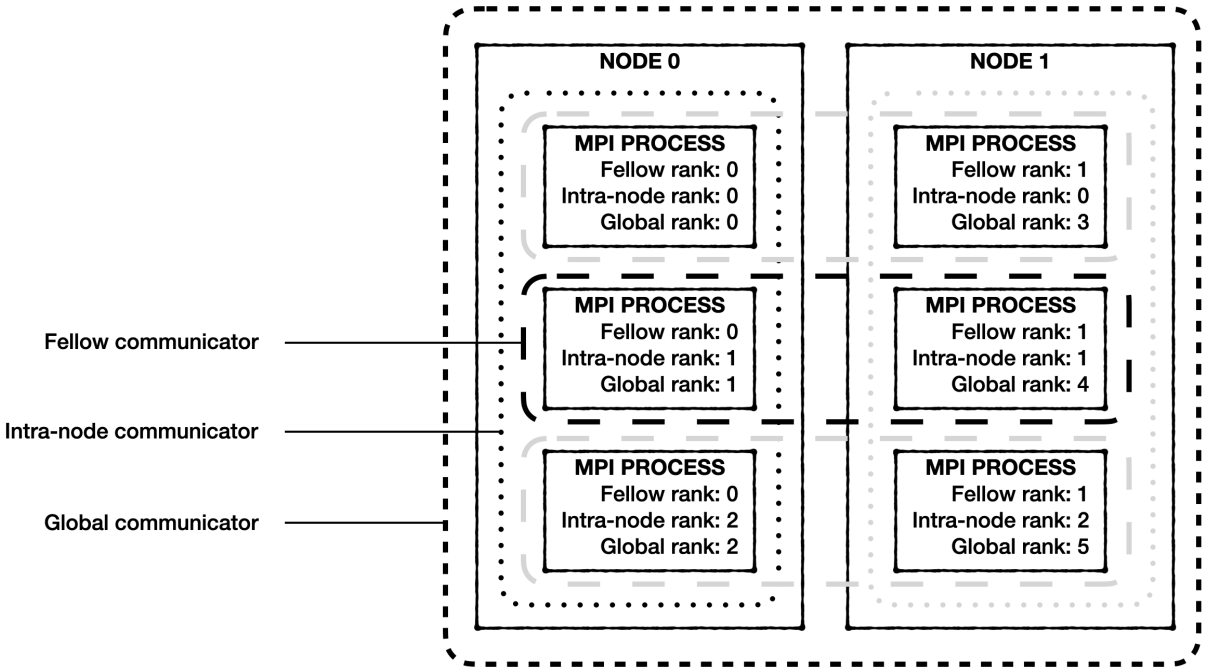

Figure 7.5: Communicator structure in the INBS *DiP* implementation.

ing on that node. This characteristic limits any intra-node process scalability, exhausting the node's memory beyond a certain point.

The approach presented in this subsection, called Intra-Node Buffer Sharing or INBS, consists of leveraging version 3 of the MPI standard, which introduced shared-memory programming where MPI processes expose and access a common memory segment. Consequently, MPI processes residing on the same node can share memory similarly to if these were running as threads. Leveraging this feature allows the flag and value buffers to be allocated only once per node regardless of the number of MPI processes that it hosts.

7.4.2.1 Communicators

As shown in Figure 7.5, in addition to the default global communicator represented with the constant `MPI_COMM_WORLD`, two types of communicators were created in the design of the INBS technique. The first one is the intra-node communicator which contains MPI processes residing on the same node as the caller. MPI version 3 allows the user to group such MPI processes by passing the flag `MPI_COMM_TYPE_SHARED` to the `MPI_Comm_split_type` routine. This communicator is necessary for MPI processes to expose and effectively share the flag and value buffer elements.

When considering the communication associated with the buffer exchange phase, a naive approach would consist of selecting one MPI process per node to handle the col-

lective operation on the entirety of buffers. However, this reverts the execution flow of collective operations to be sequential on each node. Such sequential execution could prove particularly penalising on the push version when processing the MPI reduction with a single MPI process on each node, aggravated further when facing high graph orders.

A potentially better approach consists of issuing a number of collective operations equal to that of the MPI processes. Each MPI process would, in turn, become the root of the reduction operation on the buffer subset it is responsible for. This approach preserves the parallel MPI process execution on each node by issuing P_t collective operations involving P_t MPI processes each, where P_t is the total number of MPI processes. However, given that MPI processes on a given node share the flag and value buffers, any collective operation would involve concurrent and redundant read / write from MPI processes to the same buffer subset, as well as requiring cross-MPI process aliased pointers in send-receive buffers.

The solution proposed by this research is illustrated in Figure 7.6, which consists of addressing the aliased accesses issue by restricting all collective operations to one MPI process per node. In MPI, collective operations must be called by all MPI processes in the corresponding communicator. Applying any filter to MPI processes participating in a collective operation, therefore, means filtering the MPI processes belonging to the corresponding communicator. To that end, communicators referred to as *fellow communicators* have been created. These contain MPI processes whose node-local ranks are identical, which includes exactly one MPI process per node. This solution, therefore, ensures that each buffer subset will be interacted with by a single MPI process on each node. In addition to overcoming the aliased accesses issue, this solution also decreases the number of MPI processes participating in each collective operation from P_t to N, where P_t is the total number of MPI processes and N is the number of nodes.

7.4.2.2 Push version

Algorithm 4 depicts pseudo-code for the push version of the INBS technique in the *DiP* framework. The algorithm ensures that, in each node, MPI processes applying a reduction on disjoint subsets of the shared buffer may do so concurrently. This was made possible by grouping MPI processes into fellow communicators where each communicator contains all MPI processes responsible for a given shared buffer subset and, as shown in Figure 7.5, this corresponds to one MPI process per node. Attention must be paid to synchronisation, as MPI processes may write to buffer elements virtually held by any MPI process on that node (as shown in Figure 7.3). Therefore an intra-node synchronisation must occur before buffer exchange may proceed. This synchronisation will ensure that there are no

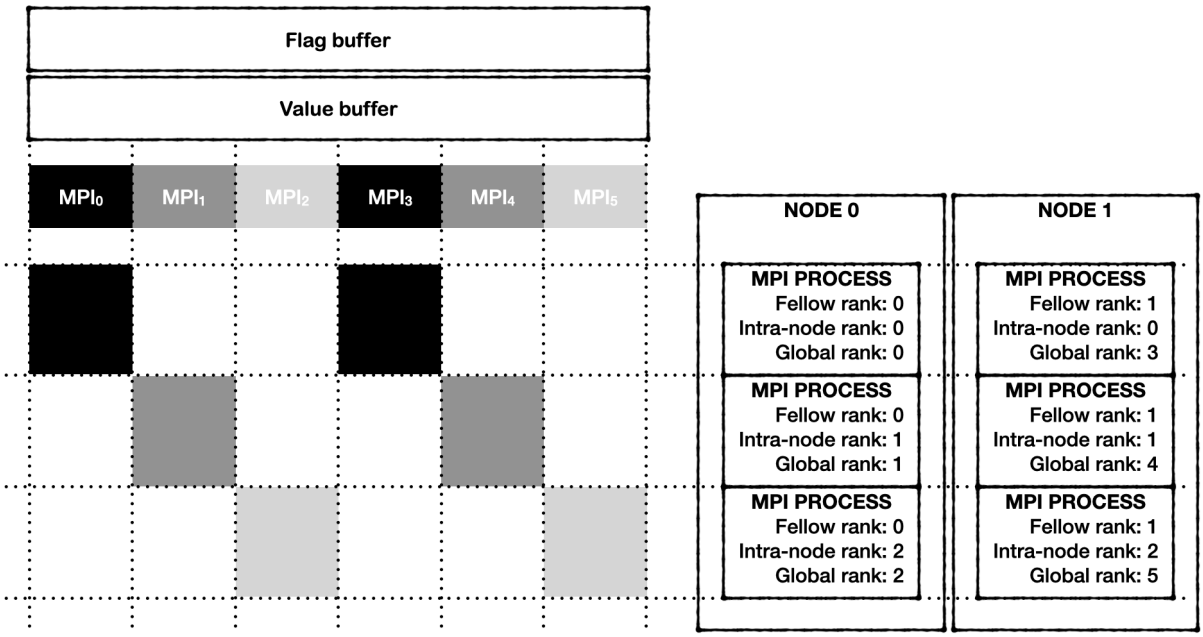

Figure 7.6: Decomposition of the buffer exchange phase into concurrent series of MPI collective operations in the INBS implementation

Algorithm 4: Pseudo-code for buffer exchange using the INBS technique with the push version of *DiP*.

```
1  begin
2      MPI_Barrier(intranode_communicator)
3      i ← 0
4      while i < fellow_communicator_size do
5          root_global_rank ← intranode_rank + i × intranode_communicator_size
6          offset ← id of first vertex handled by root_global_rank
7          offset_next ← id of first vertex handled by root_global_rank + 1
8          count ← offset_next - offset
9          MPI_Ireduce(flag-buffer[offset ; offset + count - 1])
10         MPI_Ireduce(value-buffer[offset ; offset + count - 1])
11         i ← i + 1
12     Wait on all non-blocking reductions issued
```

pending writes on buffer elements that may be in transit over the network. Similarly, an additional node synchronisation must be placed after the mailbox reset that follows, where flag-buffer elements are reinitialised to `false` and value-buffer elements to the combination neutral value. This ensures that the reset of a certain portion of the buffers will not occur concurrently to the message pushes from another MPI process which, for instance, would have already started the subsequent superstep.

7.4.2.3 Pull version

Algorithm 5: Pseudo-code for the buffer exchange using the INBS technique with the pull version of *DiP*.

```
1  begin
2      i ← 0
3      while i < fellow_communicator_size do
4          root_global_rank ← intranode_rank + i × intranode_communicator_size
5          offset ← id of first vertex handled by root_global_rank
6          offset_next ← id of first vertex handled by root_global_rank + 1
7          count ← offset_next - offset
8          MPI_Ibcast(flag-buffer[offset ; offset + count - 1]
9          MPI_Ibcast(value-buffer[offset ; offset + count - 1]
10         i ← i + 1
11     Wait on all non-blocking broadcasts issued
12     MPI_Barrier(intranode_communicator)
```

The equivalent of Algorithm 4 for the pull version is illustrated in Algorithm 5. By contrast to the push version of the INBS technique, synchronisation is required after the data exchange, not before. The reason for this is that, as shown in Figure 7.4, each MPI process may fetch messages from vertices held on any other MPI process residing on that node. This synchronisation, therefore, ensures that local buffer accesses have been completed before the message fetching phase begins. Once again, conversely to the push version, synchronisation protecting the mailbox reset must be done before, and not after, the reset. The reason for this is the access pattern, where an MPI process may reset buffer elements that have pending read access(es) from other MPI process on the node which are fetching messages from neighbours of their vertices.

7.4.2.4 Advantages

The core benefit of the INBS technique is the decorrelation of node memory footprint and MPI process count, which results in the new memory footprint calculation described

in Equation 7.2, assuming an identical number of MPI processes per node.

$$M_B = \frac{|V| * (S_F + S_M)}{P_N} \tag{7.2}$$

where M_B is the total amount of memory needed per node to store both flag and value buffers, $|V|$ is the graph order, S_F is the size of a flag, S_M is the size of the message payload and P_N is the number of MPI processes per node. Because the number of MPI processes per node is now a term in this equation, by contrast to Equation 7.1, memory footprint redundancy has been decreased from a per MPI process level down to a per node level.

In addition, the INBS technique improves network bandwidth usage. For the push version, for instance, as illustrated in Figure 7.3, without the INBS technique each MPI process sends over the network the entirety of its flag and value buffers. As a result, the amount of data sent over the network is the size of these buffers multiplied by the total number of MPI processes. When using the MPI 3 shared-memory feature, however, this amount is reduced to the size of the buffer multiplied by the number of nodes, effectively dividing the amount of data that must be sent by the number of MPI processes per node. Similarly, for the pull version, having fewer MPI processes participating in the MPI broadcast operation reduces the number of network copies.

Finally, this technique makes it possible to experiment with a mapping of MPI processes to (physical or logical) cores with a 1:1 ratio, which would likely result in an out-of-memory failure when using the naive implementation for graphs of any complexity. For the push version a higher number of MPI processes participating in the reduction also increases the degree of parallelisation of the MPI reduction. The higher the graph order, thus the larger the buffers in *DiP*, the bigger the performance gain to be expected by the MPI reduction parallelisation.

7.4.3 Interval-based message processing (IBMP)

Although the intra-node buffer sharing technique presented in Subsection 7.4.2 reduces the memory footprint replication factor from per MPI process to per node, this may not be sufficient to prevent memory exhaustion when processing very large graphs. The interval-based message processing technique presented in this subsection, shortened IBMP, aims at providing the ability to arbitrarily determine the maximum amount of memory that can be used by *DiP* buffers, thus virtually removing the possibility of an out-of-memory failure due to memory replication.

The overarching objective of the IBMP technique aims to provide the *DiP* framework

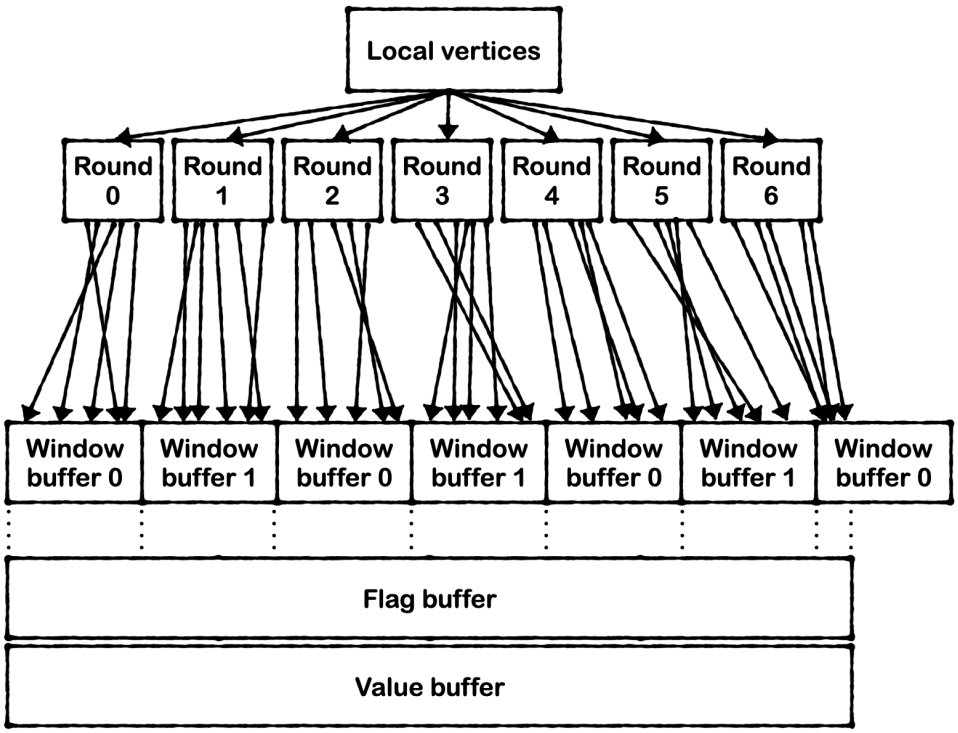

Figure 7.7: Execution flow of the interval-based message processing technique for the push version of the *DiP* framework.

with the ability to store only a portion of the entire flag and value buffers by using an interval that determines which subset of the buffers are to be interacted with at a given time. For the push version, this results in pushing a message that has been generated during the vertex processing phase, whereas for the pull version it involves reading a message during the message fetching phase. In both cases, the workflow can be redesigned so that the entirety of the flag and value buffer elements can be processed in multiple rounds, through the definition of an interval specifying the boundaries of the buffer subset to be processed at each round.

7.4.3.1 Push version

In the push version of *DiP*, generated messages are stored (and combined) into the recipient's slot in flag and value buffers. As shown in Figure 7.7, the flag and value buffers are processed in rounds. Each round covers an interval, with buffers known as *window buffers* acting as a proxy for the global flag and value buffers. Only messages intended for recipients inside the interval covered during the current round are sent, and other messages are discarded for the round. Eventually, the entirety of flag and value buffer elements will have been part of an IBMP interval exactly once, as described by the pseudo-code provided in Algorithm 6. However, processing each vertex once per round, thus several

Algorithm 6: Pseudo-code of the IBMP technique applied during one superstep in the push version of *DiP*.

```
1  begin
2      Round_Count ← ⌈ V / WindowBufferSize ⌉
3      Round_Index ← 0
4      while Round_Index < Round_Count do
5          V_First ← Round_Index × WindowBufferSize
6          V_Last ← V_First + WindowBufferSize - 1
7          if Round_Index is even then
8              WB_Current ← WB_0
9              WB_Previous ← WB_1
10         else
11             WB_Current ← WB_1
12             WB_Previous ← WB_0
13         if Round_Index ≥ 2 then
14             MPI_Wait64(WB_Current)
15         foreach v in LocalVertices do
16             if Round_Index < Round_Count - 1 then
17                 BackupVertex(v)
               // Ignore messages to recipients outside [V_First;
                  V_Last]
18             ProcessVertex(v)
19             if Round_Index < Round_Count - 1 then
20                 RestoreVertexBackup(v)
21         MPI_Ireduce64(WB_Current)
22         if Round_Index < Round_Count - 1 then
23             ResetLocalVertices()
24         Round_Index ← Round_Index + 1
25     if Round_Count > 1 then
26         MPI_Wait64(WB_Previous)
27     MPI_Wait64(WB_Current)
28     ActivateVerticesWithMessages()
29     CountActiveVertices()
```

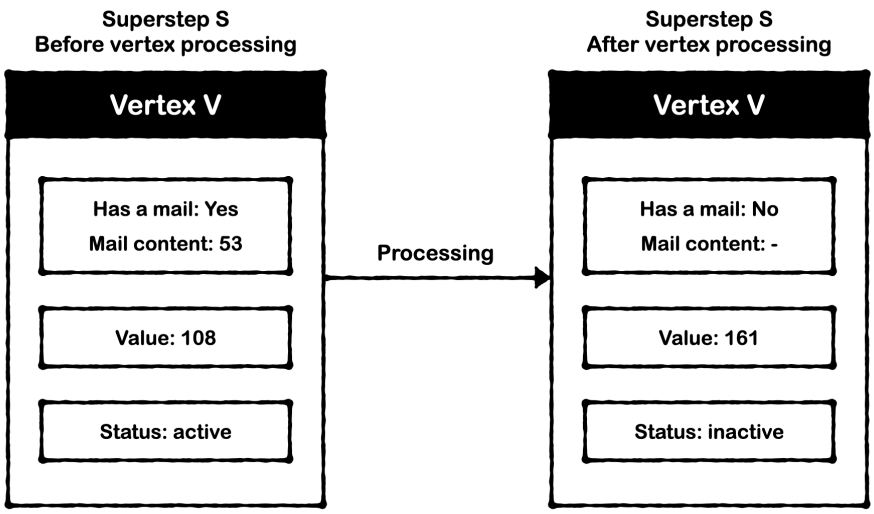

Figure 7.8: Example of state evolution of a vertex throughout execution, assuming a sum combination operation.

times per superstep, raises new challenges.

Reproducibility As shown in Figure 7.8, after the first round the state of a vertex may have been updated; for instance, it has become inactive, changed value or consumed its messages. Although this updated state is correct as the starting state for the following superstep, it is incorrect for the following IBMP round in the current superstep. Indeed, at the end of the first IBMP round in superstep S, elements of the flag and value buffers corresponding to the neighbours of vertex V have been updated with the messages generated by vertex V if they belong to the interval covered during this round. Meanwhile, messages destined for neighbours outside this interval have been discarded for this round. As mentioned earlier, every element in the flag and value buffers will be covered eventually as IBMP rounds progress. However, when the second IBMP round of superstep S begins, vertex V is processed again and this time only messages generated in the new interval are preserved. The flaw here is that at the end of the first IBMP round vertex V is inactive, and therefore at the start of the second IBMP round it is skipped. As a consequence, messages destined for any neighbour residing in the new interval are not generated. Moreover, the fact that the new state of vertex V indicates it has no new message may trigger a different response, where the underlying vertex-centric application may for instance generate messages only upon receipt of new messages. Furthermore, in vertex-centric programs, vertices typically send their current value as the message payload. That information too has changed between the first and second IBMP rounds, from 108 to 161 in the example given in Figure 7.8.

This example of Figure 7.8 highlights a non-reproducibility issue, where a mechanism must be implemented so that vertices can be *replayed* in a reproducible manner. The solution adopted here is the creation of a *vertex backup*: a copy of the vertex's original state before the first IBMP round. As shown in Figure 7.8, four pieces of information must be stored, these are the vertex's mailbox flag, mailbox content, status and current value. In every IBMP round except the last, the state of the vertex process is reset after processing by reloading this original, thus addressing the non-reproducibility issue.

Memory footprint The memory footprint of the IBMP technique comprises four components, the flag window buffers, the value window buffers, the vertices' original values and the *next* buffers. The amount of memory generated by the IBMP technique is given in Equation 7.3.

$$M_{IBMP} = 2 \times I \times (S_F + S_M) + V_L \times (S_B + S_N) \tag{7.3}$$

where M_{IBMP} is the total amount of memory required per MPI process to progress by intervals containing I vertices, S_F is the size of a flag, S_M is the size of the message payload, V_L is the number of vertices locally held, S_B is the size of a vertex backup and S_N is the size of the *next* buffers.

As mentioned earlier, the store of a vertex's original values contains four fields; the mailbox flag and the status fields, which are both booleans, and the mailbox content and current value which are of user-defined type `DIP_MESSAGE_TYPE`. The size of a vertex backup, therefore, is given by Equation 7.4.

$$S_B = 2 \times (S_F + S_M) \tag{7.4}$$

Buffers referred to as *next* buffers are part of all synchronous vertex-centric frameworks. These buffers act as mailboxes for messages received during the current superstep, and therefore intended to be accessed in the subsequent superstep. *Next* buffers are key to ensuring correctness as they prevent messages from different supersteps being mixed. The memory footprint of a next buffer is given by Equation 7.5.

$$S_N = S_F + S_M \tag{7.5}$$

where V_L is the number of vertices handled by the MPI process. When substituting S_B and S_N in Equation 7.3 with equations 7.4 and 7.5 respectively, Equation 7.6 is obtained.

$$M_{IBMP} = (3 \times V_L + 2 \times I) \times (S_F + S_M) \tag{7.6}$$

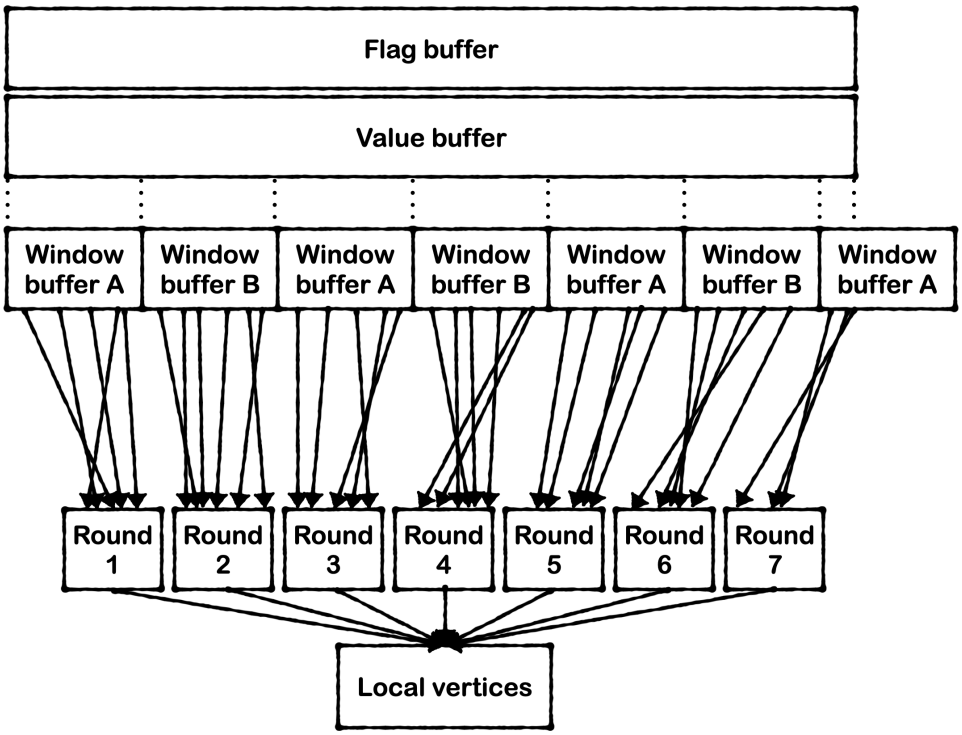

Figure 7.9: Execution flow of the interval-based message processing technique for the pull version of the *DiP* framework.

7.4.3.2 Pull version

Figure 7.9 illustrates how the IBMP technique is performed in the pull version of the *DiP* framework. It relies on the same approach as the push version, where the flag and value buffers are iterated through with the use of window buffers covering a given interval at a time. With the pull version, however, it is the message fetching phase that is repeated once per round, fetching messages from neighbours residing in the interval currently being covered.

Associativity As shown in Figure 7.4, after the buffer exchange phase, vertices fetch messages from neighbours residing at any index in the flag and value buffers. When applying the IBMP technique all vertices will fetch messages only from neighbours in the interval being covered in the current round. Therefore this fetch operation must be repeated on all vertices every IBMP round, similarly to the message generation phase in the push version.

Consequently, any potential non-reproducibility must be protected with the storing of original vertex state when needed. The operation applied on vertices during the message fetching phase updates the processed vertex's state twice. Firstly it updates the vertex's flag which indicates whether this vertex connects to a neighbour which broadcasts a

message, and this is described by Equation 7.7.

$$F_B(v) = \bigvee_{i \in \Gamma(v)} B_i \tag{7.7}$$

where $F_B(v)$ is the final state of the flag member of the vertex v being processed, $\vee$ is the logical `or` operation, B_i is the element at index i in the flag buffer and $\Gamma(v)$ is the list of incoming neighbour vertex identifiers. The second update consists of combining the messages obtained from broadcast messages if any, as described by Equation 7.8.

$$F_R(v) = \bigoplus_{i \in \Gamma(v)} R_i \tag{7.8}$$

where $F_R(v)$ is the final state of the message of the vertex v being processed, $\bigoplus$ is the combination operation specified by the user, R_i is the element at index i in the value buffer and $\Gamma(v)$ is the list of incoming neighbour vertex identifiers. As mentioned in the preconditions of the *DiP* function `dip_combine` in Figure 7.1, the provided combination operation must be associative. Therefore, both the logical or operation, $\vee$, applied in Equation 7.7 and the combination operation, $\bigoplus$, applied in Equation 7.8 are associative. This implies that the final state of a vertex will be identical regardless of whether it has been calculated on the entire flag and value buffers at once or through multiple IBMP rounds. It follows that no reproducibility issue arises in the pull version, which therefore removes the need for storing the original vertex state. The pseudo-code to implement the IBMP technique in the pull version of *DiP* is given in Algorithm 7.

Memory footprint Because the pull version does not require storing the original vertex state, this avoids the memory cost associated with their storage. The resulting memory footprint of the IBMP technique in the pull version, therefore, can be calculated by subtracting S_B in Equation 7.4 from Equation 7.3, which gives Equation 7.9.

$$M_{IBMP} = (V_L + 2 \times I) \times (S_F + S_M) \tag{7.9}$$

where M_{IBMP} is the total amount of memory needed by the IBMP buffers to progress with intervals made up of I vertices, V_L is the number of vertices locally held, S_F is the size of a flag and S_M is the size of the message payload.

7.4.3.3 Interleaved window buffer usage

To minimise the impact of the IBMP technique on performance, two window buffers are used alternatively, corresponding to lines 7-12 in Algorithms 6 and 7. The use of MPI

Algorithm 7: Pseudo-code for IBMP in the pull version of *DiP*.

```
1  begin
2      Round_Count ← ⌈V / WindowBufferSize⌉
3      Round_Index ← 0
4      while Round_Index < Round_Count do
5          V_first ← Round_Index × WindowBufferSize
6          V_last ← V_First + WindowBufferSize − 1
7          if Round_Index is even then
8              WB_Current ← WB_0
9              WB_Previous ← WB_1
10         else
11             WB_Current ← WB_1
12             WB_Previous ← WB_0
13         if Round_Index ≥ 2 then
14             MPI_Wait64(WB_Current)
15         FetchMessagesAndCombine(WB_Current)
16         if Round_Index = MyMpiRank then
17             MPI_Ibroadcast64(LocalVerticesOutbox[V_first; V_last])
18         else
19             MPI_Ibroadcast64(WB_Current)
20         Round_Index ← Round_Index + 1
21     if Round_Count > 1 then
22         MPI_Wait64(WB_Previous)
23         FetchMessagesAndCombine(WB_Previous)
24     MPI_Wait64(WB_Current)
25     FetchMessagesAndCombine(WB_Current)
26     ActivateVerticesWithMessages()
27     CountActiveVertices()
```

non-blocking operations in the buffer exchange phase enables the exchange of one window buffer to be concurrent with the processing of the other. This systematic switch between window buffers is repeated until all IBMP rounds are processed for the current superstep.

However, this interleaved pattern requires careful synchronisations because, for example, from the third superstep onwards, the buffer window to be used is already part of a buffer exchange issued two rounds prior. The *DiP* framework must therefore explicitly wait for completion of the corresponding buffer exchange as shown in line 14 of Algorithms 6 and 7. Lines 26-27 of Algorithm 6 and lines 22 and 24 of Algorithm 7 highlight the need to immediately synchronise both window buffers upon exiting the loop. The reason for this is that, inside the loop, waiting upon the completion of a buffers exchange is performed two loop iterations later (that is, two IBMP rounds later). When the last IBMP round finishes, the non-blocking operations corresponding to the buffer exchanges from current and previous IBMP rounds are still pending, and therefore must be waited upon for completion before proceeding with the rest of the algorithm.

As a result, by alternatively using the first and second window buffers, this approach enables an overlapping of computation and communication, minimising the impact of the IBMP technique on performance.

7.4.3.4 Advantages

The IBMP technique presented in this subsection enables the *DiP* framework to control its memory footprint and conform to a limit specified by the user. As a consequence, this technique mitigates the fundamental weakness of the buffer design presented in Section 7.3.3, enabling the *DiP* framework to process graphs which would otherwise have resulted in *DiP* buffers triggering an out-of-memory failure.

7.4.3.5 Limitations

The IBMP technique relies on a trade-off between memory footprint and performance, where the gain in the former comes at the cost of an overhead in the latter, proportional to the number of rounds required to cover all vertices.

7.4.3.6 Conclusion

The IBMP technique presented in this subsection allows the user to arbitrarily determine the maximum amount of memory that will be used by *DiP* for its buffers. This gain in control and reduction of memory footprint comes at the expense of additional computation due to the multiple rounds that must now be executed per superstep. However, the

primary objective of IBMP is to enable *DiP* to process graphs whose resulting buffers would otherwise have resulted in an out-of-memory failure.

Nevertheless, by adapting to an arbitrarily defined memory footprint limit, the IBMP technique mitigates the fundamental weakness of the buffer design presented in this chapter. More generally, it also offers the graph processing community a new approach to better control the memory footprint of distributed-memory vertex-centric frameworks.

7.5 Experiments

This section describes the conditions and configurations in which the experiments presented in this chapter were conducted.

7.5.1 Computing environment

Experiments are run on standard compute nodes of the Cirrus, whose technical specifications are provided in Subsection 4.6.1.2.

Compilation is undertaken using the *gcc* compiler version 8.2.0 (OpenMP version 4.5) and OpenMPI version 4.1.0. Compilation flags passed enable the support for C11 standard (`-std=c11`) and level 2 optimisations (`-O2`).

The results presented in this chapter are calculated based on the average of three runs.

7.5.2 Search space

Figure 7.10 illustrates the search space of parameters that can be set in the experiments. The overall number of possible permutations of these is very large, and as such not all combinations can be investigated. Consequently, as detailed in this section, specific decisions have been made around which parameters to investigate and which to fix.

Graph size Experiments conducted in this investigation aim to evaluate the competitiveness of the *DiP* framework compared to its shared-memory counterpart. It follows that graphs considered must be processable by both frameworks, implying that graphs must fit in the 256GB of memory available on a single node.

Graph imbalance The research presented in this chapter focusses on the buffer design introduced in Section 7.3.3. While load-balancing challenges in vertex-centric programming have been thoroughly investigated in [3], they are outside the scope of this investigation. Therefore, the standard OpenMP `dynamic` loop clause was used, with a chunk

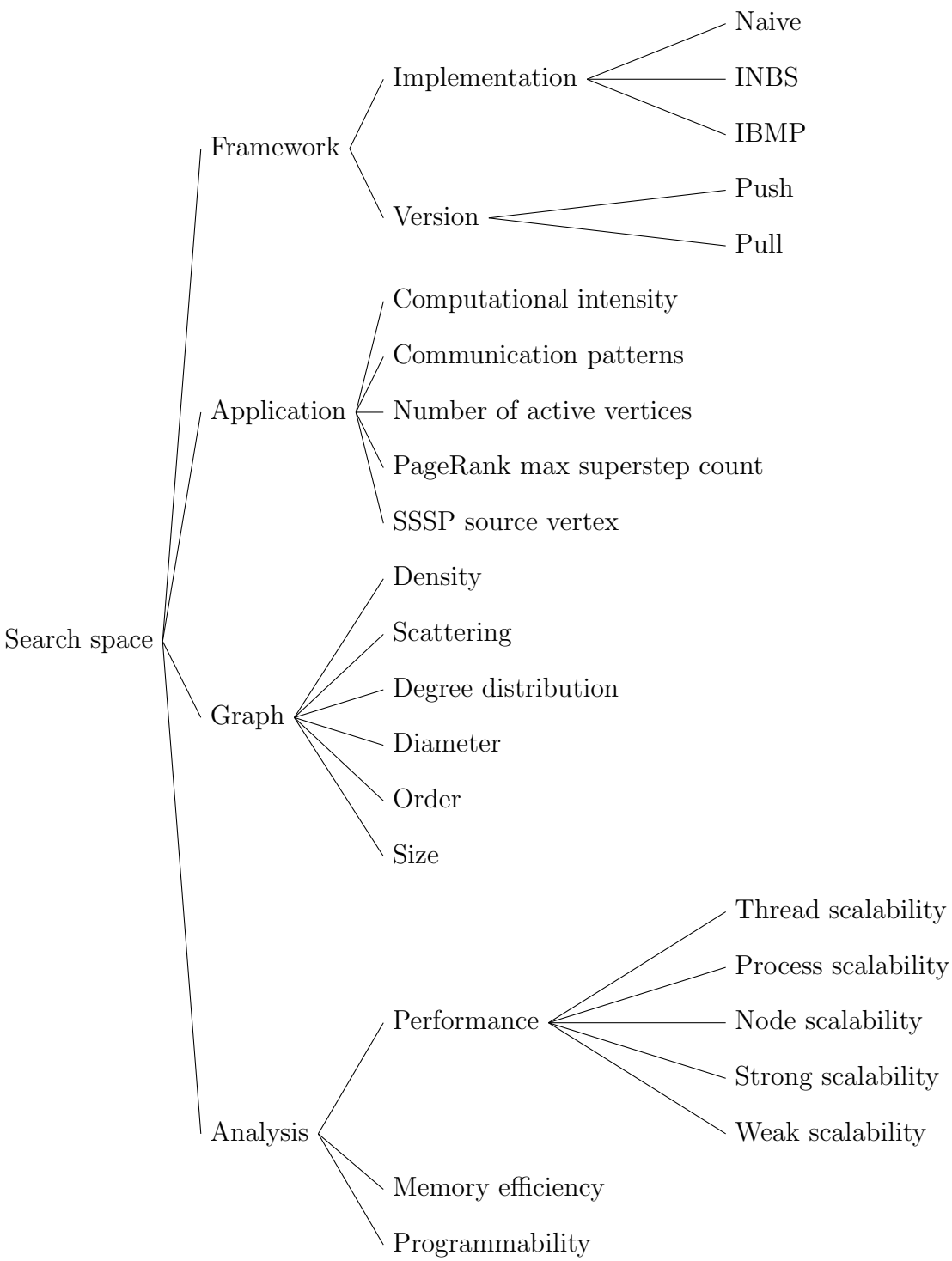

Figure 7.10: Parameter search space in *DiP* experiments.

Table 7.1: Number of vertices and edges in the graphs selected for experiments

Name	Graph order	Graph size	Average degree
S_1.5B_16_100K	1,500,000,000	24,000,000,000	16
S_50M_1K_10K	50,000,000	50,000,000,000	1,000

size empirically determined to:

$$\frac{|V|}{10 \times O_P \times P_N \times N} \tag{7.10}$$

where $|V|$ is the number of vertices in the entire graph, O_P is the number of OpenMP threads per MPI process, P_N is the number of MPI processes per node and N is the number of nodes.

Graph locality The impact of locality on vertex-centric application performance was analysed in [4] and reported in Chapter 6. However, the priority of the experiments conducted in this chapter is the evaluation of the buffer design as this is the core component of *DiP* and major research contribution presented in this chapter.

7.5.3 Graphs

Graphs at the scale considered in this investigation require disk storage, typically on the order of terabytes, beyond what can be reasonably expected from common cluster allocations. Consequently graphs are instead generated in memory, aiming at average degrees observed in existing literature and real-world data, from approximately 16 [5, 13, 24, 25, 58, 59, 91] up to 1,000 and above [24, 68]. To prevent unrealistic neighbour locality a parameter referred to as *scattering distance* is applied which determines the distance between two consecutive neighbours of a given vertex. The graphs obtained, whose configurations are reported in Table 7.1, have been produced using the graph generator introduced in Subsubsection 6.4.2 and are named using the convention A_B_C, where A is the graph order, B is the average degree and C is the scattering distance.

7.5.4 Benchmarks

The experiments conducted in this chapter include the three benchmarks: PageRank, Connected Components and Single-Source Shortest Paths, whose implementations are given in Appendices I, J and K respectively. As explained in Subsubsection 2.3.2, these

benchmarks expose a variety of algorithmic characteristics, including the variation of the number of active vertices throughout supersteps, as well as inherent properties allowing or preventing the use of certain optimisation techniques. For reasons that will be explored later in this chapter, they also represent good, average and bad case scenarios for the *DiP* framework. It should be noted that PageRank experiments run over 10 supersteps, and that the vertex identified by 0 is selected as the source vertex in SSSP experiments.

7.5.5 Frameworks

The experiments conducted in this chapter do not compare against frameworks beyond *iPregel* given that the focus of this research is the optimisation of a specific phase in vertex-centric programs. Comparing against other frameworks would likely involve runtimes with diverse sources of overheads, thus not reflecting impacts made on the buffer exchange performance in particular. In addition, the techniques developed in this section cover more than a single profiling area because they explore the network communication phase, sender-side message combination, client-side combination and client-side message dispatch.

7.6 Results

This section presents and provides an analysis of the data gathered from experiments.

7.6.1 Single node performance

The scope of the first set of experiments to be conducted was restricted to single-node runs. By doing so the competitiveness of the distributed-memory framework *DiP* against its shared-memory counterpart *iPregel* can be evaluated, as a first step before investigating its scalability in subsequent sections. Accordingly, the graphs selected are of a size permitting to be processed within the amount of memory held on a single node.

7.6.1.1 Sparse graph

All three benchmarks were run on the sparse graph, S_1.5B_16_100K, which contains 1.5 billion vertices and 24 billion edges. As mentioned in Subsection 7.5.3, the resulting average degree of 16 is observed in numerous synthetic and real-world graphs used in existing literature.

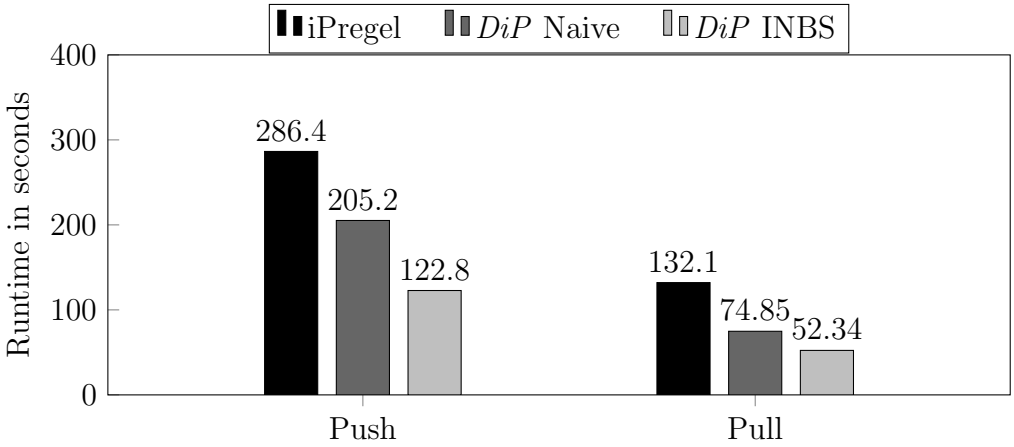

Figure 7.11: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute 10 PageRank iterations on S_1.5B_16_100K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

PageRank The results collected from PageRank are reported in Figure 7.11. The data shows that on the sparse graph S_1.5B_16_100K, the naive *DiP* implementation outperforms *iPregel* on both push and pull scenarios. The push version of the naive *DiP* implementation, for instance, is 205.19 seconds which is 28% lower than the runtime of the shared-memory *iPregel* which completes the 10 PageRank iterations in 286.36 seconds. This performance difference increases to 43% for the pull version.

Although the software of *iPregel* and *DiP* are inherently similar due to the latter being the distributed-memory port of the former, their different parallelism strategies impose different restrictions and provide different levels of freedom. One such difference resides in the combination mechanism where, upon sending a message in *iPregel*, the hybrid combiner introduced in Section 5.3 systematically checks whether the recipient's mailbox is empty. This determines whether the message to be pushed should be written to the memory location or combined with an existing message. On the other hand, in *DiP*, mailboxes are initialised to a neutral value, allowing the framework to use Compare-And-Swap (CAS) directly. The hybrid combiner still requires the flag check, and potentially a lock-based combination. Furthermore, vertex structures in *DiP*, therefore, do not carry over the lock, increasing the probability of cache hits.

Another factor is the handling of cross-NUMA interactions, where it appears that the distributed-memory framework outperforms *iPregel* by up to 20%. This may be explained through the different mapping of workers to hardware: whereas *iPregel* evenly spreads its 32 threads across both NUMA regions, *DiP* pins 1 MPI process per NUMA

region, each spawning 16 threads bound to that NUMA region. It follows therefore that *iPregel* threads may issue cross-NUMA memory accesses whereas those in *DiP* are solely intra-NUMA. Cross-NUMA communications in *DiP* take place at the MPI level, through an MPI message exchange. Therefore, it comes down to comparing cross-NUMA memory accesses at the thread-level against communications at the process-level, and the results reported in Figure 7.11 demonstrate that the later approach favours performance. This observation could be investigated further in *iPregel*, by developing a NUMA-aware distribution of data across threads in order to minimise cross-NUMA memory accesses.

The performance gain increases further when enabling the intra-node buffer-sharing (INBP) implementation of *DiP*. With an additional 82 seconds reduced from the runtime, the push version of the INBS *DiP* completes the 10 PageRank iterations in approximately 120 seconds, effectively becoming 2.3 times faster than *iPregel*. Similarly, on the pull version, the decrease of 22 seconds in the runtime when moving from the naive *DiP* implementation to INBS allows the total runtime saving to reach 80 seconds compared to *iPregel*. This makes the INBS implementation of *DiP* approximately 2.5 times faster than *iPregel*.

Profiling has helped explain the performance difference because, whilst the same time is spent on vertex processing, the buffer exchange dominates the difference in runtime observed. As per the *DiP* buffer design introduced in Section 7.3.3, a graph with 1.5 billion vertices, such as the S_1.5B_16_100K graph, requires the *DiP* buffers to contain 1.5 billion elements each. In the case of PageRank, the memory footprint of these buffers reaches 13.5GB per MPI process on the naive *DiP* implementation, and 13.5GB per node on the INBS *DiP* implementation. Also, the INBS technique permits *DiP* to halve the memory footprint of its buffers, from 27GB to 13.5GB as well as entirely bypassing the communication phase in the context of single-node processing due to MPI 3 shared-memory. In the push version, in addition to halving the amount of memory passed over the network for reduction, the INBS implementation allows the MPI_Reduce to be parallelised further by distributing it across 32 MPI processes per node instead of 2.

When comparing push against pull versions, the latter proves to be consistently faster. Overall, the speedups obtained range from 2.2 up to 2.7. This difference observed is due to the buffer exchange no longer being performed using an MPI reduction but instead a broadcast, which avoids the need to apply an operation to the value; as well as write memory accesses being issued only to locally held memory in the pull version.

Connected Components Figure 7.12 plots the runtime of processing the first 20 supersteps of Connected Components. A fixed number of supersteps was selected to limit the overall machine time required for this experiment, as the total number of supersteps

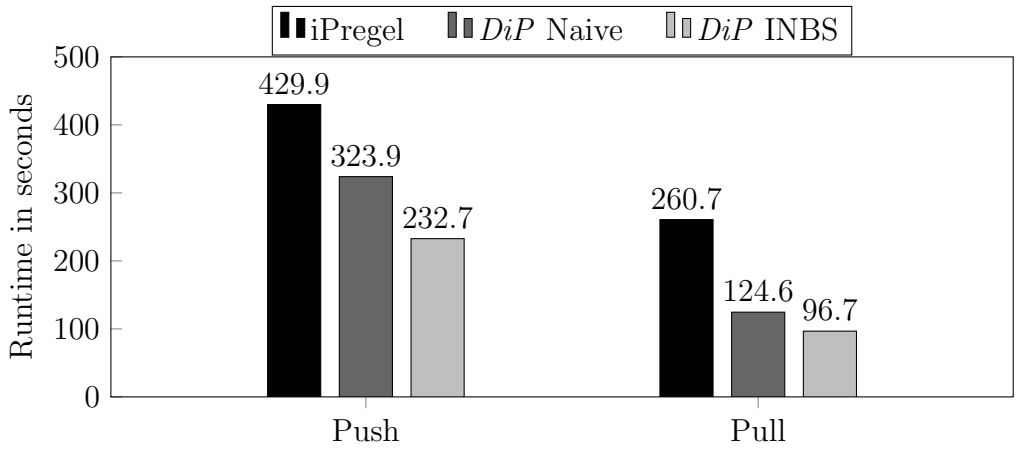

Figure 7.12: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute the first 20 Connected Components iterations on S_1.5B_16_100K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

that are required to run the algorithms which are message-propagation based, such as Connected Components, to completion is equal to the graph diameter. Using the graph generator introduced in Chapter 6, the formula to calculate the diameter of the graphs generated is given in Equation 7.11.

$$D = L + \left\lfloor \frac{|V|}{L} \right\rfloor \tag{7.11}$$

where D is the graph diameter, L is the scattering distance and $|V|$ is the graph order. Therefore, when considering graph S_1.5B_16_100K, the total number of supersteps required is 110,000, which would require an excessive amount of execution time.

Results collected show that the naive *DiP* implementation is faster than its shared-memory counterpart, by a factor of 1.3 for the push version and almost 2 times for the pull version. In addition, the INBS implementation provides further gains, with a speedup against the naive *DiP* implementation of a factor of approximately 1.3 for both versions.

This data is comparable to that obtained from the PageRank experiment. This is partially due to the sparse nature of the S_1.5B_16_100K graph which, after the first 20 supersteps, results in more than 99.99% of all graph vertices still being active. Combined with the fact that both benchmarks have the same communication pattern and computational intensity, from a workload perspective, the processing of Connected Components is similar to the processing of PageRank.

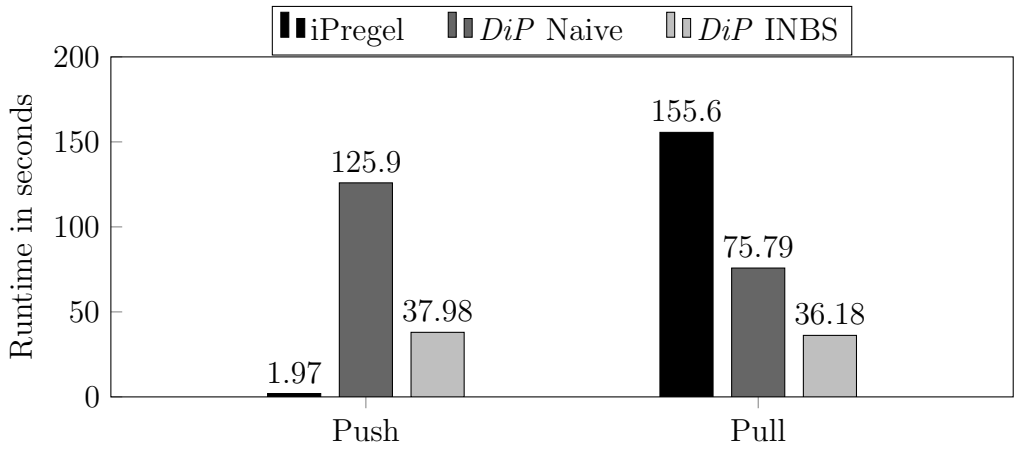

Figure 7.13: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute the first 20 Single-Source Shortest Paths iterations on S_1.5B_16_100K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

Single-Source Shortest Paths The results obtained by running the SSSP benchmark on the S_1.5B_16_100K graph are reported in Figure 7.13. Similarly, to the Connected Components experiment, only the first 20 supersteps were run to keep the runtime within a reasonable limit.

Unlike what is observed for both PageRank and Connected Components benchmarks on this graph, the push version of *iPregel* is significantly faster with SSSP than both *DiP* versions, especially the naive one. With approximately 2 seconds against 125, *iPregel* provides a 60 times speedup over its distributed-memory counterpart. This situation only holds for the push version, however, because runtimes obtained with the pull version are in line with that observed for previous experiments where *iPregel* is slower than the naive implementation of *DiP*. The explanation for the performance of *iPregel* in the push version is twofold. Firstly, this benchmark, SSSP, has a very low number of active vertices in the first 20 supersteps. With a peak of 301 active vertices, out of 1.5 billion, at most 0.0001% of the graph vertices are active in the first 20 supersteps. As explained in Section 7.3.3, in *DiP* the amount of information exchanged during the buffer exchange phase is independent of the number of active vertices, for both push and pull versions. In *iPregel* however, in the push version, inactive vertices are checked but result in no additional processing beyond that check. This checking alone imposes an additional overhead of 52 seconds. The runtime decreases by a further 50 seconds when the selection bypass technique presented in Subsubsection 4.3.4.1 is used. The main benefit of this technique is that only active vertices are iterated, thus skipping 99.9999%

of the graph vertices which are inactive in each of the 20 supersteps. However, the pull version of *iPregel* cannot leverage this technique due to the memory requirement of the selection bypass which, when combined with the pull version, requires more memory than the amount available on a single node.

When comparing the naive implementation against the INBS one, the former remains slower, which is in line with what was observed in previous experiments. However, for the push version, the gap between the naive and INBS implementations is greater than that in previous experiments. Indeed, the INBS implementation completes the first 20 SSSP supersteps in 38 seconds, compared to 126 seconds for the naive implementation, making INBS 3.3 times faster. This is twice the speedup of 1.67 that was obtained between the same two implementations on the PageRank benchmark, where this speedup doubling can be explained by the change in the calculation basis. The runtimes between the PageRank and SSSP benchmarks decreased from 205 to 126 seconds for the naive implementation, and from 123 to 38 for INBS. This is a reduction of approximately 82 seconds for both implementations, due to a faster vertex processing phase caused by the lower number of active vertices in SSSP. Therefore, this difference of slightly more than 80 seconds observed between the runtimes of both implementations on PageRank, is present with SSSP too, where the INBS implementation delivers a runtime that is 88 second lower than that of the naive implementation. However, given that the absolute value of the runtimes is lower on the SSSP benchmark, the difference of approximately 85 seconds translates to a speedup of 3.3 times rather than 1.67 times for PageRank.

Between the push and pull versions, the results collected exhibit three different variations. Firstly, in the case of *iPregel*, an increase of nearly two orders of magnitude is observed when moving to the pull version, requiring over 150 seconds compared to 2 seconds for the push version. As explained earlier, the selection bypass technique is particularly efficient in this scenario, thus considerably improving *iPregel*'s performance compared to without selection bypass. In the case of the push version, where *iPregel* completed in under 2 seconds. A performance gain was expected for the pull version too, similar to that presented in Figure 7 in [1]. However, the combination of the selection bypass technique and the pull version was not possible in this experiment due to the increased memory footprint. Indeed, by design, the pull version requires storing incoming edges, in order to fetch messages. In the meantime, as explained in Subsubsection 4.3.4.1 the selection bypass requires storing outgoing edges in order to schedule the recipients for processing during the next superstep. Considering the S_1.5B_16_100K graph comprising 24 billion edges, each encoding a 4-byte identifier, outgoing edges alone, therefore, require 96GB of memory, implying that storing incoming edges as well would

require an additional 96GB of memory. When added to the memory footprint required by other components of the *iPregel* framework, such as the vertex structures the amount of memory available on a single node became insufficient.

Secondly, for the naive *DiP* implementation, moving from the push to the pull version improves the performance observed, decreasing the runtime from 126 seconds to 76, which is a speedup of 1.65. Similarly, to previous benchmarks, this is due to the buffer exchange phase now being achieved with an MPI broadcast rather than a reduction suffering from poor parallelisation. In addition, the lower number of active vertices results in less vertex processing for the push version thus decreasing its runtime. The workload of the pull version is less proportional to the number of active vertices than the push version, preventing it from benefiting from a similar performance gain in SSSP.

Thirdly, while moving from push to pull results in a runtime increase for *iPregel* and a decrease for the naive *DiP* implementation, it appears to have no impact on that of the INBS *DiP* implementation, which maintains a runtime of approximately 37 seconds. This contrasts to what was observed on PageRank, where the runtime of the INBS *DiP* implementation decreased from 122 seconds to 52 seconds when switching to the pull version. The performance gain observed on PageRank by switching to the pull version was mostly due to a reduction in the runtime of the vertex processing phase. However, with SSSP the low number of active vertices results in a vertex processing phase with a runtime noticeably smaller. When moving to the pull version, the drop observed in the buffer exchange phase is compensated by an equivalent increase due to the introduction of the message fetching phase. Paired with the small runtime of the vertex processing phase, the resulting benefit is close to null.

General Across all 3 benchmarks, the distributed-memory framework *DiP* outperforms *iPregel*, except for the push version in the SSSP benchmark. The INBS implementation provided an additional performance gain compared to the naive implementation in all experiments. The speedups obtained from the INBS implementation against *iPregel* range between 1.8 and 6.8 times. On average, the results collected indicate that a speedup of 3.6 times resulted.

According to the results collected from experiments run in this subsubsection, sparse graphs appear to be beneficial to *DiP*, and even more so to the INBS implementation. Given the architecture change between *iPregel* and the naive *DiP* implementation, this suggests that in the case of sparse graphs, batching messages and keeping direct memory access patterns intra-NUMA regions yields performance improvements. As mentioned earlier, this direction could be pursued further to develop new optimisations in *iPregel*, such as NUMA-local copies paired with a mechanism ensuring their consistency.

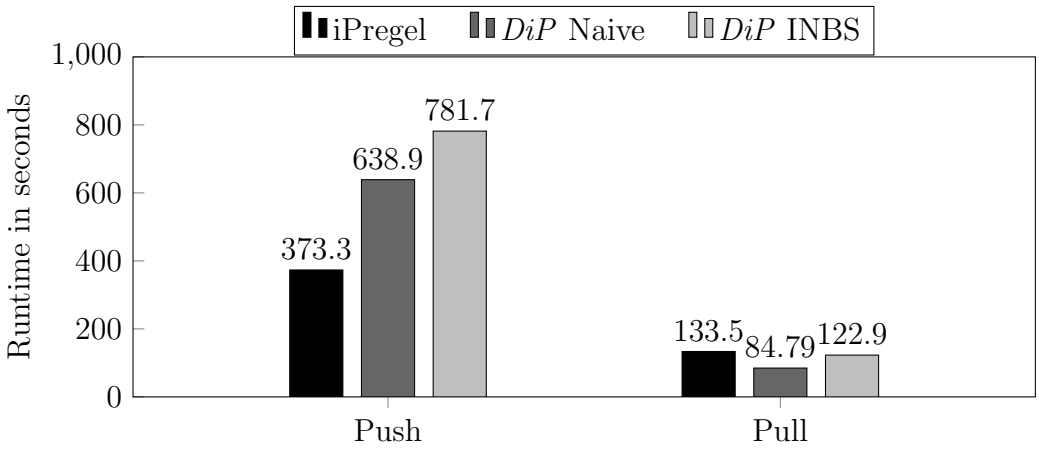

Figure 7.14: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute 10 PageRank iterations on S_50M_1K_10K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

Additional gains were observed for the push version specifically, where the buffer exchange was noticeably faster on the INBS implementation. This is due to the better parallelisation of the underlying single-threaded MPI reduction given that the reduction applied on buffers is distributed across 32 MPI processes instead of 2.

Finally, as an extra benefit of the switch to the INBS implementation, the fact that the INBS implementation moves the replication cost of the *DiP* buffers from per MPI process to per node, which in this case halves the memory footprint per node.

7.6.1.2 Dense graph

By contrast to the sparse graph S_1.5B_16_100K, the S_50M_1K_10K connects 60 times more neighbours to each vertex. Such dense graphs are more representative of social networks, for instance, as seen in [68].

PageRank The runtimes reported in Figure 7.14 show that for the push version, *iPregel* is more than 70% faster than the naive *DiP* implementation. On the pull version, however, the naive *DiP* implementation outperforms *iPregel*, by completing the same 10 PageRank iterations in approximately 85 seconds against 133. This is the opposite of what was observed for PageRank on a sparse graph, where *iPregel* was slower. Therefore the number of vertices and graph density have an impact on performance.

When comparing the naive *DiP* implementation against INBS, it can be observed that the latter is 22% and 45% slower for the push and pull versions respectively, unlike what

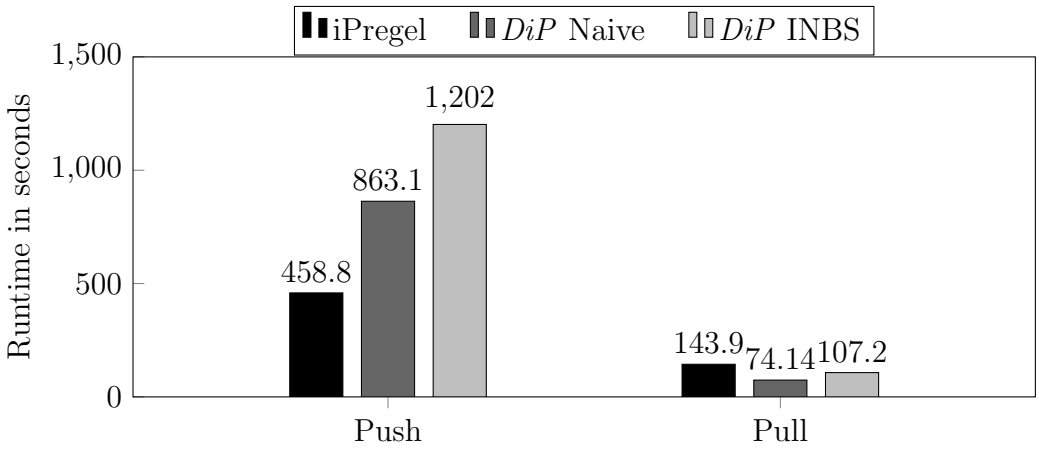

Figure 7.15: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute the first 20 Connected Components iterations on S_50M_1K_10K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

was observed on the sparse graph. For the push version, the main advantage of INBS is the parallelisation of the MPI reduction. For the dense graph however, the number of vertices is 50 million instead of 1.5 billion, and therefore the importance of performance improvements in the reduction reduces by a factor of 60. Furthermore, the batched approach of the naive *DiP* implementation is beneficial when processing a dense graph because of the higher risk of write collisions occurring. Therefore, with dense graphs the main source of performance gains has been lost from INBS and the overhead of this technique now dominates.

Similarly to what was observed on the sparse graph, the characteristics of PageRank make it the optimal benchmark for leveraging the pull version. For each of the three implementations, performance gains were observed, with a speedup of 2.8 for *iPregel*, 7.5 for naive *DiP* and 6.4 for INBS. These performance improvements are higher than those observed for the sparse graph, with an average speedup of 5.6 times compares to 2.4 times. It follows that the explanation lies in the intrinsic characteristics of the graph that changed between the two experiments. Once again, the higher degree of the dense graph implies a higher risk for write collisions because each vertex now has 60 times more neighbours than the sparse graph. The live-locking or extra CAS attempts that result from write collisions imply that moving to the lock-free structure of the pull version yields significant benefits.

Connected Components Figure 7.15 depicts the runtimes measured to complete the first 20 supersteps for the Connected Components benchmark on graph S_50M_1K_10K.

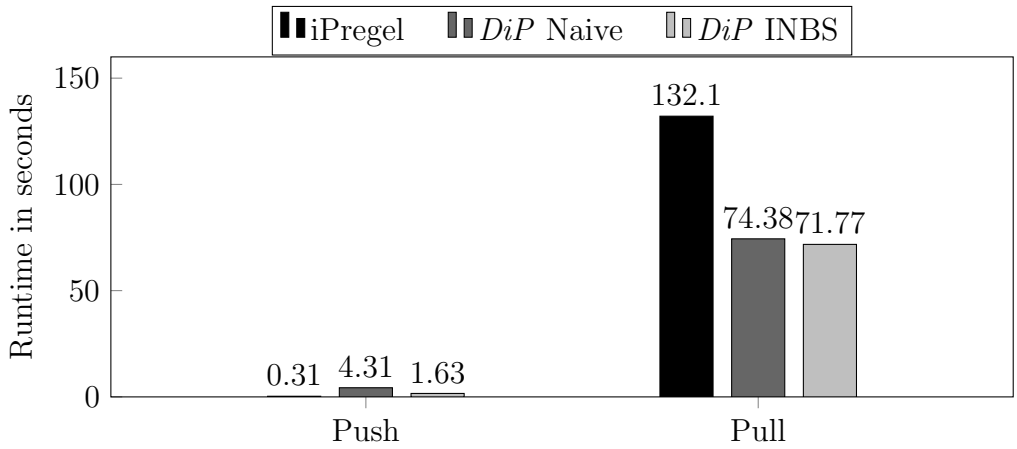

Figure 7.16: Runtime (in seconds) of push and pull versions of *iPregel* and *DiP* to execute the first 20 Single-Source Shortest Paths iterations on S_50M_1K_10K, on one Cirrus node, using 32 threads for *iPregel*, 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

Results obtained are in line with those observed on PageRank in the same conditions. The push version of *iPregel* is almost 90% faster than the naive *DiP* implementation, and the opposite is again observed for the pull versions, where *iPregel* requires twice as much time to complete than the naive *DiP* implementation.

Similarly to what was observed with PageRank, switching from the naive *DiP* implementation to INBS for a dense graph reduces performance between 40% to 45%. On both naive and INBS implementations, switching to the pull version yields speedups of 11.2 times and 11.6 times respectively. The primary source of these performance gains remains the constantly high number of vertices kept active throughout the first 20 supersteps of the Connected Components. Therefore, the same performance trends were observed with Connected Components as with PageRank, with *iPregel* outperforming the naive *DiP* implementation for the push version but not the pull version. Meanwhile, the INBS implementation reduces performance of the naive implementation for both versions by approximately 42%.

Single-Source Shortest Paths Figure 7.16 illustrates the runtimes measured to complete the first 20 supersteps of the SSSP benchmark on the S_50M_1K_10K graph. As mentioned in Subsection 7.5.2, the vertex with identifier 0 was picked as the source.

When comparing *iPregel* against the *DiP* implementations, the runtimes observed vary by a factor of up to 2.3. The *iPregel* framework completes the first 20 supersteps of SSSP in nearly 2 seconds, whilst the naive implementation *DiP* requires approximately 4 seconds and INBS less than 2 seconds. Compared to the sparse graph, the runtimes of

both naive and INBS implementations have been divided by 30. This ratio also corresponds to the ratio between the sparse and dense graphs orders, where the sparse graph contains 30 times more vertices than its dense counterpart. Given that *DiP* implementations do not have the equivalent of the *iPregel* selection bypass technique, they must iterate through each vertex regardless of its active state. This systematic overhead grows to become a major obstacle to *DiP* implementations performance compared to *iPregel* on graphs with a high number of vertices such as the S_1.5B_16_100K. However, on a graph with only 50 million vertices, the overhead appears to be 30 times smaller, permitting the *DiP* implementations to deliver performance comparable to that of *iPregel*.

General The much lower graph order of graph S_50M_1K_10K has meant that *DiP* buffers contained only 50 million elements, compared to 1.5 billion with graph S_1.5B_16_100K. This meant that the MPI reductions of the push version of *DiP* were no longer a source of significant overhead. Similarly for the message exchange, where the reduction by a factor of 30 in the size of the buffer reduced the amount of data passed over the network accordingly. However, the higher risk of write collisions during combinations degraded the overall performance of the *DiP* versions. With the pull versions, however, where by design there cannot be write collisions during combinations, results showed that *DiP* outperformed *iPregel*.

7.6.2 Node scalability

The second set of experiments conducted focussed on evaluating multi-node performance, on the same graphs as Subsection 7.6.1. Due to its high similarity with PageRank, the Connected Components application was not included to avoid redundant series of visualisations and analyses. However, SSSP contrasts against PageRank by having a very low number of active vertices, which makes it a a meaningful and interesting benchmark to include in the experiments presented below.

7.6.2.1 Strong scalability

In this section results are visualised by reporting the runtime against the number of nodes for a specific graph. There are four series on each plot, beginning with the *vertex processing* phase, which reports the time to run the `dip_compute` function on all vertices, including the time needed to handle combinations in flag and value buffers. Secondly the time for *buffer exchange* is reported, which represents the time spent in exchanging the flag and value buffers, as well as updating vertex mailboxes and resetting buffers to neutral values. *Total* reports the sum of the first two, representing the total time spent

in the application whilst the fourth, *Total (ideal)*, depicts what the total runtime should have been assuming ideal node scalability.

PageRank Figure 7.17 shows the runtime scalability of the push version of both naive and INBS *DiP* implementations against the number of nodes when executing 10 PageRank iterations on each graph. It can be observed that, for the sparse graph S_1.5B_16_100K, the runtime of the naive implementation gradually decreases from 200 seconds to approximately 150 seconds at 16 nodes, equivalent to a node scalability efficiency of 8%. The profiling information gathered reports that the runtime is quickly dominated by the buffer exchange phase. Comprising over 40% of the total runtime on one node, the buffer exchange phase grows until eventually accounting for 95% of the runtime at 16 nodes. As explained in Subsection 7.6.1, the poor parallelisation of the MPI reduction in the naive implementation is responsible for the overhead observed in single-node performance.

In the meantime, the INBS implementation does not scale much better, with only a 15% node scalability efficiency at 16 nodes. Starting at 122 seconds on one node, the runtime gradually decreases to 50 seconds at 16 nodes, representing a speedup of approximately 2.5 times. Once again, the runtime eventually becomes dominated by the buffer exchange phase, representing 83% of the runtime at 16 nodes. By running over 32 MPI processes per node, the INBS implementations deliver better parallelisation of the MPI reductions than their naive counterpart. Nonetheless, due to the high number of vertices present in the graph, it remains that the size of the buffers does not permit satisfactory node scalability.

Despite both *DiP* implementations suffering from poor node scalability due to the buffer exchange phase, it can be observed that the INBS implementation clearly outperforms the naive implementation. INBS is 1.2 times faster with 1 node than the naive implementation with 16 nodes, increasing that speedup to 3 when running over 16 nodes too.

The starting point for *DiP*, based on single-node performance, is already 28% and 58% faster than *iPregel* for the naive and INBS *DiP* implementations respectively. Despite the low performance gains, both implementations further improve performance when compared to their shared-memory counterpart. When using 16 nodes, the naive implementation delivers a speed up of 1.9 times against the performance of *iPregel*, which raises to 5.7 times for INBS.

The second row of experiments illustrated in Figure 7.17 focussed on the dense graph S_50M_1K_10K. The trends observed are opposite of those from the sparse graph. The naive *DiP* implementation now delivers near ideal node scalability until 16 nodes. With

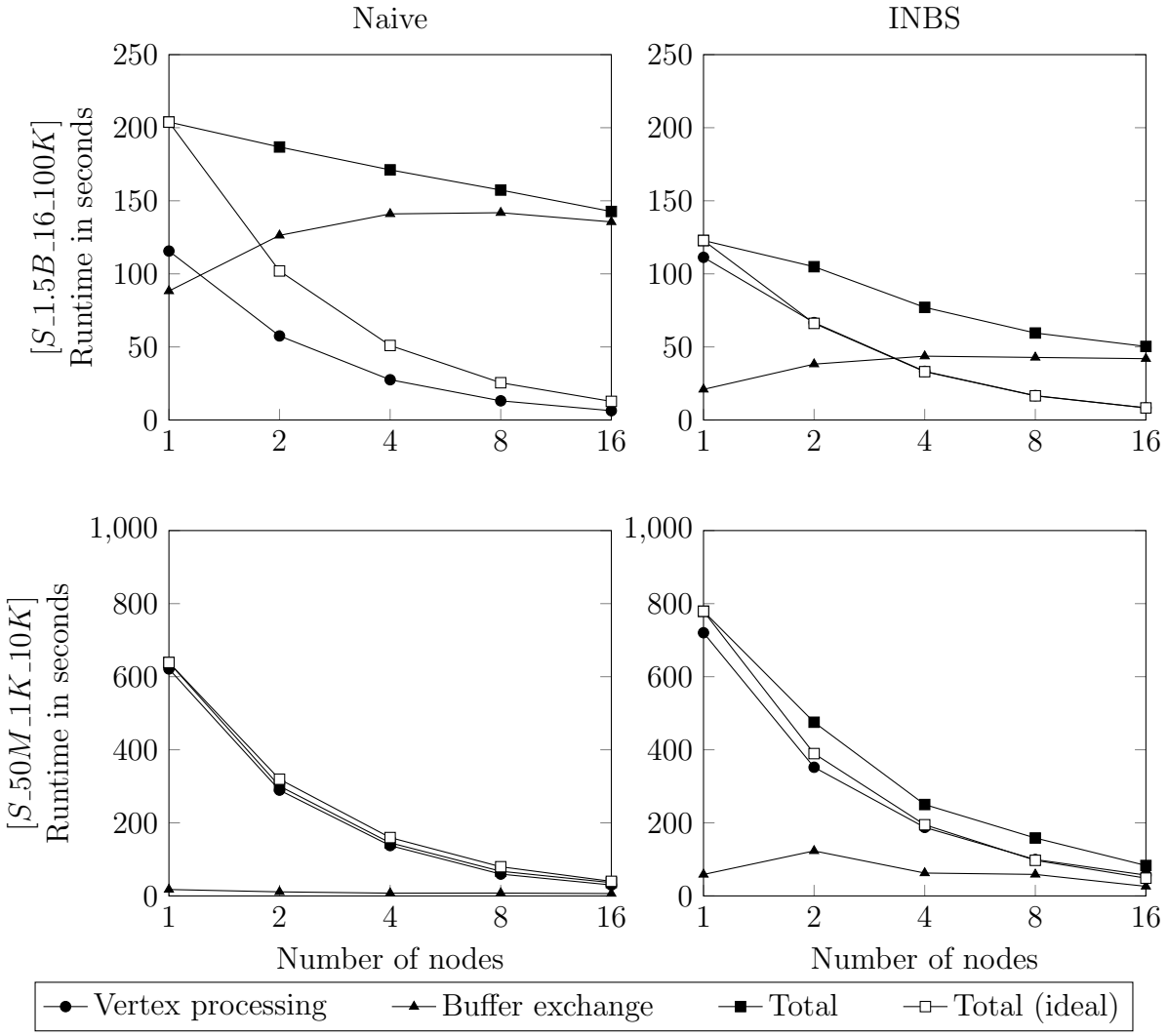

Figure 7.17: Variation of the runtime distribution (in seconds) of the push version of the *DiP* naive and INBS implementations against the number of nodes used to process 10 PageRank iterations on both graphs S_1.5B_16_100K and S_50M_1K_10K, using 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

a single-node runtime of 638 seconds, the 36 seconds measured at 16 nodes represents super-linear scalability. The reason for this difference in performance between the dense and sparse graphs is illustrated by Figure 7.17, where the buffer exchange phase, which accounted for 40% of the single-node performance of the naive implementation on the sparse graph is now less than 3% for the dense graph. This is caused by the number of vertices in the dense graph being 30 times lower than that of the sparse graph, which in turn means that *DiP* buffers are 30 times smaller. Consequently, for the dense graph, the favourable node scalability that is observed is because the vertex processing phase now dominates the runtime at 95% of the overall execution time.

A similar trend is observed for the INBS implementation when applied to the dense graph, despite the runtime not being as close to the ideal. Starting at 780 seconds, the runtime gradually decreases to nearly 80 seconds over 16 nodes. This speedup of 9.4 times obtained corresponds to a node scalability efficiency of 58%. The different runtimes reported in Figure 7.17 highlight that the explanation of this suboptimal performance is multifold. To begin with, the vertex processing phase is measured at 720 seconds in the single node configuration, which involves a starting overhead of 100 seconds, or 14%, compared to that of the naive implementation. Furthermore, by decreasing to 57 seconds over 16 nodes, and a speed up of 12.6 times, the node scalability of the vertex processing phase is not competitive with that observed for the naive implementation. This difference in vertex processing time suggests that, for the *DiP* framework, the batching technique used by the naive implementation is beneficial in the context of dense graphs, where the risk of write collisions is high.

When comparing the push versions of the naive and INBS implementations, two trends are observed. Firstly, the large number of vertices in sparse graphs results in large buffers being required for the buffer exchange. As expected, this is a major obstacle to performance, and even more so with the naive implementation. INBS partially mitigates this overhead by relying on an improved parallelisation of the MPI reduction used in the buffer exchange, resulting in at one node the benchmark outperforming the naive implementation with any number of nodes tested. Secondly, the degree of the graph and the subsequent chances of write collisions determine whether the batching technique used in the naive implementation will be beneficial compared to its direct memory access equivalent in INBS. On dense graphs, the performance showed that, on *DiP*, batching messages not only provided improved better single-node performance, but also great node scalability by exhibiting a linear speedup continuously until 16 nodes.

In addition to experiments run using the push versions of *DiP* implementations, identical experiments were conducted on pull versions. The results collected are re-

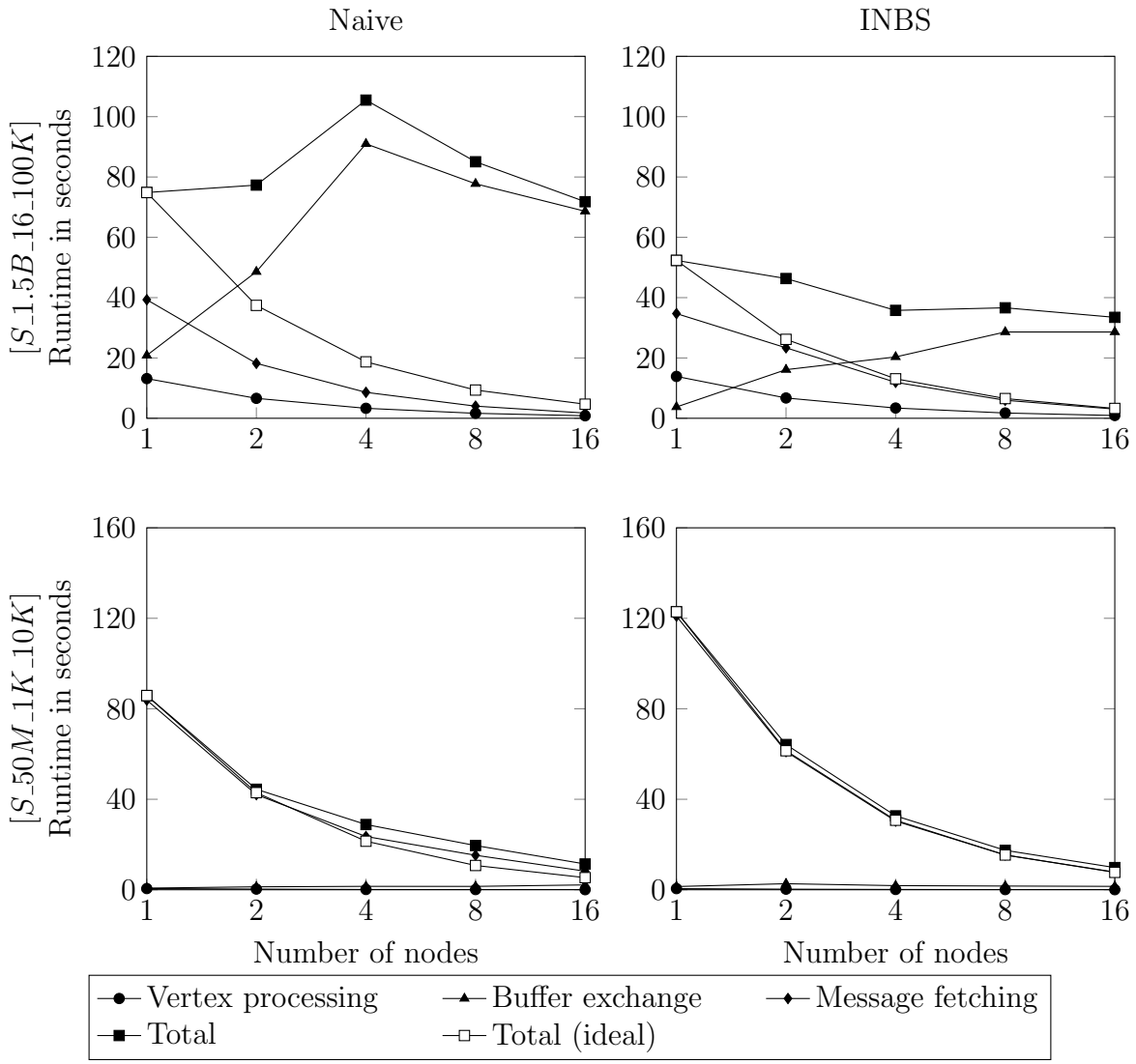

Figure 7.18: Variation of the runtime distribution (in seconds) of the pull version of the *DiP* naive and INBS implementations against the number of nodes used to process 10 PageRank iterations on graphs S_1.5B_16_100K and S_50M_1K_10K, using 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

ported in Figure 7.18. On the sparse graph S_1.5B_16_100K, the INBS implementation demonstrates to deliver single-node performance greater than that of naive *DiP* over 16 nodes, in line with what was observed for push versions. The 20-second runtime difference illustrates a 30% performance gain of the single-node INBS implementation over its naive counterpart with 16 nodes. However, the INBS implementation also delivers poor scaling, providing approximately 10% of node scalability efficiency at 16 nodes. The naive implementation delivers near to no performance gains at 16 nodes. According to the profiling information collected, similarly to identical experiments for push versions, the buffer exchange phase is the cause of this poor node scalability.

Figure 7.18 illustrates that performance becomes dominated by the buffer exchange phase at low node counts, 2 and 4 for naive and INBS implementations respectively. The proportion of runtime required by the buffer exchange phase rises to 96% over 16 nodes for the naive implementation, and 85% for the INBS implementation. However, for the naive implementation, the node scalability of the buffer exchange phase follows one trend before diverging at 4 nodes. Although the high number of vertices continues to be an aggravating factor by resulting in buffers of almost 14GB, no such spike was observed with the push version of the naive implementation on the same sparse graph. The source of this change might therefore be a threshold in the underlying MPI implementation triggering an optimisation in the implementation of the MPI broadcast collective used in pull versions. For instance, the underlying MPI implementation may rely on individual sends or binary-tree decomposition depending on the amount of data emitted per MPI process and the total number of MPI processes. The conditions to trigger such optimisation may be met at 4 nodes, therefore switching to a different broadcast algorithm delivering different node scalability properties.

The message fetching phase, which is a new phase compared to the push version, demonstrates node scalability comparable to that of the vertex processing phase. For the sparse graph this involves approximately half of the runtime that was measured for single-node runs for both naive and INBS implementations. At 16 nodes, however, that proportion reduces to approximately 2% and 9% of the total runtime respectively. As explained in Subsubsection 4.3.4.3, the pull-based combination does not generate write conflicts as threads only issue writes to thread-local memory locations. Reading values however requires memory accesses that can suffer from cross-NUMA region overheads, as observed with the INBS implementation.

With the dense graph S_50M_1K_10K, at 16 nodes, the naive implementation reduces single-node runtime from approximately 85 seconds to 11 seconds, which is a speedup of 7.5 times, or a 47% node scalability efficiency. The INBS implementation improves node

scalability efficiency to 78% over 16 nodes, decreasing the runtime from 123 seconds to 10 seconds. In both cases, the runtime appears to be dominated by the message-fetching phase in all node counts tested. As explained for push versions, the dense graph has relatively few vertices, which results in relatively small buffers which therefore do not noticeably hinder network communication performance. In addition, the vertex processing phase in the pull versions only consists of flagging vertices for broadcast when applicable. Therefore, the classical hotspot of vertex-centric programs, message combinations, takes place solely in the message fetching phase.

For the dense graph, similarly to what was observed with the push versions, the pull version of INBS involves a 43% starting overhead compared to the naive implementation. These results suggest that the batching of messages performed by the naive implementation outperforms the direct memory accesses of its INBS counterpart.

Single-Source Shortest Paths Figure 7.19 depicts the runtime obtained by running the first 20 supersteps of the SSSP application using the push version of both naive and INBS *DiP* implementations, on both sparse and dense graphs, against the number of nodes used.

For the sparse graph, the performance of naive *DiP* counter scales, similarly to what was observed for this configuration with PageRank. Starting with a single-node runtime of 126 seconds, it increases to 155 seconds at 16 nodes. When plotted against the number of nodes, the variation of the runtime first increases and then decreases, with a peak of 170 seconds over 4 nodes. Profiling information shows that 93% to 99% of the runtime is spent on buffer exchange. This is a consequence of the vertex processing phase now requiring only a few seconds. Unlike PageRank, where single-node runtime of the vertex processing phase was 80 seconds, the equivalent starting point in SSSP is at less than 8 seconds. Despite running over a graph comprising 1.5 billion vertices, the first supersteps of SSSP activates only a small number of vertices, which therefore only require a small amount of runtime for processing.

The INBS implementation exhibits worse counter scaling than the naive implementation, albeit over a smaller runtime interval. Whilst the single-node run completes the first 20 SSSP supersteps in 30 seconds, using 16 nodes requires up to 53 seconds, almost doubling the amount of time needed. Nonetheless, over 1 and 16 nodes, the INBS implementation is 4.2 times and 2.9 times faster than its naive counterpart. The runtime distribution drawn from the profiling information gathered indicates that the INBS implementation follows the same performance trend as naive *DiP*. Where the vertex processing phase has become negligible due to the low number of active vertices, which subsequently increases the proportion of the runtime required by the buffer exchange phase.

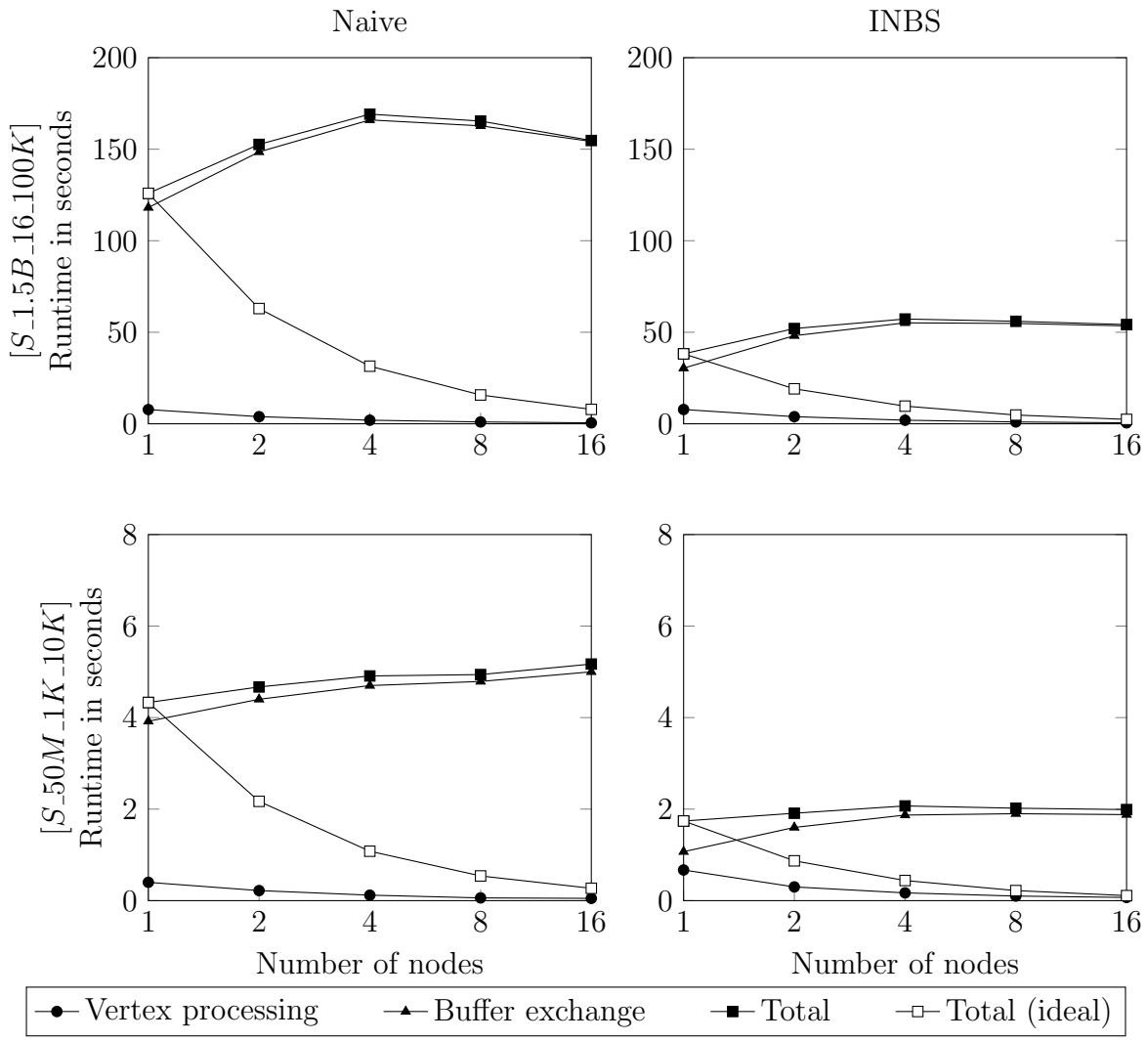

Figure 7.19: Variation of the runtime distribution (in seconds) of the push version of the *DiP* naive and INBS implementations against the number of nodes used to process 20 Single-Source Shortest Paths iterations on graphs S_1.5B_16_100K and S_50M_1K_10K, using 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

The same behaviour is observed for the dense graph, with the naive implementation counter scaling, albeit on a smaller scale. With buffer exchange dominating single-node performance by 91%, the lack of node scalability delivered by the buffer exchange, therefore, means that it is dominant in the overall performance behaviour that is observed. In the meantime, runtime of the vertex processing phase reduces by 8 times between 1 and 16 nodes. Despite exhibiting a node scalability efficiency of 50% at 16 nodes, the performance gains obtained from the vertex processing phase do not outweigh the runtime increase in the buffer exchange on the same node count interval. This is due to the low number of active vertices in SSSP making the vertex process phase no longer significantly impacting the overall runtime.

The INBS implementation exhibits the same performance trends as observed with naive *DiP* over the dense graph. The buffer exchange counterscales, and the subsequent runtime increase is not mitigated by performance gains obtained from the vertex processing node scalability. The total runtime variation ranges from 1.7 seconds over 1 node to 2 seconds over 16 nodes, with a peak of 2.1 seconds over 4 nodes. Although the interval covered is smaller than that of the naive implementation, the variation follows the same pattern, with an increase followed with a decrease.

The results reported in Figure 7.20 are the pull version counterparts of those presented in Figure 7.19, illustrating the runtime of both naive and INBS *DiP* implementation pull versions against the node count, for both graphs, to process the first 20 iterations of the SSSP application.

The trends observed on the sparse graph are similar to those seen on their PageRank equivalents. For the naive implementation, the variation of the runtime against the number of nodes follows a bell curve peaking at four nodes. The runtime over 16 nodes delivers no improvement compared to the single-node performance. As profiling data shows, the cause remains the poor node scalability of the buffer exchange, which dominates the runtime. Due to the number of vertices in the sparse graph, the size of *DiP* buffers becomes a noticeable performance challenge. By contrast, the vertex processing phase delivers performance gains when increasing the number of nodes. However, the low proportion of runtime that this represents does not allow performance gains to outweigh the overhead imposed by network communications.

For the sparse graph, the results collected for INBS are also similar to those obtained on PageRank under the same conditions. Namely, the runtime remains steady, at approximately 35 seconds in this configuration, regardless of the number of nodes used. Once again, the time spent on buffer exchange by INBS increases with the number of nodes. However, the rate at which it increases is lower such that, unlike its naive counterpart,

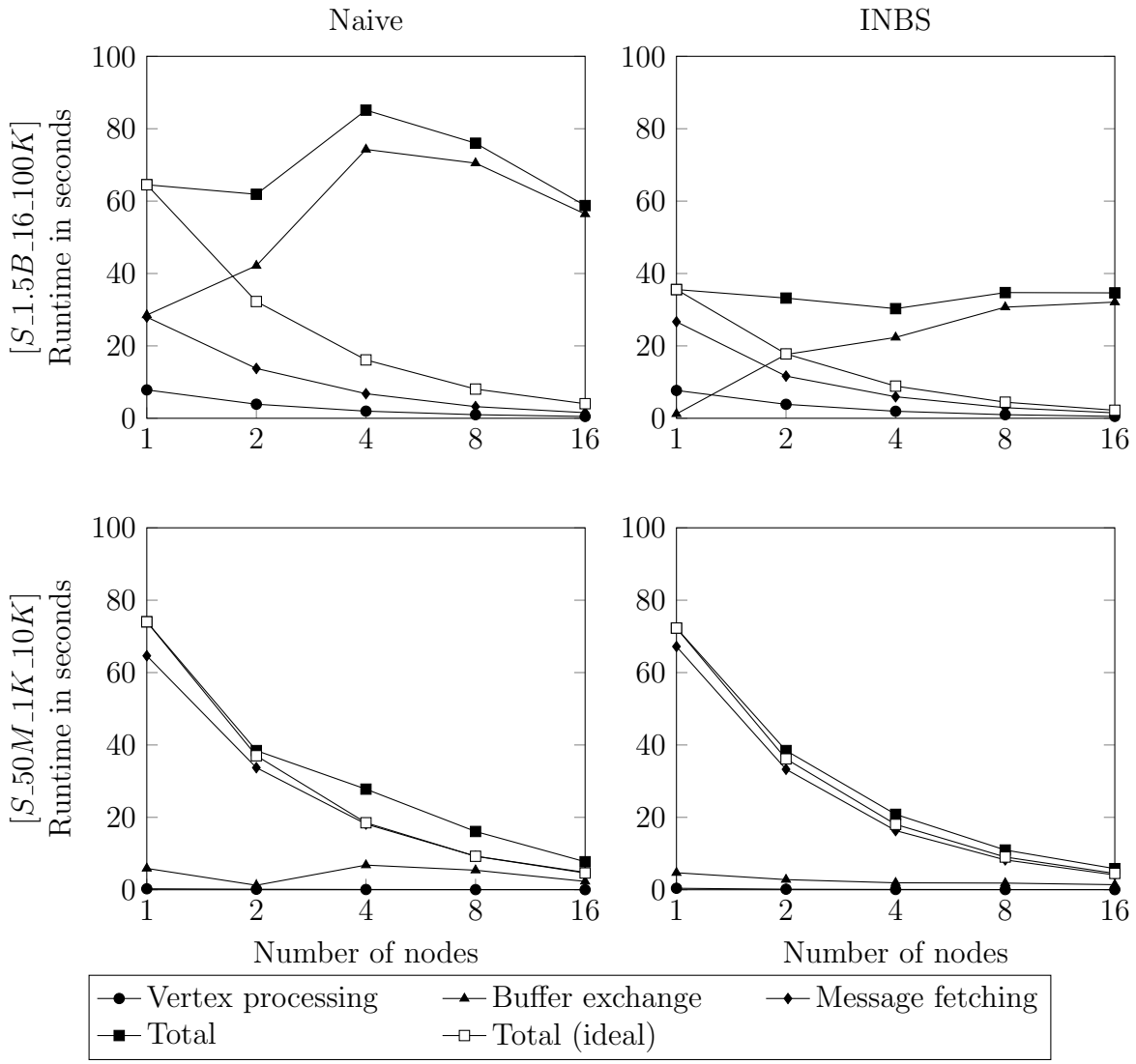

Figure 7.20: Variation of the runtime distribution (in seconds) of the pull version of the *DiP* naive and INBS implementations against the number of nodes used to process the first 20 Single-Source Shortest Paths iterations on graphs S_1.5B_16_100K and S_50M_1K_10K, using 2 MPI processes per node and 16 OpenMP threads per MPI process for the naive implementation and 32 MPI processes per node and 1 OpenMP thread per MPI process for the INBS implementation.

the runtime increase from the buffer exchange phase is negated by the performance gains obtained by the vertex processing and message fetching phases. The reason for this is the runtime starting point in the buffer exchange phase. Both vertex processing and message fetching phases exhibit an identical runtime across naive and INBS implementations. The difference, therefore, comes down to the improvement of network communications in the INBS implementation, thus increasing the proportion of runtime spent on vertex process and buffer exchange, allowing these phases to have an impact on the runtime sufficiently big to hide the growing overhead of the buffer exchange phase.

General Across the 16 sets of strong node scalability results collected in this section, two main trends were visible. Firstly, for graphs with a large number of vertices such as the sparse graph S_1.5B_16_100K, the size of the *DiP* buffers prevents any node scaling benefits. However, the vertex processing phase, which does encounter performance gains when the number of nodes is increased, represents a non-negligible amount of runtime, therefore the performance gains delivered in this phase partially mitigate the growing overhead of network communications. Nevertheless, for graph processing algorithms such as SSSP where the number of active vertices is low, the performance gains from parallelising the vertex processing phase no longer noticeably mitigate the increasing overhead of network communications. Secondly, in graphs with a low number of vertices, the size of *DiP* buffers shrinks accordingly, allowing the runtime of the buffer exchange to reduce to a few seconds. As a consequence, the runtime becomes dominated by the vertex processing or message-fetching phase. The exception is the push version running SSSP which, for the sparse graph, requires only a couple of seconds to handle the vertex processing phase, thus meaning that the buffer exchange is still a significant part of the overall runtime despite the decrease in absolute runtime.

As mentioned in Subsubsection 7.3.3.4, the fundamental weakness of the buffer design implemented in the *DiP* framework is poor node scalability. Although the intra-node buffer sharing technique delays this overhead, and typically results in runtime between two and three times smaller, the node scalability observed is not improved. As seen by results collected from experiments conducted in this subsection, this noticeably impacts performance on graphs with a number of vertices in the billion and above range such as the S_1.5B_16_100K. On graphs with tens of millions of vertices, such as the S_50M_1K_10K, the size of the *DiP* buffers is less than a gigabyte, which means that the buffer exchange does not significantly impact overall performance.

7.6.2.2 Weak scalability

The second node scalability aspect investigated in this section is weak scalability, where the workload per node is kept constant as the node count increases. The graphs selected are scaled up by multiplying both their order and size by the number of nodes used.

In addition to exploring the weak scalability of the *DiP* framework, this subsubsection also intends to provide an insight into the memory footprint scalability of *DiP*. Since the memory footprint of the naive implementation of *DiP* is proportional to the number of MPI processes, whilst that of INBS is proportional to the number of nodes, only the INBS implementation was selected to be executed as part of these weak scaling experiments.

Switch to 64-bit identifiers Initially, both graphs presented in Table 7.1 were selected for weak scaling experiments. However, once scaled up by a factor of four, the graph S_1.5B_16_100K reached an order of six billion, which represents a number of vertices that is no longer encodable by a 32-bit integer. As a result, a switch to 64-bit integers was required. This change doubled the size of a vertex's identifier, which subsequently doubles the amount of memory needed to store edges as well as increasing the size of vertex structures. For graph S_1.5B_16_100K scaled up by four, the total amount of memory required to store only the vertex structures and edges is given by Equation 7.12.

$$\begin{aligned} Total_{INBS} &= S_V \times \frac{|V|}{N} + S_E \times \frac{|E|}{N} \\ &= 40 \times \frac{6 \times 10^9}{4} + 8 \times \frac{96 \times 10^9}{4} \\ &= 60 + 192 \\ &= 252 \end{aligned} \tag{7.12}$$

where $Total_{INBS}$ is the total amount of memory needed per node to store vertex structures and edges, S_V is the size of a vertex structure, $|V|$ is the graph order, N is the number of nodes, S_E is the size of an edge and $|E|$ is the graph size.

Due to this switch to 64-bit vertex identifiers, out of the 256GB available on a node, 252GB are occupied by vertex structures and edges. Whilst the memory footprint of vertex structures increased from 48GB to 60GB, the growth in overall memory footprint is mainly due to the doubling of the edges' memory footprint, now requiring 192GB. Therefore, under these circumstances, processing the graph S_1.5B_16_100K exhausts the memory available on a node of Cirrus, before considering the memory necessary to store the buffers used in the *DiP* buffer design presented in Subsection 7.3.3, or required by MPI and operating system internals.

Table 7.2: Correspondence between weak scaling multiplier and both graph order and size.

Multiplier	S_50M_1K_10K		S_1.5B_8_100K	
	Graph order (in millions)	Graph size (in billions)	Graph order (in millions)	Graph size (in billions)
1	50	50	1,500	12
2	100	100	3,000	24
4	200	200	6,000	48
8	400	400	12,000	96
16	800	800	24,000	192
32	1,600	1,600	48,000	384

As a consequence, graph S_1.5B_16_100K was replaced with graph S_1.5B_8_100K, which contains half the number of edges. The updated list of graphs selected for weak scaling experiments is given in Table 7.2.

PageRank Figure 7.21 shows the runtimes measured to complete 10 PageRank iterations on graphs S_1.5B_8_100K and S_50M_1K_10K. The missing data points on the former reflect out-of-memory failures, where the total amount of memory required exceeded 256GB, which was mostly due to the growing memory footprint of the *DiP* buffers. Over 1 node, for the S_1.5B_8_100K graph, *DiP* buffers account for 13.5GB, eventually reaching 432GB at 32 nodes. The turning point, at which a given graph can no longer be further scaled up without generating out-of-memory failures, differs between push and pull versions on the graph S_1.5B_8_100K. As shown in Figure 7.21, the pull version encounters this point at 8 nodes whereas for this push version this occurs at 4 nodes. Over 8 nodes, 108GB is required to hold *DiP* buffers and 156GB for vertex structures and edges total 254GB, leaving 2GB for all other software layers, including the operating system and MPI internals. However, over 4 nodes, the size of the *DiP* buffers is 54GB, totalling 210GB when including vertex structures and edges. Therefore 46GB of memory remains on the node to contain aspects including the operating system and MPI. For instance, the MPI reduction used in the push version requires the use of internal temporary buffers, unlike the MPI broadcast used in the pull version. The allocation and handling of such temporary buffers are dependent on the MPI implementation used. However, experiments suggest that this overhead may reach up to 30GB for the graph size over 4 nodes, with this third-party memory requirement being the determining factor between a successful run or out-of-memory error.

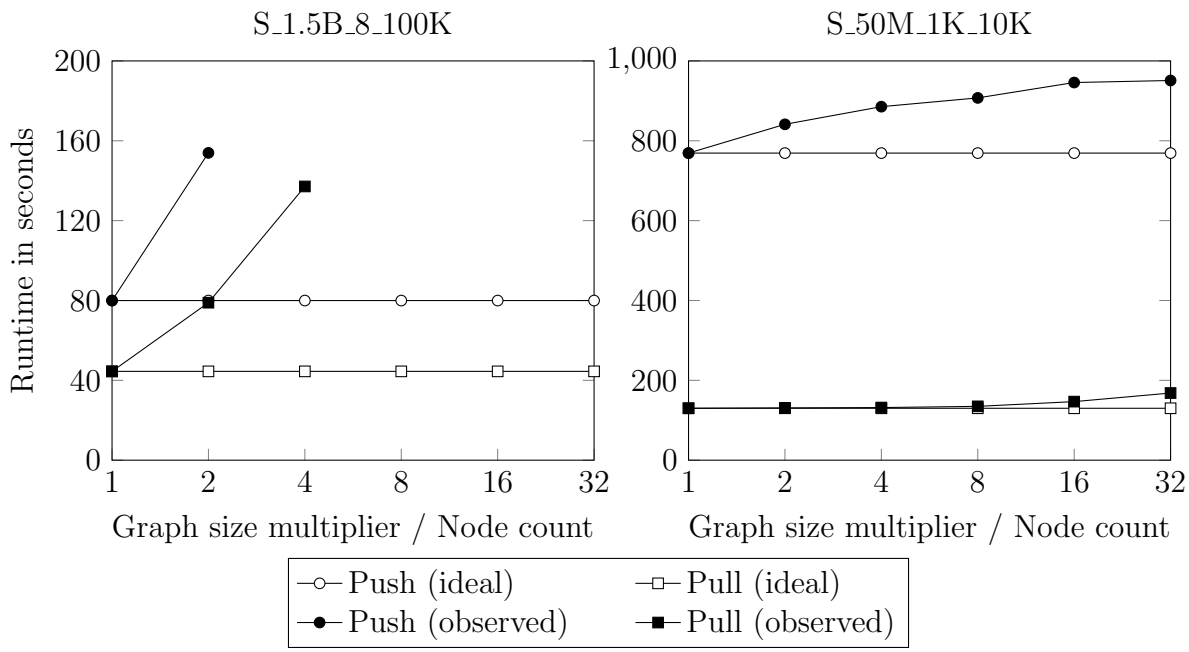

Figure 7.21: Variation of the runtime of the push and pull versions of the INBS implementations against the graph size multiplier, for both graphs S_1.5B_8_100K and S_50M_1K_10K, to process 10 PageRank iterations, using 32 MPI processes per node and 1 OpenMP thread per MPI process.

Despite the out-of-memory failures that were encountered, results collected from experiments for graph S_1.5B_8_100K illustrate that both push and pull versions deliver a runtime which approximately doubles as the weak scaling factor doubles. Given that weak scaling experiments keep the local computation constant on each node, the runtime increase observed is therefore caused by network communications. Nodes are connected with InfiniBand providing a bandwidth of 54.5Gb/s. Over 2 nodes, the 27GB of *DiP* buffers, therefore, require at-best 4 seconds to be transmitted. Given that 10 iterations of PageRank are executed, this corresponds closely to the difference in execution time observed between 1 and 2 nodes for the pull version, and similarly between 2 and 4 nodes. The increase observed on the push version, however, is greater than 40 seconds because of the reduction operation.

By comparing the weak scalability on both graphs, it can be seen that the pull version performs better on graph S_50M_1K_10K than on S_1.5B_8_100K. This is explained by the size of the buffers being 30 times smaller, due to the number of vertices being 30 times lower. However, the pull version on S_50M_1K_10K also demonstrates weak scalability behaviour that is different to that of its push counterpart, highlighting the presence of the extra operations required in an MPI reduction, such as buffer copies and reduction operation, compared to an MPI broadcast.

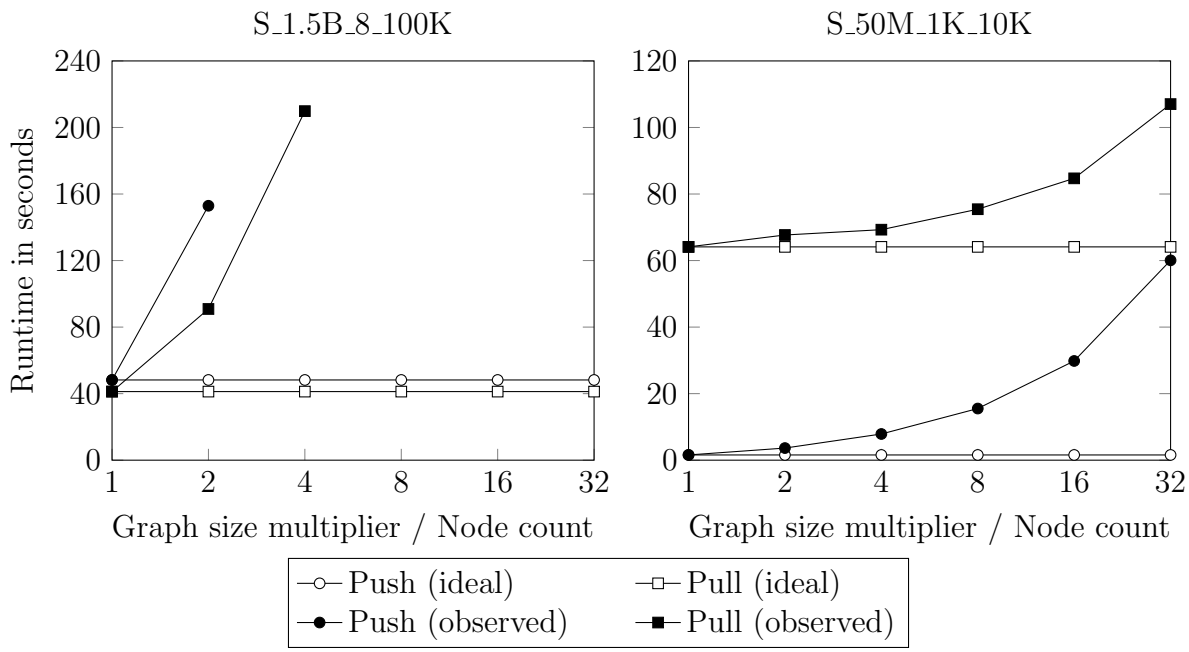

Figure 7.22: Variation of the runtime of the push and pull versions of the INBS implementation against the graph size multiplier, for both graphs S_1.5B_8_100K and S_50M_1K_10K, to process the first 20 Single-Source Shortest Paths iterations, using 32 MPI processes per node and 1 OpenMP thread per MPI process.

In addition to inducing out-of-memory failures, the worst weak scalability is observed on graph S_1.5B_8_100K, experiencing an overhead of 250% over 4 nodes. As explained throughout previous sections, the high number of vertices in such graphs results in large buffers, and therefore expensive network communications and reduction operations. Conversely, the best weak scalability is seen with the pull version on graph S_50M_1K_10K, which, over 32 nodes, processes the biggest graph generated in this research, 1.6 trillion edges, while limiting the overhead to 30%.

Single-Source Shortest Paths Figure 7.22 reports weak scalability of both the push and pull versions of *DiP* when processing the first 20 supersteps of SSSP on S_1.5B_8_100K and S_50M_1K_10K graphs. Similarly, to PageRank, for graph S_1.5B_8_100K, out-of-memory failures are generated from 4 nodes onwards for the push version and 8 nodes onwards for the pull version. Although differences exist between PageRank and SSSP on an algorithmic basis, they do not influence the memory footprint, thus raising out-of-memory failures at the same node counts.

Moreover, the weak scaling behaviour observed for the S_1.5B_8_100K graph is consistent with that observed for PageRank. The low number of active vertices present in SSSP permits the push version to provide performance equal to that of the pull version.

However the low number of active vertices does not influence network communications, which have been analysed above as the cause of this runtime increase.

The results obtained by running SSSP over the S_50M_1K_10K graph illustrate that there is an overhead of approximately 60 seconds over 32 nodes, for both push and pull versions. However, the small number of active vertices in SSSP cannot be exploited by the pull version, which, as observed in previous sections, is required to process each vertex regardless of its state. The ability of the push version to skip inactive vertices allows it to complete the first 20 supersteps of SSSP in only a few seconds at one node, compared to over 60 for its pull counterpart.

Finally, in the case of the push version on the S_50M_1K_10K graph, when weakly scaled by a factor of 32, the performance observed when processing the corresponding graph of 1.6 trillion edges is equivalent to more than 250 billion edges traversed per second.

General The weak scaling experiments presented in this subsubsection support the observations made in the strong scaling experiments, where the impact of network communications on graphs with a higher order is greater due to the size of *DiP* buffers growing proportionally to the graph order.

Furthermore, the reduction which is applied in the push version typically results in an overall performance worse than its pull counterpart, which instead relies on simpler broadcast and lockless memory accesses. The exception to this is SSSP, where the ability of the push version to skip inactive vertices outweighs the benefits from the pull versions.

Finally, these experiments also demonstrate that when processing a graph comprising 1.6 trillion edges, the *DiP* framework is able to maintain its overhead at 30% in the case of PageRank, or traverse 250 billion edges per second in the case of SSSP.

7.6.3 Memory footprint

Memory footprint is the second aspect evaluated in this section. As highlighted in previous chapters, the impact of memory footprint can be two-fold, impacting performance as well as generating an out-of-memory failure. In distributed-memory systems, the latter can usually be overcome by the addition of nodes. However, the replicated nature of the buffer design presented in Subsection 7.3.3 challenges this possibility, in return for the multiple benefits presented in Subsubsection 7.3.3.3.

7.6.3.1 Predictability

Although the memory footprint of the *DiP* buffers is replicated, it does not vary throughout execution and therefore can be calculated before execution. These properties make the memory footprint of *DiP* predictable, thus helping foresee out-of-memory failures or anticipate hardware requirements given the size of the graph considered. Moreover, in *DiP*, any out-of-memory failure caused by the buffer design presented in Subsection 7.3.3 occurs during the initialisation phase. This contrasts against those buffer designs which are based on dynamically resizeable structures, which may trigger an out-of-memory failure virtually at any point during execution. This may prove particularly inconvenient when occurring towards the end of long, and expensive, graph processing jobs.

The formula to calculate the memory usage per node of the naive *DiP* implementation is given in Equation 7.13, assuming an identical number of MPI processes on each node.

$$M_{NAIVE} = (S_F + S_M) \times |V| \times P_N + \left(\frac{|V| \times S_V + |E| \times S_E}{N} \right) + \zeta \tag{7.13}$$

where M_{NAIVE} is the memory usage per node for the naive implementation, M_B is the amount of memory needed to store both flag and value buffers as described in Equation 7.1, P_N is the number of MPI processes per node, $|V|$ is the graph order, S_V is the size of a vertex structure, $|E|$ is the graph size, S_E is the size of an edge and ζ is the memory footprint of all underlying software layers. As explained in Subsection 7.4.2, the memory footprint of the intra-node buffer sharing optimisation decreases the replication level from per MPI process to per node, thus yielding Equation 7.14.

$$M_{INBS} = (S_F + S_M) \times |V| + \left(\frac{|V| \times S_V + |E| \times S_E}{N} \right) + \zeta \tag{7.14}$$

where M_{INBS} is the memory usage per node for the INBS implementation

However, as shown in weak scaling experiments presented in Subsubsection 7.6.2.2, the memory footprint of underlying software layers represented by ζ must not be neglected. In situations where concurrent non-blocking reductions occur, the implicit use of internal temporary buffers by MPI_Reduce can dynamically allocate a memory amount equal to the size of *DiP* buffers: $(S_F + S_M) \times |V| \times P_N$ or $(S_F + S_M) \times |V|$, for the naive and INBS implementations respectively.

7.6.3.2 Size

As demonstrated by the previous subsubsection, the memory footprint of the naive implementation, when using one MPI process, is equal to that of the INBS implementation.

The difference for the naive implementation comes from the number of additional MPI processes per node. Therefore, only the INBS implementation is considered in the experiments conducted in this subsubsection.

Single node Equation 7.14 provides the formula to calculate the memory footprint required by the INBS implementation. When considering the single node experiments presented in Subsection 7.6.1, the resulting memory footprint obtained for each graph has two possible values depending upon the benchmark selected. The term S_M, which represents the size of the type used to encode message payloads, is commonly a double-precision float (8 bytes) in PageRank, while being of the same type as the vertex identifier (4 bytes for the graph considered) in both CC and SSSP.

Processing PageRank on the graph S_1.5B_16_100K requires 157.5GB of memory, out of which 13.5GB are due to the *DiP* buffers. When compared against the total memory footprint, *DiP* buffers represent less than 9%. The difference highlighted in the previous paragraph for CC and SSSP, results in 151.5GB being required instead, where 7.5GB are allocated to *DiP* buffers or approximately 5% of the overall memory footprint. Because the size of *DiP* buffers is proportional to the graph order, this makes them larger on graphs such as S_1.5B_16_100K.

Conversely, the graph S_50M_1K_10K has a relatively low graph order, resulting in smaller *DiP* buffers. To process PageRank, a total of 203GB is required, compared to 158GB on S_1.5B_16_100K. This increase is due to the larger graph size. However, the size of the underlying *DiP* buffers is 0.45GB or less than 1% of the total memory footprint. This is a reduction of 97% compared to what was observed for the S_1.5B_16_100K graph. Similarly, for CC and SSSP, only 0.25GB of memory is required for storing *DiP* buffers, out of a total of 202GB.

When compared against the amount of memory available on a node, which is 256GB, the portion taken by *DiP* buffers ranges between 0.1% and 5%. In other words, the memory footprint overhead of the buffer design in *DiP* is relatively low for the graphs considered here.

Strong scaling As explained in Subsubsection 7.3.3.4, the main weakness of the buffer design introduced in Subsection 7.3.3 is its replication across nodes for the INBS implementation. Because the term $(S_F + S_M) \times |V|$ from Equation 7.14 remains constant, this is the limit towards which the memory footprint tends to grow as the node count increases. Table 7.3 lists the amount of memory required per node, calculated from Equation 7.14 (neglecting ζ), in the configuration used in strong scaling experiments presented in Subsubsection 7.6.2.1.

Node count	S_1.5B_16_100K		S_50M_1K_10K	
	PR	CC or SSSP	PR	CC or SSSP
1	157.5	151.5	202.5	202.1
2	85.5	79.5	101.7	101.3
4	49.5	43.5	51.3	50.9
8	31.5	25.5	26.1	25.7
16	22.5	16.5	13.5	13.1

Table 7.3: Total memory footprint (in GB) of the INBS implementation against the node count, to process PageRank, Connected Components and Single-Source Shortest Paths on both S_1.5B_16_100K and S_50M_1K_10K graphs, using one MPI process per node.

A metric similar to node scalability could be used on the memory footprint, which assesses the extent to which the memory requirements grow as the node count increases. Considering graph S_1.5B_16_100K for instance, the memory footprint of *DiP* buffers has been calculated at 13.5GB or 9% of the total memory footprint for PageRank. The remaining 91% are therefore distributed across the nodes. This means that, from 12 nodes onwards, more than half of the entire memory footprint on a node is occupied by the *DiP* buffers. This is illustrated in Table 7.3 at 16 nodes for instance, where 13.5GB, or 60%, of the 22.5GB are allocated to *DiP* buffers.

Continuing the analogy with performance speedup and parallel efficiency, at 16 nodes, the original memory footprint is divided by 7, or 44% compared to what can be considered as the ideal where the memory footprint would have been divided by 16. Due to their smaller messages on this graph, Connected Components and Single-Source Shortest Paths the memory footprint of CC and SSSP are dominated by the *DiP* buffers at 20 nodes instead of 12. The memory footprint of 16.5GB calculated over 16 nodes is therefore relatively better, being 9.2 times lower than its single-node counterpart, equivalent to 57% of the ideal.

Due to its lower graph order, graph S_50M_1K_10K results in smaller *DiP* buffers while generating a larger memory footprint overall. With PageRank, for instance, over 16 nodes, the memory footprint is divided by 15, which is 94% of the ideal. Similarly, for CC and SSSP where the memory footprint calculated is 15.4 times lower than the single-node one, equivalent to 96% of the ideal.

As a result, this analysis demonstrates that the core weakness of the *DiP* buffers, their replication, does not prevent this design from delivering strong scaling gains.

Weak scaling As observed in weak scaling experiments conducted in Subsection 7.6.2.2, the growing size of *DiP* buffers eventually led both push and pull versions to exhaust memory. For graph S_1.5B_8_100K, this occurred over two and four nodes, for the push

and pull versions respectively. Processing the graph S_50M_1K_10K, however, did not generate such errors in the range of weak scaling factors considered. According to Equation 7.14, at 64 nodes, the memory footprint of the INBS version, excluding ζ, is estimated at 230GB, which remains under the 256GB available on a node. However, the estimation for 128 nodes projects a memory footprint, excluding ζ, of 460GB due to the switch to 64-bit vertex identifiers.

The weak scaling experiments of Subsubsection 7.6.2.2, stopped over 32 nodes, and at this scale the 256GB of RAM available on each node total 8,192GB of distributed memory. For the scaled-up version of graph S_50M_1K_10K, which contains 1.6 trillion edges, storing edges requires 6,400 GB of RAM, or 200GB per node, assuming 4-byte vertex identifiers. In *DiP*, the buffer design introduced in Section 7.3.3 requires 14GB per node to process PageRank, representing only 7% of the total memory footprint, excluding ζ. This is similar to the observations made in the previous subsubsection focussing on strong scaling.

Therefore, also in a weak scaling context, the efficiency of memory footprint of *DiP*'s buffers can vary significantly between graphs, from experiencing out-of-memory failures over as few as 4 nodes, to enabling a trillion-edge graph to be processed over a thousand cores while generating a memory overhead as small as 400MB per core.

7.6.3.3 IBMP

As mentioned in Subsubsection 7.6.2.2, while the out-of-memory failures encountered with the original S_1.5B_16_100K graph were triggered by the switch to 64-bit types, those raised with graph S_1.5B_8_100K however were due to the growth of *DiP* buffers. For example, on the latter graph with a weak scaling factor of 32, *DiP* buffers alone would occupy a total of 432GB of memory.

This issue can be addressed with the IBMP technique introduced in Subsection 7.4.3, enabling *DiP* to limit the memory footprint of *DiP* buffers to a threshold specified by the user. At the time of writing, the implementation of the IBMP technique is not fully functional for some of the large-scale graph processing considered in the weak scaling experiments. Nonetheless, the results that have been collected are discussed in this section.

Table 7.4 summarises the findings obtained when running, in a weak scaling configuration, PageRank with the push version of the IBMP implementation on the graph S_1.5B_8_100K. The results obtained broadly support the analysis from Subsection 7.4.3, where an overhead in performance is encountered due to the multiple rounds of processing required. On a single node, the 3 IBMP rounds that are required result in an

Weak scaling factor	Number of IBMP rounds	IBMP runtime (in seconds)	INBS runtime (in seconds)	Slowdown
1	3	255.41	79.96	3.19
2	6	652.52	153.94	4.23
4	11	1295.86	-	-

Table 7.4: Summary of weak scaling experiments run on graph S_1.5B_8_100K with the push version of the IBMP implementation, to process 10 PageRank iterations using an IBMP allocation of 10GB per MPI process, using 1 MPI process per node and 32 OpenMP threads per MPI process

Weak scaling factor	Number of IBMP rounds	IBMP runtime (in seconds)	INBS runtime (in seconds)	Slowdown
1	3	189.00	44.49	4.25
2	6	332.04	78.89	4.20
4	11	584.03	137.12	4.26

Table 7.5: Summary of weak scaling experiments run on graph S_1.5B_8_100K with the pull version of the IBMP implementation, to process 10 PageRank iterations using an IBMP allocation of 10GB per MPI process, using 1 MPI process per node and 32 OpenMP threads per MPI process

overall runtime that is 3.2 times higher than the INBS reference. With a weak scaling factor of 2, the runtime obtained is 4.2 times higher than that of the INBS implementation, instead of 6, which could be expected given that 6 IBMP rounds are executed. This difference can be explained by the fact that only local computation is indeed repeated across IBMP rounds to combine messages, however, communications are issued only once overall at the end of the superstep. Moreover, the performance observed does not seem to suffer from the poor parallelisation of the MPI reduction highlighted earlier in this chapter. As shown in Figure 7.7, when selecting one MPI process per node, the IBMP implementation alternatively uses two buffers so that local computation and communications overlap. This overlapping enables the non-blocking MPI reduction to execute concurrently with vertex processing, reducing the overhead of communication. Most importantly, at a weak scaling factor of 4, the IBMP implementation successfully executes PageRank whilst INBS results in an out-of-memory failure. This particular experiment acts as a proof of concept, demonstrating the ability of the IBMP technique to enable *DiP* to process a graph it otherwise could not.

Table 7.5 reports runtime results from the pull version of the *DiP* framework. The runtime of IBMP is again proportional to the number of rounds executed. In this configuration, however, no out-of-memory failure was generated by the INBS implementation on the weak scaling interval considered. The INBS implementation proves to be approx-

imately four times faster in all three experiments, however, performance of the IBMP technique is secondary, as the primary objective is to enable *DiP* to process graphs that otherwise would exhaust memory.

Results collected, therefore, demonstrate that the IBMP technique enables the *DiP* framework to process graphs that would otherwise be impractical. An example of this is provided by Table 7.4, where IBMP enables *DiP* to process a graph of 6 billion vertices and 48 billion edges.

7.6.4 Programmability

The third and final aspect used in the evaluation of the *DiP* framework is programmability. More specifically, assessing whether the original vertex-centric abstractions have been preserved.

The fundamental vertex-centric abstraction remains the expression of computation from the perspective of a vertex. Not requiring the user to retain a global view of the graph, but instead providing them with a localised approach to graph computation is the core objective of vertex-centric programming. As shown in Subsection 7.5.4, implementing benchmarks in the *DiP* framework conforms by this paradigm by allowing the user to express computation from a vertex's point of view via a single function, `dip_compute`. As mentioned in Section 7.3, the interface provided by *DiP* has been inspired and built from that of *iPregel*. A difference is that, in its current state, the *DiP* framework restricts the combination operations that are possible to the set of predefined reduction operations in MPI. This is a precondition to the original assumption that MPI one-sided operations, which do not accept user-defined operations, may be used in *DiP*. Nevertheless, the standard set of predefined MPI reduction operations covers the majority of operations required by graph processing combiners, and the *DiP* framework could be extended in the future to accept arbitrary operations.

Another important component of this programming model is the ability for the user to focus on the implementation of their algorithm, whilst performance-related aspects are offloaded to the underlying framework. To that end explicit parallelism is not exposed to the user, and neither are atomic operations. In *DiP*, parallelism is automatically applied and no low-level concerns are required to be addressed by the user, and this was illustrated by the benchmark implementations that were discussed in Subsection 7.5.4.

The final factor used in this consideration of programmability is the message-passing abstraction provided to implement inter-vertex interactions. In *DiP*, the user continues to be provided with a message-passing abstraction via the `dip_send` and `dip_broadcast` functions, following the same usage behaviour as corresponding *iPregel* functions.

Therefore, based on the above, it is concluded that the *DiP* framework successfully preserves the cornerstone of vertex-centric, programmability providing the user with the very programming abstractions that define this programming model, regardless of the underlying implementation used whether it be naive, INBS or IBMP.

7.7 Conclusions and future work

The work presented in this chapter focussed on the design and development of the distributed-memory framework *DiP*, porting the overarching philosophy, as well as adapting optimisation techniques, from the shared-memory framework *iPregel*. At its core, the buffer design introduced in Subsection 7.3.3, leverages the direct mapping property to provide numerous benefits to the overall vertex-centric workflow.

For the sender, this buffer design removes the need for hash-based indexing of vertices, commonly found in state-of-the-art distributed-memory vertex-centric frameworks. By semantically enriching the array indexes to represent vertex identifiers, the direct mapping property of this buffer design also removed the need to append recipient vertex identifiers to generated messages. Moreover, by not relying on a dynamically resizeable structure nor delaying sender-side combinations, this buffer design also removed the need for the sorting phase that is commonly required by other frameworks. This consequently reduced the algorithmic complexity of the message generation phase from $O(n \times log_2(n))$, or $O(n \times log_2(n) + n)$, as explained in Section 7.2; to $O(n)$, as explained in Subsubsection 7.3.3.3. Benefits from the *DiP* buffer design are also observed on the receiver side, where the message unwrapping phase common in distributed-memory frameworks is no longer necessary as vertex identifiers are not appended to messages. Furthermore, message dispatches are also no longer needed because the combined data can be received directly into the vertex recipient's mailbox, due to the externalisation of vertex structure members to guarantee contiguity in memory. The final benefit of this buffer design is the limitation of its maximum memory footprint, from being proportional to the graph size in most distributed-memory frameworks [12, 25, 88], to being proportional to the graph order, which is typically at least an order of magnitude smaller.

However, despite the benefits listed above, this buffer design also encounters a fundamental weakness: its memory footprint is fixed and replicated. In other words, the memory footprint is constant, and this is independent of the number of empty elements which are contained within the buffer. This fixing of the memory size is required to support direct mapping, and each MPI process must store buffers with a number of elements equal to the graph order. The replicated nature of buffers partially negates the benefits

of adding more nodes, and also implies that this will not address situations where the memory footprint of the *DiP* buffers alone exceeds the amount of memory available on a node. In addition, its replication basis is the MPI process, meaning that the maximum number of MPI processes that can be used is also limited, itself resulting in a poor parallelisation of underlying MPI collective operations used in the buffer exchange phase, such as the MPI reduction.

To address the parallelisation issue, as well as attempting to reduce the memory footprint, the intra-node buffer sharing (INBS) technique has been developed and was presented in Subsection 7.4.2. By leveraging shared-memory, the INBS technique enables MPI processes residing on a given node to share their *DiP* buffers, therefore allocating only one buffer per node. This technique effectively decouples the memory footprint per node of *DiP* from the number of MPI processes residing on it. As a result, a higher number of MPI processes per node can be used, therefore improving the parallelisation of MPI collective operations. Furthermore, with a careful design of communication patterns, and the use of several levels of communicators, the buffer exchange phase is decomposed into a series of concurrent collective operations making use of all MPI processes while minimising the number of MPI processes taking part in each series. This yielded noticeable performance gains, both for single-node performance as well as node scalability. This was particularly evident for the push version, where the single-threaded execution of the MPI reduction was penalising overall performance.

Nonetheless, the *DiP* buffers must continue to allocate a number of elements equal to the graph order, thus limiting the maximum graph order that can be processed. To address this issue, the interval-based message processing technique has been developed, introduced in Section 7.4.3. This provides *DiP* with the ability to constrain the size of buffers within the amount of memory specified by the user at the expense of additional computation. The IBMP technique therefore virtually removes the risk of out-of-memory failures due to *DiP* buffers. Experiments conducted in Subsubsection 7.6.3.3 demonstrated that, with the use of the IBMP technique, the *DiP* framework is able to process a graph comprising 6 billion vertices and 48 billion edges, which it was unable to do so using the INBS implementation.

Overall, experiments demonstrated that the *DiP* framework manages to deliver the performance of *iPregel* to distributed-memory architectures. Over a single node, *DiP* outperformed *iPregel* in eight out of twelve experiments. However, the selection bypass technique introduced in Subsubsection 4.3.4.1 enables *iPregel* to continue to perform well when an application with a typically low number of active vertices in its execution flow, such as SSSP, is applied to a graph with high order. This resulted in a performance

advantage over *DiP* of up to a factor of 60. Investigating the feasibility of porting this technique to distributed-memory could therefore greatly improve the performance of vertex-centric frameworks in similar configurations, including *DiP*. In addition to single-node experiments, multi-node experiments were conducted. Numerous scenarios were considered and results showed that whilst *DiP* does exhibit poor scalability in certain situations, typically on graphs with a high order, it can prove very efficient in other situations.

When considering future directions, the IBMP technique evaluated in Subsubsection 7.6.3.3 may prove to have unexploited potential. For instance, the memory footprint of IBMP buffers remains replicated on a per MPI process basis. It would therefore benefit from the buffer-sharing feature provided by the INBS technique. Furthermore, early algorithmic analysis shows that the IBMP and INBS techniques are not mutually exclusive, but a new implementation would need to be developed to exploit this observation. Another potential direction for future work could explore the use of double buffering in vertex-centric programs. As observed with the push version, computation and communication overlapping obtained by the alternating use of two buffers in the IBMP implementation appeared to mitigate some of the expected performance penalty due to the poor parallelisation of the MPI reduction. Finally, in addition to the porting of the selection bypass technique to distributed-memory, another direction to investigate could be that of a buffer-less, or near buffer-less, design where messages are sent as soon as generated. Such an approach would require technologies that focus on low latency, typically one-sided, communications. An attempt has been made in this research, using the *GASPI* [92] technology. However, the use of GASPI was not possible due to implementation limitations incompatible with the support of combiners. In *GASPI*, the function that implements a one-sided accumulation, equivalent to MPI_Accumulate in MPI, only supports the sum combination operation. Although the inability to pass user-defined operations to one-sided functions is present in MPI too, the latter does provide a standard set of predefined operations covering most reduction operations required by commonly used graph algorithms. Another option in *GASPI* is to use the one-sided remote compare-and-swap, however restrictions imposed on the type of the target variable meant that this was not suitable in its current form.

Chapter 8

Conclusions and future work

This chapter summarises the findings presented in this thesis and discusses their relevance towards the verification of the original research hypothesis given in Chapter 1, as well as presenting potential future directions that may prove interesting to explore.

The first research direction consisted in demonstrating whether optimisation techniques can be developed without sacrificing programmability in vertex-centric shared-memory frameworks. As shown in Chapter 4, existing shared-memory frameworks can be divided into those preserving vertex-centric programmability, and those that sacrifice vertex-centric abstractions for performance and memory efficiency gains. Experiments conducted demonstrate that the former suffers from performance and memory efficiency penalties of orders of magnitudes, while in the latter a removal of certain fundamental vertex-centric abstractions, such as message-passing or exposing explicit parallelism to the user, is observed.

The vertex-centric framework developed in this research, *iPregel*, implements a highly modular design that allows underlying modules to be switched without requiring application rewritings. Moreover, multiple optimisation techniques have been designed, such as the selection bypass, the hybrid combiner and the vertex structure externalisation which leverage the clear distinction between the user interface and underlying implementations provided through vertex-centric abstractions. As observed the experiments of Chapters 4 and 5, the techniques implemented improve the handling of vertex-centric phases such as vertex selection and message combination, as well as tackling more general challenges such as load imbalance and memory locality. The results collected show that the *iPregel* framework successfully reaches the performance and memory efficiency of the fastest and most memory-efficient shared-memory frameworks tested. When compared against the then state-of-the-art distributed-memory vertex-centric framework *Pregel+*, *iPregel* demonstrates a memory footprint that is an order of magnitude smaller, while also ex-

hibiting a single-node performance that *Pregel+* can only match by using ten times the hardware resources. However, the most important aspect is that *iPregel* manages to achieve these results without having sacrificed vertex-centric abstractions, therefore verifying the first research direction.

The second research direction focussed on how the benefits obtained from shared-memory can be preserved while overcoming the memory limitations associated with a single node. To that end, non-volatile memory technology has been explored. Able to act as a filesystem, NVRAM can also behave as an extension to DRAM, thus allowing shared-memory frameworks to run within a single memory space containing terabytes of memory at low cost and with low power consumption. However, these benefits come at the expense of technology that is slower than DRAM, and provides asymmetric speeds for reads and writes. These characteristics make NVRAM particularly sensitive to data placement, as well as data movements between NVRAM and DRAM. The persistent-memory programming Software Development Kit (SDK) enables NVRAM to be used in multiple modes, for example acting as main memory while DRAM implicitly acts as the last-level cache, in which case underlying data movements and allocations are automatically handled, or behaving as a distinct medium requiring the use of explicit data manipulations. To evaluate the multiple memory modes available, several implementations of *iPregel* were developed, and experiments showed that careful placement of data and well-timed data movements, driven by insights gained from analysing the application's memory access patterns, are key to leveraging the performance of NVRAM.

Although non-volatile memory enables the *iPregel* framework to scale to a graph of 750 billion edges, shared-memory frameworks cannot benefit from the extra resources available on additional nodes, such as fast DRAM memory and compute power. In addition to performance, the results gathered also show that non-volatile memory is efficient in terms of energy and cost, consuming two orders of magnitude less energy and costing approximately half a million pounds less than using the equivalent amount of resources from the supercomputer used to process the biggest graph to date. Therefore, by successfully utilising non-volatile memory to enable the shared-memory vertex-centric framework *iPregel* to process a graph with nearly a trillion edges, this investigation verifies the second research direction.

The third research direction was the development of a design allowing the benefits of shared-memory *iPregel* to be leveraged over a distributed-memory architecture, in the scope of low to medium node counts. The research led to the development of a distributed-memory vertex-centric framework named *DiP*, which ports some of the techniques from *iPregel*, as well as exposing an interface consistent with that of *iPregel*. At its

core, the *DiP* framework implements a new buffer design that provides several benefits including a lighter message structure, an indexing mechanism not relying on hashing, an immediate combination upon queue which does not require prior sorting, a straightforward message dispatch not requiring intermediate copies or unwrapping. However, it is limited by the fact that its memory footprint is fixed, in order to provide the direct mapping property underlying most optimisations, and replicated across MPI processes. An intra-node buffer-sharing technique has been proposed in Subsection 7.4.2 to address the second drawback to this distributed approach. By enabling MPI processes on a given node to share buffers, this technique reduces the memory footprint replication basis from per MPI process to per node. To evaluate the competitiveness of the *DiP* framework against a shared-memory state-of-the-art, experiments have been carried out to compare against *iPregel*. Both push and pull versions of each framework were evaluated on sparse and dense graphs, across the three common vertex-centric benchmarks. Out of the twelve experiments, eight concluded that the *DiP* framework exhibited better single-node performance, although the key objective here is to enable the processing of larger graphs rather than provide greater performance.

Experiments were then carried out to investigate node scalability. The results show that the fewer the number of vertices, and thus the smaller the amount of data that is exchanged, the better the strong scaling properties of the *DiP* framework. By removing the memory footprint replication across MPI processes, the INBS technique enables nodes to host more MPI processes, which allows the parallelisation of MPI reductions to a greater extent. Weak scaling experiments highlight that the growth of *DiP* buffers eventually generates out-of-memory failures. The interval-based message processing technique proposed in Subsection 7.4.3 mitigates this issue by enabling the *DiP* framework to restrict the memory footprint of its buffers to a limit specified by the user and execute the processing phase in multiple rounds.

In terms of performance it has been demonstrated that the distributed-memory vertex-centric framework is competitive with its shared-memory counterpart. Furthermore, it was found that the memory footprint of the communication buffers, albeit replicated across MPI processes, typically accounts for less than 10% of the entire memory footprint per node due to the immediate combination of messages when they are queued up. This contrasts against existing distributed-memory frameworks, which are known to generate a noticeable memory overhead, where the message buffers account for up to 96% of the entire memory footprint. For example, as discussed in Section 4.1, where 264GB of DRAM was required by *Giraph* to process PageRank over a graph comprising nearly 2 billion edges [25], which can be stored in 8GB of memory. Despite the stress on network

bandwidth caused by large buffers, the scalability provided by *DiP* means that it is still able to process graphs of up to 1.6 trillion edges across a thousand cores, using approximately 6.8TB of DRAM in total, for a graph whose edges alone require 6.4TB of memory. Comparatively, at this scale, a framework imposing a memory overhead of 96% would require over 160TB of memory. Moreover, as shown in Subsection 7.6.4, the porting of *iPregel* optimisation techniques, the implementation of the buffer design and the development of the INBS and IBMP techniques successfully preserve vertex-centric programmability. Based on these findings, and the research conducted, therefore, the third research direction is verified.

To summarise, the main contributions made throughout this research, and whose relevance towards demonstrating the original research hypothesis was discussed further in this chapter, can be recapitulated as follows:

- Several techniques able to provide performance equal to, or competitive with, that of other non-vertex-centric frameworks, whilst maintaining a minimum memory footprint. The shared-memory framework developed in this research, *iPregel*, demonstrated this across experiments.

- An exploration of non-volatile memory in the context of shared-memory vertex-centric processing to overcome the limitation in-memory processing. Multiple data placement and data movement strategies were implemented in the *iPregel* framework to assess the suitability of each approach in maximising the asymmetric performance of read-writes, combined with the two-level DRAM-NVRAM memory system, with experiments conducted on graphs comprising between 250 and 750 billion edges.

- The porting of applicable techniques originally implemented in shared-memory, illustrated in *iPregel*, to distributed-memory and the introduction of a buffer design that reduces the algorithmic complexity of a full round of message generation from $O(n \times log(n))$ to $O(n)$ by exploiting the direct mapping property. These techniques and buffer design are demonstrated through the *DiP* distributed-memory framework.

- Two improvements to the memory footprint of the buffer design implemented in the *DiP* framework: an INBS technique relying on MPI-3 shared-memory and IBMP technique that enables *DiP* to adapt its size to a memory limit specified by the user.

- The fact that none of the techniques and methods presented above results in a

degradation of the vertex-centric programmability, thus maintaining the key benefit delivered by this approach.

Nonetheless, there are numerous further directions worthy of further exploration based upon the research reported in this thesis. As mentioned in Chapter 4, providing a built-in support for message concatenation as the combination operation, albeit counter-intuitive since combiners aim to merge messages, would improve the flexibility of both frameworks developed in this research. This would allow a larger class of algorithms to be implemented, which themselves would benefit from the techniques that have been developed and integrated with combiners. However, supporting message concatenation requires a careful design of the combiner in order to avoid requiring intermediate copies that may result in memory leaks unbeknown to the user such as implicit mailbox swaps occurring at the end of supersteps.

As discussed in Chapter 7, supporting user-defined combination operations in *DiP* would widen the flexibility that is provided to the user of the framework. However this feature is incompatible with the use of MPI one-sided routines that are leveraged for implementing the distributed buffer. This is because MPI Remote Memory Access (RMA), unlike MPI reduction calls, does not support user-defined operations.

A further approach worthy of exploration which could improve the handling of vertex communications is the incrementalisation of vertex-centric programs, presented in Subsection 3.3.8. This is where the exchange of messages is redesigned to communicate data representing inter-superstep variations. By sending deltas, representing only the changed values, this will potentially reduce the overall amount of communications required because unchanged values need not be communicated. Existing research [36] reports that the incrementalisation technique can be applied automatically to vertex-centric programs and, depending upon the specific graph processing algorithm, has the potential to reduce the number of messages exchanged. Consequently, such an approach would reduce the pressure placed on memory and/or network bandwidths.

More generally, as mentioned in Chapter 4, further investigations into load-balancing and work-stealing strategies could be beneficial to the performance of both frameworks developed in this research, and more widely to graph processing applications. As shown in Chapter 5, when switching from the default `static` OpenMP scheduling policy to `dynamic`, with an experimentally determined chunk size, this resulted in an average speedup of 1.50. However, techniques that better handle the distribution of workload across workers and potentially adjust these choices at runtime with work-stealing strategies are known to be particularly difficult to design in the context of distributed-memory parallelism and incur overhead that must be offset by the benefits gained.

Finally, the results reported in this thesis do not include the time required to load the graph processed, and this follows the standard way in which performance numbers are calculated by contemporary research in the field. However, such overheads can become significant when considering large files with a size typically beyond terabytes. Graph loading is a performance challenge which can be improved, for instance, by leveraging MPI parallel IO over dedicated parallel file systems or by exploring the possibility of pipelining the loading and computing phases, developing techniques allowing the first superstep to begin processing based on a graph partially loaded at any given point in time.

To prospective users, the compilation process for *iPregel* is straightforward. Benchmarks provided compile into multiple binaries: one for each available variation when applicable, namely: push and pull combiners, with and without selection bypass, using 32-bit and 64-bit vertex identifiers. For users to leverage optimisation techniques in their own applications, they only need to pass compilation flag defines given in Appendix L. The distributed memory counterpart *DiP* relies on the same approach. Certain flags are direct equivalent to those of *iPregel*, whereas others are specific to *DiP*, such as defines specific to MPI, to determine for instance the MPI reduction operation to use for the vertex-centric combination operation or the MPI datatype representing the datatype of values exchanged during reduction. Compilations flags for *DiP* are provided in Appendix M.

When considering the appropriate framework for a given application, several factors are to be included in the decision-making process. One such factor is the size of the graph in relation to the available memory. Situations where single-node processing is not applicable renders *iPregel* unsuitable. In other cases, it is worth noting that the selection bypass technique is currently only available in *iPregel*, which may make it a more versatile option in terms of the available optimisations and a more fitting starting point. However, as demonstrated in this thesis, the differences in implementation between *iPregel* and *DiP* are minimal, allowing for easy transfer of applications between the two frameworks. The *DiP* framework also offers a more fine-grain control over the datatypes used in the graphs to read.

As previously noted in this chapter, both frameworks are designed upon combiners, which can result in certain restrictions, such as limiting the types of combination operations that can be supported. For algorithms that require concatenation, implementation may become error-prone due to potential memory leaks resulting from pointer passing in the various combination copies that may be generated during the combination process. While this issue can be addressed in some cases in *iPregel* through the use of global vari-

ables, *DiP* does not offer this option due to its distributed memory structure. Ultimately, appreciating the different tradeoffs offered by the non-vertex-centric and vertex-centric solutions available, and determining the approach followed, remains the choice of the user.

Bibliography

[1] L. A. R. Capelli, Z. Hu, T. A. K. Zakian, iPregel: A Combiner-Based In-memory Shared Memory Vertex-Centric Framework, Proceedings of the 47th International Conference on Parallel Processing Companion - ICPP '18 (2018). doi:10.1145/3229710.3229719.
URL http://dx.doi.org/10.1145/3229710.3229719

[2] L. A. R. Capelli, Z. Hu, T. A. K. Zakian, N. Brown, J. M. Bull, iPregel: Vertex-centric programmability vs memory efficiency and performance, why choose?, Parallel Computing 86 (2019) 45 – 56. doi:10.1016/j.parco.2019.04.005.
URL http://www.sciencedirect.com/science/article/pii/S0167819118303788

[3] L. A. R. Capelli, N. Brown, J. M. Bull, iPregel: Strategies to Deal with an Extreme Form of Irregularity in Vertex-Centric Graph Processing, Proceedings of the The International Conference for High Performance Computing, Networking, Storage, and Analysis - SC '19 (2019). doi:10.1109/IA349570.2019.00013.

[4] L. A. R. Capelli, N. Brown, J. M. Bull, NVRAM as an Enabler to New Horizons in Graph Processing, SN Computer Science 3 (5) (2022) 1–13. doi:10.1007/s42979-022-01317-4.
URL https://doi.org/10.1007/s42979-022-01317-4

[5] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu, W. Zheng, J. Xu, ShenTu: Processing Multi-trillion Edge Graphs on Millions of Cores in Seconds, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, Piscataway, NJ, USA, 2018, pp. 56:1–56:11. doi:10.1109/SC.2018.00059.
URL https://doi.org/10.1109/SC.2018.00059

[6] A. Brighen, H. Slimani, A. Rezgui, H. Kheddouci, Listing all maximal cliques in large graphs on vertex-centric model, The Journal of Supercomputing 75 (8) (2019) 4918–4946.

[7] S. E. Schaeffer, Graph clustering, Computer science review 1 (1) (2007) 27–64.

[8] J. A. Bondy, U. S. R. Murty, et al., Graph theory with applications, Vol. 290, Macmillan London, 1976.

[9] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, M. Saltz, A Distributed Vertex-Centric Approach for Pattern Matching in Massive Graphs, in: 2013 IEEE International Conference on Big Data, IEEE, 2013, pp. 403–411.

[10] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, N. Mamoulis, SPARTex: A Vertex-centric Framework for RDF Data Analytics, Proc. VLDB Endow. 8 (12) (2015) 1880–1883. doi:10.14778/2824032.2824091.
URL http://dx.doi.org/10.14778/2824032.2824091

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A System for Large-scale Graph Processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146. doi:10.1145/1807167.1807184.
URL http://doi.acm.org/10.1145/1807167.1807184

[12] D. Yan, J. Cheng, Y. Lu, W. Ng, Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation, in: Proceedings of the 24th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee, 2015, pp. 1307–1317.

[13] F. Khorasani, K. Vora, R. Gupta, L. N. Bhuyan, CuSha: Vertex-centric Graph Processing on GPUs, in: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, ACM, New York, NY, USA, 2014, pp. 239–252. doi:10.1145/2600212.2600227.
URL http://doi.acm.org/10.1145/2600212.2600227

[14] F. Khorasani, High performance vertex-centric graph analytics on gpus, Ph.D. thesis, UC Riverside (2016).

[15] S. Che, GasCL: A vertex-centric graph model for GPUs, in: 2014 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2014, pp. 1–6.

[16] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, F. Yu, Seastar: vertex-centric programming for graph neural networks, in: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 359–375.

[17] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, C. Guestrin, GraphGen: An FPGA Framework for Vertex-Centric Graph Computation, in: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, 2014, pp. 25–28. doi:10.1109/FCCM.2014.15.

[18] N. Engelhardt, H. K.-H. So, GraVF: A vertex-centric distributed graph processing framework on FPGAs, in: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on, IEEE, 2016, pp. 1–4.

[19] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, L. Zhou, GraM: Scaling Graph Computation to the Trillions, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, ACM, New York, NY, USA, 2015, pp. 408–421. doi:10.1145/2806777.2806849.
URL http://doi.acm.org/10.1145/2806777.2806849

[20] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, H. Iwasaki, Think Like a Vertex, Behave Like a Function! A Functional DSL for Vertex-centric Big Graph Processing, in: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, ACM, New York, NY, USA, 2016, pp. 200–213. doi:10.1145/2951913.2951938.
URL http://doi.acm.org/10.1145/2951913.2951938

[21] Y. Zhang, H.-S. Ko, Z. Hu, Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access, in: Asian Symposium on Programming Languages and Systems, Springer, 2017, pp. 301–320.

[22] O. Coll Ruiz, K. Matsuzaki, S. Sato, S6Raph: Vertex-centric Graph Processing Framework with Functional Interface, in: Proceedings of the 5th International Workshop on Functional High-Performance Computing, FHPC 2016, ACM, New York, NY, USA, 2016, pp. 58–64. doi:10.1145/2975991.2976000.
URL http://doi.acm.org/10.1145/2975991.2976000

[23] M. Zhou, M. Imani, S. Gupta, Y. Kim, T. Rosing, GRAM: Graph Processing in a ReRAM-based Computational Memory, in: Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC '19, ACM, New York, NY, USA,

2019, pp. 591–596. doi:10.1145/3287624.3287711.
URL http://doi.acm.org/10.1145/3287624.3287711

[24] A. Kyrola, G. Blelloch, C. Guestrin, GraphChi: Large-scale Graph Computation on Just a PC, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 31–46.
URL http://dl.acm.org/citation.cfm?id=2387880.2387884

[25] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, C. Zhang, GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit, IEEE Transactions on Parallel and Distributed Systems (2017).

[26] S.-W. Jun, A. Wright, S. Zhang, S. Xu, et al., GraFBoost: Using accelerated flash storage for external graph analytics, in: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2018, pp. 411–424.

[27] C. Martella, D. Logothetis, A. Loukas, G. Siganos, Spinner: Scalable Graph Partitioning in the Cloud, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 1083–1094. doi:10.1109/ICDE.2017.153.

[28] C. Zhou, J. Gao, B. Sun, J. X. Yu, Mocgraph: Scalable distributed graph processing using message online computing, Proceedings of the VLDB Endowment 8 (4) (2014) 377–388.

[29] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, C. He, VENUS: Vertex-centric streamlined graph computation on a single PC, in: 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 1131–1142. doi:10.1109/ICDE.2015.7113362.

[30] Q. Liu, Z. Li, J. C. Lui, J. Cheng, Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition, in: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, 2016, pp. 195–204.

[31] E. Nurvitadhi, D. Marr, Heterogeneous hardware accelerator architecture for processing sparse matrix data with skewed non-zero distributions (Jan 2019).

[32] N. T. Bao, T. Suzumura, Towards Highly Scalable Pregel-based Graph Processing Platform with X10, in: Proceedings of the 22nd International Conference on World Wide Web, 2013, pp. 501–508.

[33] I. Hoque, I. Gupta, LFGraph: Simple and fast distributed graph analytics, in: Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, 2013, pp. 1–17.

[34] J. Yan, G. Tan, N. Sun, GRE: A graph runtime engine for large-scale distributed graph-parallel applications, arXiv preprint arXiv:1310.5603 (2013).

[35] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727. doi:10.14778/2212351.2212354. URL https://doi.org/10.14778/2212351.2212354

[36] T. A. Zakian, L. A. Capelli, Z. Hu, Incrementalization of Vertex-Centric Programs, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 1019–1029.

[37] J. Li, Y. Cao, Y. Zhang, M. Z. A. Bhuiyan, B. Li, SPFC: An Effective Optimization for Vertex-Centric Graph Processing Systems, IEEE Transactions on Sustainable Computing 4 (1) (2019) 118–131. doi:10.1109/TSUSC.2017.2780320.

[38] B. Alex, W. Benjamin, R. Ioan, FemtoGraph: A Pregel Based Shared-memory Graph Processing Library, poster at SC'16.

[39] H. Sutter, et al., The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's journal 30 (3) (2005) 202–210.

[40] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, A. R. LeBlanc, Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions, IEEE Journal of solid-state circuits 9 (5) (1974) 256–268.

[41] E. Strohmaier, H. W. Meuer, J. Dongarra, H. D. Simon, The TOP500 list and progress in high-performance computing, Computer 48 (11) (2015) 42–49.

[42] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, D. M. Tullsen, Simultaneous Multithreading: A Platform for Next-Generation Processors, IEEE micro 17 (5) (1997) 12–19.

[43] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, M. Roth, Challenges of memory management on modern NUMA systems, Communications of the ACM 58 (12) (2015) 59–66.

[44] Persistent memory documentation, https://docs.pmem.io/persistent-memory/getting-started-guide/introduction, retrieved: 2022-10-27.

[45] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Computational Science and Engineering 5 (1) (1998) 46–55. doi:10.1109/99.660313.

[46] M. P. Forum, MPI: A Message-Passing Interface Standard, Tech. rep., Knoxville, TN, USA (1994).

[47] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, 1967, pp. 483–485.

[48] R. Benner, J. Gustafson, G. Montry, Development and analysis of scientific application programs on a 1024-processor hypercube, SAND 88-0317, Sandia National Laboratories (1988).

[49] L. G. Valiant, A Bridging Model for Parallel Computation, Communications of the ACM 33 (8) (1990) 103–111.

[50] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer networks and ISDN systems 30 (1-7) (1998) 107–117.

[51] Y. Han, R. A. Wagner, An efficient and fast parallel-connected component algorithm, Journal of the ACM (JACM) 37 (3) (1990) 626–642.

[52] R. K. Ahuja, K. Mehlhorn, J. Orlin, R. E. Tarjan, Faster algorithms for the shortest path problem, Journal of the ACM (JACM) 37 (2) (1990) 213–223.

[53] J. Shun, G. E. Blelloch, Ligra, Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13 (2013). doi:10.1145/2442516.2442530.
URL http://dx.doi.org/10.1145/2442516.2442530

[54] R. R. McCune, T. Weninger, G. Madey, Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing, ACM Computing Surveys 48 (2) (2015) 1–39. doi:10.1145/2818185.
URL http://dx.doi.org/10.1145/2818185

[55] X. Zhu, W. Chen, W. Zheng, X. Ma, Gemini: A computation-centric distributed graph processing system, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 301–316.

[56] D. Yan, J. Cheng, Y. Lu, W. Ng, Blogel: A block-centric framework for distributed computation on real-world graphs, Proceedings of the VLDB Endowment 7 (14) (2014) 1981–1992.

[57] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, V. Prasanna, Goffish: A sub-graph centric framework for large-scale graph analytics, in: European Conference on Parallel Processing, Springer, 2014, pp. 451–462.

[58] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, J. McPherson, From "Think Like a Vertex" to "Think Like a Graph", Proc. VLDB Endow. 7 (3) (2013) 193–204. doi:10.14778/2732232.2732238.
URL http://dx.doi.org/10.14778/2732232.2732238

[59] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, in: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX, Hollywood, CA, 2012, pp. 17–30.
URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[60] S. Salihoglu, J. Widom, Gps: A graph processing system, in: Proceedings of the 25th international conference on scientific and statistical database management, 2013, pp. 1–12.

[61] Y. Zhang, Z. Hu, Composing Optimization Techniques for Vertex-Centric Graph Processing via Communication Channels, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 428–438.

[62] Y. Shiloach, U. Vishkin, An O(log n) parallel connectivity algorithm, Tech. rep., Computer Science Department, Technion (1980).

[63] S. Gong, C. Tian, Q. Yin, W. Yu, Y. Zhang, L. Geng, S. Yu, G. Yu, J. Zhou, Automating incremental graph processing with flexible memoization, Proceedings of the VLDB Endowment 14 (9) (2021) 1613–1625.

[64] L. Quick, P. Wilkinson, D. Hardcastle, Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis, in: Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012), ASONAM '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 457–463. doi:10.1109/ASONAM.2012.254.
URL http://dx.doi.org/10.1109/ASONAM.2012.254

[65] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, N. Mamoulis, Spartex: A vertex-centric framework for RDF data analytics, Proceedings of the VLDB Endowment 8 (12) (2015) 1880–1883.

[66] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, Combining Vertex-Centric Graph Processing with SPARQL for Large-Scale RDF Data Analytics, IEEE Transactions on Parallel and Distributed Systems 28 (12) (2017) 3374–3388. doi:10.1109/TPDS.2017.2720174.

[67] V. Kalavri, V. Vlassov, S. Haridi, High-Level Programming Abstractions for Distributed Graph Processing (07 2016). arXiv:1607.02646.
URL https://arxiv.org/abs/1607.02646

[68] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One Trillion Edges: Graph Processing at Facebook-Scale, Proceedings of the VLDB Endowment 8 (12) (2015) 1804–1815.

[69] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein, Graphlab: A new framework for parallel machine learning. arXiv preprint, arXiv preprint arXiv:1006.4990 1 (2010).

[70] Y. Bu, Pregelix: Dataflow-based Big Graph Analytics, in: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, ACM, New York, NY, USA, 2013, pp. 54:1–54:2. doi:10.1145/2523616.2525962.
URL http://doi.acm.org/10.1145/2523616.2525962

[71] A. Khan, Vertex-Centric Graph Processing: The Good, the Bad, and the Ugly (12 2016). arXiv:1612.07404.
URL https://arxiv.org/abs/1612.07404

[72] S. Liu, A. Khan, An empirical analysis on expressibility of vertex centric graph processing paradigm, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 242–251.

[73] M. Han, On Improving Distributed Pregel-like Graph Processing Systems, Master's thesis, University of Waterloo (2015).

[74] X. Hu, An experimental evaluation of vertex-centric k-core decomposition using giraph and graphchi (2017).

[75] J. Kunegis, KONECT: The Koblenz Network Collection, in: Proceedings of the 22Nd International Conference on World Wide Web, WWW '13 Companion, ACM, New York, NY, USA, 2013, pp. 1343–1350. doi:10.1145/2487788.2488173. URL http://doi.acm.org/10.1145/2487788.2488173

[76] T. C. for Discrete Mathematics, T. C. S. (DIMACS), 9th DIMACS Implementation Challenge, http://www.dis.uniroma1.it/challenge9/download.shtml (2006).

[77] J. Leskovec, A. Krevl, SNAP Datasets: Stanford Large Network Dataset Collection, http://snap.stanford.edu/data (Jun. 2014).

[78] A. Roy, I. Mihailovic, W. Zwaenepoel, X-Stream: Edge-centric Graph Processing Using Streaming Partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, ACM, New York, NY, USA, 2013, pp. 472–488. doi:10.1145/2517349.2522740. URL http://doi.acm.org/10.1145/2517349.2522740

[79] I. Hadade, T. M. Jones, F. Wang, L. di Mare, Software prefetching for unstructured mesh applications, in: 2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3), IEEE, 2018, pp. 11–19.

[80] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, S. Swanson, Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing, in: SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–11. doi:10.1109/SC.2010.56.

[81] Intel Announces Broadest Product Portfolio for Moving, Storing and Processing Data, https://newsroom.intel.com/news-releases/intel-data-centric-launch/#gs.no8yic, retrieved: 2022-10-27.

[82] M. Weiland, Evaluation of Intel Optane DCPMM for memory and I/O intensive HPC applications, https://www.ixpug.org/resources/download/

micheleweiland-hpcasia2020, iXPUG Workshop at HPC Asia 2020. Retrieved: 2022-10-27 (2020).

[83] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, M. Parsons, An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–19.

[84] G. Gill, R. Dathathri, L. Hoang, R. Peri, K. Pingali, Single machine graph analytics on massive datasets using intel optane DC persistent memory, arXiv preprint arXiv:1904.07162 (2019).

[85] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, J. Shun, Semi-Asymmetric Parallel Graph Algorithms for NVRAMs, arXiv preprint arXiv:1910.12310 (2019).

[86] Persistent Memory Development Kit, https://pmem.io/pmdk/, retrieved: 2022-10-27.

[87] Gong Zhe, Sunway Taihulight: Things you may not know about China's supercomputer, https://news.cgtn.com/news/3d517a4d324d444e/share_p.html (2017).

[88] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng, Hama: An efficient matrix computation with the mapreduce framework, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, IEEE, 2010, pp. 721–726.

[89] J. Stern, Q. Xiong, A. Skjellum, M. Herbordt, A novel approach to supporting communicators for in-switch processing of MPI collectives, in: Workshop on Exascale MPI, 2018.

[90] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, D. K. Panda, BluesMPI: Efficient MPI Non-Blocking Alltoall Offloading Designs on Modern BlueField Smart NICs, in: International Conference on High Performance Computing, Springer, 2021, pp. 18–37.

[91] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, W. Pan, GraphHP: A hybrid platform for iterative graph processing, arXiv preprint arXiv:1706.07221 (2017).

[92] C. Simmendinger, M. Rahn, D. Gruenewald, The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures, in: Sustained Simulation Performance 2014, Springer, 2015, pp. 17–32.

Appendices

Appendix A

Supporting functions in iPregel

```
/**
 * @brief This function returns the current superstep.
 * @details The superstep is 0-indexed, meaning that the first superstep is
 * the superstep 0.
 * @return The current superstep.
 **/
size_t ip_get_superstep(void);

/**
 * @brief This function consumes the next message in the mailbox of the
 * specified vertex.
 * @details This function consumes the next message and stores it in the
 * message pointed by the pointer provided. Prior to consuming the message,
 * this function checks if the vertex specified has at least one message to
 * read. The result of this check is returned by this function, which
 * allows for this function to be used in a loop condition statement.
 * @param[inout] me The vertex whose mailbox is to be fetched.
 * @param[out] message The variable in which to store the message consumed.
 * @return The result indicating whether a fetch took place.
 * @retval true The vertex specified had a message to read; this message
 * has been consumed and is stored in the message variable passed.
 * @retval false The vertex specified did not have a message left to read.
 * The message variable passed is left unmodified.
 * @pre The vertex pointer given points to an allocated memory area
 * containing a vertex.
 * @pre The message pointer given points to an allocated memory area
 * containing a message.
 **/
bool ip_get_next_message(struct ip_vertex_t* me, IP_MESSAGE_TYPE* message);

/**
 * @brief This function sends the message given to the vertex identified
 * by the vertex identifier specified.
 * @param[in] id The identifier of the destination vertex.
 * @param[in] message The message to send.
 * @pre id is an existing vertex identifier.
 * @post The message is delivered to the destination vertex, and combined
 * with any pre-existing message on that recipient vertex if any.
 **/
```

```
40 void ip_send_message(IP_VERTEX_ID_TYPE recipient_vertex_id,
41                      IP_MESSAGE_TYPE message);
42
43 /**
44  * @brief This function sends the message specified to all neighbours of
45  * the emitting vertex.
46  * @param[out] me The emitting vertex.
47  * @param[in] message The message to broadcast.
48  * @pre The vertex pointer given points to an allocated memory area
49  * containing a vertex.
50  * @post All neighbours of the emitting vertex will have received the
51  * message given before next superstep. Note that it may be combined
52  * during the process.
53  **/
54 void ip_broadcast(struct ip_vertex_t* me, IP_MESSAGE_TYPE message);
55
56 /**
57  * @brief This function halts the vertex specified.
58  * @details This function is safe to call on a vertex already inactive.
59  * @param[out] me The vertex to halt.
60  * @pre The vertex pointer given points to an allocated memory area
61  * containing a vertex.
62  * @post The vertex specified is inactive.
63  **/
64 void ip_vote_to_halt(struct ip_vertex_t* me);
```

Appendix B

Additional user-defined structures needed by the PageRank compute function in Ligra.

```
1  template <class vertex> struct PR_F {
2      double* p_curr;
3      double* p_next;
4      vertex* V;
5      PR_F(double* _p_curr, double* _p_next,vertex* _V):
6          p_curr(_p_curr),
7          p_next(_p_next),
8          V(_V) {
9      }
10     inline bool update(uintE s, uintE d) {
11         p_next[d] += p_curr[s] / V[s].getOutDegree();
12         return 1;
13     }
14     inline bool updateAtomic(uintE s, uintE d) {
15         writeAdd(&p_next[d], p_curr[s]/V[s].getOutDegree());
16         return 1;
17     }
18     inline bool cond(intT d) {
19         return cond_true(d);
20     }
21 };
22 struct PR_Vertex_F {
23     double damping;
24     double addedConstant;
25     double* p_curr;
26     double* p_next;
27     PR_Vertex_F(double* _p_curr, double* _p_next, double _damping, intE n):
28         p_curr(_p_curr),
29         p_next(_p_next),
30         damping(_damping),
31         addedConstant((1 _damping) *(1 / (double)n)) {
32     }
33     inline bool operator () (uintE i) {
34         p_next[i] = damping * p_next[i] + addedConstant;
```

```
35          return 1;
36      }
37  };
38  struct PR_Vertex_Reset {
39      double* p_curr;
40      PR_Vertex_Reset(double* _p_curr):
41          p_curr(_p_curr) {
42      }
43      inline bool operator () (uintE i) {
44          p_curr[i] = 0.0;
45          return 1;
46      }
47  };
```

Appendix C

Implementation of PageRank in Blogel, using the vertex mode

```
struct PRValue_pregel {
    double pr;
    vector<VertexID> edges;
};

class PRVertex_pregel: public Vertex<VertexID, PRValue_pregel, double> {
    public: virtual void compute(MessageContainer & messages) {
        if(step_num() == 1) {
            value().pr = 1.0 / get_vnum();
        }
        else {
            double sum = 0;
            for(MessageIter it = messages.begin(); it != messages.end();
                it++) {
                sum += *it;
            }
            double* agg = (double*)getAgg();
            double residual = *agg/get_vnum();
            value().pr = 0.15 / get_vnum() + 0.85 * (sum+residual);
        }
        if(step_num() < ROUND) {
            double msg = value().pr / value().edges.size();
            for(auto it = value().edges.begin(); it != value().edges.end();
                it++) {
                send_message(*it, msg);
            }
        }
        else {
            vote_to_halt();
        }
    }
};

class PRAgg_pregel: public Aggregator<PRVertex_pregel, double, double> {
    private: double sum;
    public: virtual void init(){
        sum = 0;
```

```
38      }
39
40      public: virtual void stepPartial(PRVertex_pregel* v) {
41          if(v->value().edges.size() == 0) {
42              sum+=v->value().pr;
43          }
44      }
45
46      public: virtual void stepFinal(double* part) {
47          sum += *part;
48      }
49
50      public: virtual double* finishPartial() {
51          return &sum;
52      }
53
54      public: virtual double* finishFinal() {
55          return &sum;
56      }
57  };
```

Appendix D

Implementation of PageRank in Blogel, using the block mode

```
#define EPS 0.01

struct PRValue {
    double pr;
    vector<triplet> edges;
    int split;
};

class PRVertex: public BVertex<VertexID, PRValue, char> {
    public: virtual void compute(MessageContainer& msgs) {
    }
};

struct doublepair {
    bool converge;
    double accum;
};

struct tuple {
    int block;
    double weight;
    int worker;
};

struct PRBlockValue {
    double pr;
    double delta;
    vector<tuple> edges;
};

class PRBlock: public Block<PRBlockValue, PRVertex, double> {
    public: virtual void compute(MessageContainer& msgs,
                                 VertexContainer& vertexes) {
        if(step_num() == 1) {
            value().pr = 1.0 / get_bnum();
            value().delta = EPS / get_bnum() + 1;
        }
```

```
38          else {
39              double sum = 0;
40              for(MessageIter it = msgs.begin(); it != msgs.end(); it++) {
41                  sum += *it;
42              }
43              doublepair* agg = ((doublepair*)getAgg());
44              if(agg->converge) {
45                  vote_to_halt();
46                  return;
47              }
48              double residual = agg->accum / get_bnum();
49              double newVal = 0.15 / get_bnum() + 0.85 * (sum + residual);
50              value().delta = fabs(newVal - value().pr);
51              value().pr = newVal;
52          }
53          for(auto it = value().edges.begin(); it != value().edges.end();
54              it++) {
55              send_message(it->block, it->worker, it->weight * value().pr);
56          }
57      }
58  };
59
60  class PRSum : public BAggregator<PRVertex, PRBlock, doublepair,
61                                   doublepair> {
62      private: doublepair pair;
63
64      public: virtual void init() {
65          pair.converge = true;
66          pair.accum = 0;
67      }
68
69      public: virtual void stepPartialV(PRVertex* v) {};
70
71      public: virtual void stepPartialB(PRBlock* b) {
72          if(b->value().edges.size() == 0) {
73              pair.accum += b->value().pr;
74          }
75          if(b->value().delta > EPS / get_bnum()) {
76              pair.converge = false;
77          }
78      }
79
80      public: virtual void stepFinal(doublepair* part) {
81          pair.accum += part->accum;
82          if(part->converge == false) {
83              pair.converge = false;
84          }
85      }
86
87      public: virtual doublepair* finishPartial() {
88          return &pair;
89      }
90
91      public: virtual doublepair* finishFinal() {
92          return &pair;
```

```
93      }
94  };
95
96  class PRWorker : public BWorker<PRBlock, PRSum> {
97      public: void localPR(PRBlock* block, VertexContainer& vertexes) {
98          bool converge = false;
99          double accum = 0;
100         int num = block->size;
101         double* pr_buf;
102         int round = 1;
103         double threshold = EPS / num;
104         while(converge == false) {
105             double oldaccum;
106             double* old_pr_buf;
107             if(round > 1) {
108                 converge = true;
109                 oldaccum = accum;
110                 accum = 0;
111                 old_pr_buf = pr_buf;
112             }
113             pr_buf = new double[num];
114             for(int i = 0; i < num; i++) {
115                 pr_buf[i] = 0;
116             }
117             for(int i = block->begin; i < block->begin + block->size;
118                 i++) {
119                 PRVertex* vertex = vertexes[i];
120                 if(round == 1) {
121                     vertex->value().pr = 1.0 / num;
122                 }
123                 else {
124                     int logID = i - block->begin;
125                     double impact = old_pr_buf[logID] + oldaccum / num;
126                     double new_pr = 0.15 / num + 0.85 * impact;
127                     double delta = new_pr - vertex->value().pr;
128                     if(fabs(delta) > threshold) {
129                         converge = false;
130                     }
131                     vertex->value().pr = new_pr;
132                 }
133                 vector<triplet>& edges = vertex->value().edges;
134                 int split = vertex->value().split;
135                 if(split == -1) {
136                     accum += vertex->value().pr;
137                 }
138                 else {
139                     double msg = vertex->value().pr / (split + 1);
140                     for(int j = 0; j <= split; j++) {
141                         triplet nb = edges[j];
142                         int phyID = nb.wid;
143                         int logID = phyID - block->begin;
144                         pr_buf[logID] += msg;
145                     }
146                 }
147             }
```

```
148             if(round > 1) {
149                 delete old_pr_buf;
150             }
151             round++;
152         }
153         delete pr_buf;
154     }
155 
156     virtual void blockInit(VertexContainer& vertexes,
157                            BlockContainer& blocks) {
158         ResetTimer(4);
159         hash_map<int, int> map;
160         for(int i = 0; i < vertexes.size(); i++) {
161             map[vertexes[i]->id] = i;
162         }
163         for(BlockIter it = blocks.begin(); it != blocks.end(); it++) {
164             PRBlock* block = *it;
165             for(int i = block->begin; i < block->begin + block->size;
166                 i++) {
167                 PRVertex* vertex = vertexes[i];
168                 vector<triplet>& edges = vertex->value().edges;
169                 vector<triplet> tmp;
170                 vector<triplet> tmp1;
171                 for(int j = 0; j < edges.size(); j++) {
172                     if (edges[j].bid == block->bid) {
173                         edges[j].wid = map[edges[j].vid];
174                         tmp.push_back(edges[j]);
175                     }
176                     else {
177                         tmp1.push_back(edges[j]);
178                     }
179                 }
180                 edges.swap(tmp);
181                 vertex->value().split = edges.size() - 1;
182                 edges.insert(edges.end(), tmp1.begin(), tmp1.end());
183             }
184         }
185         for(BlockIter it = blocks.begin(); it != blocks.end(); it++) {
186             PRBlock* block = *it;
187             localPR(block, vertexes);
188         }
189         for(BlockIter it = blocks.begin(); it != blocks.end(); it++) {
190             PRBlock* block = *it;
191             for(int i = block->begin; i < block->begin + block->size;
192                 i++) {
193                 PRVertex* vertex = vertexes[i];
194                 vector<triplet>& edges = vertex->value().edges;
195                 int split = vertex->value().split;
196                 for(int j = 0; j <= split; j++) {
197                     edges[j].wid = _my_rank;
198                 }
199             }
200         }
201         for(BlockIter it = blocks.begin(); it != blocks.end(); it++) {
202             PRBlock* block = *it;
```

```
203             hash_map<int, tuple> bmap;
204             for(int i = block->begin; i < block->begin + block->size;
205                 i++) {
206                 PRVertex* vertex = vertexes[i];
207                 vector<triplet>& vedges = vertex->value().edges;
208                 int degree = vedges.size();
209                 hash_map<int, tuple> count;
210                 for(int j = 0; j < degree; j++) {
211                     int blockID = vedges[j].bid;
212                     int workerID = vedges[j].wid;
213                     hash_map<int, tuple>::iterator cit = count.find(blockID);
214                     if(cit == count.end()) {
215                         tuple cur = { blockID, 1, workerID };
216                         count[blockID] = cur;
217                     }
218                     else {
219                         cit->second.weight++;
220                     }
221                 }
222                 for(auto cit = count.begin(); cit != count.end(); cit++) {
223                     int blockID = cit->first;
224                     double cnt = cit->second.weight;
225                     int workerID = cit->second.worker;
226                     double val = vertex->value().pr * cnt / degree;
227                     hash_map<int, tuple>::iterator bit =
228                         bmap.find(blockID);
229                     if(bit == bmap.end()) {
230                         tuple cur = { blockID, val, workerID };
231                         bmap[blockID] = cur;
232                     }
233                     else {
234                         bit->second.weight += val;
235                     }
236                 }
237             }
238             vector<tuple>& adj_list = block->value().edges;
239             double wsum = 0;
240             for(auto bit = bmap.begin(); bit != bmap.end(); bit++) {
241                 adj_list.push_back(bit->second);
242                 wsum += bit->second.weight;
243             }
244             for(int i = 0; i < adj_list.size(); i++) {
245                 adj_list[i].weight /= wsum;
246             }
247         }
248     }
249 };
250
251 class PRCombiner : public Combiner<double> {
252     public: virtual void combine(double& old, const double& new_msg) {
253         old += new_msg;
254     }
255 };
```

Appendix E

Pseudo-code of the implementation of the Connected Components benchmark in Giraph++

```
1  begin
2      if getSuperstep() = 0 then
3          sequentialCC()
4          foreach bv IN boundaryVertices() do
5              sendMsg(bv.getVertexId(), bv.getVertexValue())
6      else
7          equiCC={}
8          foreach iv IN activeInternalVertices() do
9              minValue = min(iv.getMessages())
10             if minValue < iv.getVertexValue() then
11                 equiCC.add(iv.getVertexValue(), minValue)
12         equiCC.consolidate()
13         foreach iv IN internalVertices() do
14             changedTo=equiCC.uniqueLabel(iv.getVertexValue())
15             iv.setVertexValue(changedTo)
16         foreach bv IN boundaryVertices() do
17             changedTo=equiCC.uniqueLabel(bv.getVertexValue())
18             if changedTo!=bv.getVertexValue() then
19                 bv.setVertexValue(changedTo)
20                 sendMsg(bv.getVertexId(), bv.getVertexValue())
21     allVoteToHalt()
```

Appendix F

Implementation of the Shiloach-Vishkin algorithm in channel-based Pregel system

```
class SVWorker: public Worker<int> {
private:
  int phase;
  vector<int> P, GP, MNP, Dup;

  ScatterCombine<int, int> MSG_1;
  PushCombine<int, int> MSG_2;
  RequestRespond<int, int> REQ;

  Combiner<int> combiner;

public:
  SVWorker(): phase(1), REQ(this),
      MSG_1(this, make_combiner(c_min, INT_MAX)),
      MSG_2(this, make_combiner(c_min, INT_MAX)),
      combiner(make_combiner(c_min, INT_MAX)) {}

  ~SVWorker() {
  }

  void load_channels(const EdgeBuffer &es) override {
    P.resize(numv());
    MNP.resize(numv());
    Dup.resize(numv());
    MSG_1.load(es);
  }

  bool compute() override {
    if(phase == 1) {
      for(int u = 0; u < numv(); u++) {
        P[u] = MNP[u] = get_id(u);
      }
      REQ.add_requests(P);
      phase = 2;
```

```
35        return false;
36      }
37      if(phase == 2) {
38        Dup = P;
39        REQ.respond(P);
40        MSG_1.scatter(P);
41        phase = 3;
42        return false;
43      }
44      if(phase == 3) {
45        copyFrom(GP, REQ);
46        for(int u = 0; u < numv(); u++) {
47          MNP[u] = min(MNP[u], MSG_1.get_message(u));
48          if(GP[u] == P[u]) {
49            if(MNP[u] < P[u]) {
50              MSG_2.add_message(P[u], MNP[u]);
51            }
52          }
53          else {
54            P[u] = GP[u];
55          }
56        }
57        GP = P;
58        MSG_2.activate();
59        phase = 4;
60        return false;
61      }
62      if(phase == 4) {
63        combineFrom(P, MSG_2, combiner);
64        REQ.add_requests(P);
65        phase = 2;
66        return (P == Dup);
67      }
68      return true;
69    }
70
71    void output() {
72      int sum = 0;
73      for(int u = 0; u < numv(); u++) {
74        if (P[u] == get_id(u)) {
75          sum += 1;
76        }
77      }
78      int num_cc = all_sum(sum);
79      if(get_rank() = 0) {
80        printf("number␣of␣CCs:␣%d\n", num_cc);
81      }
82    }
83  };
```

Appendix G

Expected and measured aggregated speedups

Benchmark	Graph	Expected	Measured
PR	DBLP	1.61	1.61
	LiveJournal	2.97	3.14
	Orkut	3.01	3.07
	Friendster	1.63	1.63
CC	DBLP	1.94	2.05
	LiveJournal	2.77	2.96
	Orkut	2.48	2.41
	Friendster	1.98	2.12
SSSP	DBLP	1.21	1.09
	LiveJournal	1.50	1.75
	Orkut	3.90	3.18
	Friendster	7.56	5.73

Table G.1: Expected and measured aggregated speedups, across all three benchmarks and all four graphs.

Appendix H

Main functions, not all, provided by the DiP framework

```
/**
 * @brief This function returns the current superstep.
 * @details The superstep is 0-indexed, meaning that the first superstep is
 * the superstep 0.
 * @return The current superstep.
 **/
unsigned long int dip_get_superstep(void);

/**
 * @brief This function consumes the next message in the mailbox of the
 * specified vertex.
 * @details This function consumes the next message and stores it in the
 * message pointed by the pointer provided. Prior to consuming the message,
 * this function checks if the vertex specified has at least one message to
 * read. The result of this check is returned by this function, which
 * allows for this function to be used in a loop condition statement.
 * @param[inout] me The vertex whose mailbox is to be fetched.
 * @param[out] message The variable in which to store the message consumed.
 * @return The result indicating whether a fetch took place.
 * @retval true The vertex specified had a message to read; this message
 * has been consumed and is stored in the message variable passed.
 * @retval false The vertex specified did not have a message left to read.
 * The message variable passed is left unmodified.
 * @pre The vertex pointer given points to an allocated memory area
 * containing a vertex.
 * @pre The message pointer given points to an allocated memory area
 * containing a message.
 **/
bool dip_get_next_message(dip_vertex* me, DIP_MESSAGE_TYPE* message);

/**
 * @brief This function sends the message given to the vertex identified
 * by the vertex identifier specified.
 * @param[in] id The identifier of the destination vertex.
 * @param[in] message The message to send.
 * @pre id is an existing vertex identifier.
 * @post The message is delivered to the destination vertex, and combined
```

```
 * with any pre-existing message on that recipient vertex if any.
 **/
void dip_send(DIP_VERTEX_ID_TYPE id, DIP_MESSAGE_TYPE message);

/**
 * @brief This function sends the message specified to all neighbours of
 * the emitting vertex.
 * @param[out] me The emitting vertex.
 * @param[in] message The message to broadcast.
 * @pre The vertex pointer given points to an allocated memory area
 * containing a vertex.
 * @post All neighbours of the emitting vertex will have received the
 * message given before next superstep. Note that it may be combined
 * during the process.
 **/
void dip_broadcast(dip_vertex* me, DIP_MESSAGE_TYPE message);

/**
 * @brief This function halts the vertex specified.
 * @details This function is safe to call on a vertex already inactive.
 * @param[out] me The vertex to halt.
 * @pre The vertex pointer given points to an allocated memory area
 * containing a vertex.
 * @post The vertex specified is inactive.
 **/
void dip_halt(dip_vertex* me);
```

Appendix I

Implementation of PageRank in the DiP framework

```
1  void dip_compute(dip_vertex* me) {
2      if(dip_get_superstep() == 0) {
3          me->value = 1.0 / dip_get_vertex_count();
4      } else {
5          DIP_MESSAGE_TYPE sum = 0.0;
6          if(dip_has_message(me)) {
7              dip_get_next_message(me, &sum);
8          }
9          me->value = ratio + 0.85 * sum;
10     }
11
12     if(dip_get_superstep() < 10) {
13         if(me->neighbours_count > 0) {
14             dip_broadcast(me, me->value / me->neighbours_count);
15         }
16     }
17     else {
18         dip_halt(me);
19     }
20 }
```

Appendix J

Implementation of Connected Components in the DiP framework

```
1  void dip_compute(dip_vertex* me) {
2      if(dip_get_superstep() == 0) {
3          me->value = me->id;
4          dip_broadcast(me, me->value);
5      }
6      else {
7          DIP_MESSAGE_TYPE valueTemp = me->value;
8          DIP_MESSAGE_TYPE message_value;
9          while(dip_get_next_message(me, &message_value)) {
10             if(me->value > message_value) {
11                 me->value = message_value;
12             }
13         }
14         if(valueTemp != me->value) {
15             dip_broadcast(me, me->value);
16         }
17     }
18     dip_halt(me);
19 }
```

Appendix K

Implementation of unweighted Single-Source Shortest Paths in the DiP framework

```
#define START_VERTEX 0

void dip_compute(dip_vertex* me) {
    if(dip_get_superstep() == 0) {
        if(me->id == START_VERTEX) {
            me->value = 0;
            dip_broadcast(me, me->value + 1);
        }
        else {
            me->value = UINT_MAX;
        }
    }
    else {
        DIP_MESSAGE_TYPE m_initial = UINT_MAX;
        DIP_MESSAGE_TYPE m;
        while(dip_get_next_message(me, &m)) {
            if(m_initial > m) {
                m_initial = m;
            }
        }
        if(m_initial < me->value) {
            me->value = m_initial;
            dip_broadcast(me, m_initial + 1);
        }
    }

    dip_halt(me);
}
```

Appendix L

iPregel compilation flags

The following flags are used to enable certain defines, so they must be prefixed with `-D`.

IP_USE_SPREAD Passing this flag enables the selection bypass technique, which was introduced in Subsubsection 4.3.4.1.

IP_USE_SINGLE_BROADCAST Indicates which combiner use, introduced in Subsubsection 4.3.4.3. Specifying this flag enables the use of the pull-based combiner, otherwise uses the push-based combiner.

IP_VERTEX_ID_TYPE Specifies the datatype of a vertex identifier in the graph to load.
Example values: `uint32_t`, `uint64_t`...
Example: `-DIP_VERTEX_ID_TYPE=uint32_t`

IP_ID_OFFSET Specifies the lowest vertex identifier. This is used to apply an offset when addressing vertices. If the lowest vertex identifier is 0, this flag has no consequence.
Example values: 0, 40, 125...
Example: `-DIP_ID_OFFSET=300`

IP_FORCE_DIRECT_MAPPING Indicates that vertices are to be stored at the index equal to their identifier, see Subsubsection 4.3.4.4. If used in conjunction with `IP_ID_OFFSET`, it trumps it.
Example: `-DIP_ID_OFFSET=300 -DIP_FORCE_DIRECT_MAPPING`

Appendix M

DiP compilation flags

The following flags are used to enable certain defines, so they must be prefixed with `-D`.

DIP_BRANCH The version of the buffer design to use. Possible values are:

- **master** the naive version of *DiP*.
- **inbs** the version of *DiP* using the Intra-Node Buffer Sharing buffer design.
- **ibmp** the version of *DiP* using the Interval-Based Message Processing buffer design.

Example: `-DDIP_BRANCH=inbs`

DIP_EDGE_OFFSET_TYPE The datatype of an offset in the CSR-stored graph to load.
Example values: `uint32_t`, `uint64_t`...
Example: `-DDIP_EDGE_OFFSET_TYPE=uint32_t`

DIP_MPI_EDGE_OFFSET_TYPE The MPI datatype equivalent of `DIP_EDGE_OFFSET_TYPE`.
Example values: `MPI_UINT32_T`, `MPI_UINT64_t`...
Example: `-DDIP_MPI_EDGE_OFFSET_TYPE=UINT32_T`

DIP_VERTEX_ID_TYPE The datatype of a vertex identifier in the CSR-stored graph to load.
Example values: `uint32_t`, `uint64_t`...
Example: `-DDIP_VERTEX_ID_TYPE=uint32_t`

DIP_MPI_VERTEX_ID_TYPE The MPI datatype equivalent of `DIP_VERTEX_ID_TYPE`.
Example values: `MPI_UINT32_T`, `MPI_UINT64_t`...
Example: `-DDIP_MPI_VERTEX_ID_TYPE=uint32_t`

DIP_MESSAGE_TYPE The datatype of values exchanged between vertices.
Example values: `float`, `double`, `uint32_t`...
Example: `-DDIP_MESSAGE_TYPE=float`

DIP_MPI_MESSAGE_TYPE The MPI datatype equivalent of `DIP_MESSAGE_TYPE`.
Example values: `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_UINT32_T`...
Example: `-DDIP_MPI_MESSAGE_TYPE=MPI_FLOAT`

DIP_COMBINATION_OPERATION_MIN Indicates that the reduction operation used for combination is the minimum operation. Automatically defines the function `void dip_combine(DIP_MESSAGE_TYPE* old, DIP_MESSAGE_TYPE new)` as well as sets `DIP_COMBINATION_OPERATION_NEUTRAL_VALUE` accordingly.

DIP_COMBINATION_OPERATION_SUM Using a reduction operation that is a sum. Automatically defines the function `void dip_combine(DIP_MESSAGE_TYPE* old, DIP_MESSAGE_TYPE new)` accordingly.

DIP_COMBINATION_OPERATION_NEUTRAL_VALUE Specifies the value that is neutral to the combination operation applied.

DIP_COMBINATION_OPERATION_CUSTOMISED Indicates that the reduction operation is user-defined. Requires the function `void dip_combine(DIP_MESSAGE_TYPE* old, DIP_MESSAGE_TYPE new)` to be defined by the user.