Task Scheduling in Data Stream Processing Systems

Leila Eskandari

a thesis submitted for the degree of

Doctor of Philosophy

at the University of Otago, Dunedin,

New Zealand.

2019

Abstract

In the era of big data, with streaming applications such as social media, surveillance monitoring and real-time search generating large volumes of data, efficient Data Stream Processing Systems (DSPSs) have become essential. When designing an efficient DSPS, a number of challenges need to be considered including task allocation, scalability, fault tolerance, QoS, parallelism degree, and state management, among others.

In our research, we focus on task allocation as it has a significant impact on performance metrics such as data processing latency and system throughput. An application processed by DSPSs is represented as a Directed Acyclic Graph (DAG), where each vertex represents a task and the edges show the dataflow between the tasks. Task allocation can be defined as the assignment of the vertices in the DAG to the physical compute nodes such that the data movement between the nodes is minimised. Finding an optimal task placement for stream processing systems is NP-hard. Thus, approximate scheduling approaches are required to improve the performance of DSPSs.

In this thesis, we present our three proposed schedulers, each having a different heuristic partitioning approach to minimise inter-node communication for either homogeneous or heterogeneous clusters. We demonstrate how each scheduler can efficiently assign groups of highly communicating tasks to compute nodes. Our schedulers are able to outperform two state-of-the-art schedulers for three micro-benchmarks and two real-world applications, increasing throughput and reducing data processing latency as a result of a better task placement.

Acknowledgements

There have been many people who have helped me throughout my PhD, without whom the completion of this thesis would not have been possible.

I would like to thank my supervisors, Associate Professor David Eyers and Associate Professor Zhiyi Huang for the invaluable advice and guidance they have provided. I am very grateful to them for giving me a chance to do my PhD under their supervision. I have learnt a great deal and appreciate the time I have spent working with them. I would also like to thank my thesis reviewers who have provided constructive feedback and suggestions to improve my work.

I am grateful to the University of Otago and the Department of Computer Science for providing me with a doctoral scholarship and travel grants. My great regards go to all the staff in the department for their support throughout my time at the University of Otago. I am especially grateful to Zhiyi, Nick, Haibo, Yawen and Sandy for giving me an opportunity to be a demonstrator. I would also like to thank the cshelp team, especially Allan, for their support during my experiments and our department administrator, Kaye, for her extraordinary organisation.

I would like to acknowledge my friends in the Department of Computer Science for their support and helpful discussions. Also, my special thanks go to my friends outside the department or even outside New Zealand who always kept my spirits up.

Last but not least, I would like to express my sincere gratitude to my dearest family, and my beloved Jason for their endless support, patience and encouragement during my thesis. I owe them a lot.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents three scheduling algorithms for DAG-based data stream processing systems. The proposed schedulers aim to efficiently find highly communicating tasks, and by placing them on the same node, increase system throughput while decreasing latency. In this chapter, we first provide an overview of data stream processing systems. Then, we highlight the challenges in stream processing, before outlining our key contributions. Finally, we present the structure of this thesis.

1.1 Data stream processing systems

The recent growth in data generation has arisen from new data-intensive services such as scientific experiments, social networks, media, stock trading and weather monitoring, among others. More than 30,000 gigabytes of data is generated every second and the rate is accelerating (Marz and Warren, 2015). According to IBM,[1] 90% of the data that existed in 2012 was created in the two years prior.

These data sources need to be analysed to gain insights, and find trends such as determining the most frequent events for a continuous dataflow occurring over a certain period of time. For example, to follow events as they begin trending on Twitter, we could track the top 10 topics over the past 30 minutes, which requires continuous analysis of tweets as they are generated.

Data Stream Processing Systems (DSPSs) are designed to process such dataflows, by operating on data streams, which are a continuous, unbounded sequence of data items, with a number of data attributes, processed in the order in which they arrive.

[1] https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/

Some of the characteristics of data streams are (Chakravarthy and Jiang, 2009; Babcock *et al.*, 2002):

- The data items in the stream arrive continuously, and are processed in the order they arrive without being stored.
- The input rate of data is uncontrolled and can be bursty or irregular.
- Data streams are unbounded in size.
- As data comes from external sources, it may contain missing values needing to be handled, i.e., some values might not have been logged properly. Further, data may be corrupt due to network problems.
- Data streaming applications are required to handle varied types of data, i.e., structured, semi-structured or unstructured.

DSPS applications such as financial applications, network monitoring, security surveillance, telecommunications data management, web applications, traffic monitoring and smart homes operate on continuous flows of data (Chakravarthy and Jiang, 2009; Abadi *et al.*, 2003; Babcock *et al.*, 2002; Querzoni and Rivetti, 2017). In comparison, batch processing systems store the data before performing *ad hoc* queries, which is not suited for real-time analysis.

1.2 Challenging issues in data stream processing systems

A general purpose data stream processing system faces a number of competing challenges, such as providing high performance and throughput while remaining reliable. It should allow users to express their queries with continuous SQL-based query languages or to develop their queries with a programming language. Some of the challenges in designing a DSPS include:

- Scheduling—A distributed stream processing system needs to efficiently schedule tasks across distributed nodes to improve system performance.
- Rescheduling—Triggering rescheduling creates some overhead in the DSPS execution, requiring a balance to be struck between the potential performance gains of the new scheduling and the overhead incurred.

- Parallelism degree—It is important to find the right number of tasks per operator, providing parallelism. This has a significant impact on performance as too few tasks may create bottlenecks, degrading performance, while too many may waste system resources.
- Scalability—A DSPS should be able to respond to changes in the load by scaling in or out, as required.
- Fault-tolerance—Like any other distributed system, DSPSs are prone to software and hardware faults, which requires the ability to efficiently recover from failures without significant interruption.
- Reliability—A DSPS needs to replay data streams in the event of a failure.
- State management—When rescheduling, it is important to preserve the state of each task being migrated to minimise data loss.
- QoS specifications—Some applications may require specific QoS guarantees such as throughput, tuple latency and memory usage, among others.
- Data security—When handling data streams of sensitive information, DSPSs may require some guarantees for data confidentiality, integrity and availability, to secure end-to-end communication.

It is not possible to optimise all of the above mentioned challenges at the same time as there will be tradeoffs. The specific streaming application requirements determine which challenges need to be addressed. For instance, a security surveillance system needs to be reliable, which can be achieved through replication of data within the system, coming at the cost of increased latency. In comparison, a stock trading system requires low latency, which can be achieved by an efficient, high performance task placement policy.

While each of these challenges are current areas of research, we focus on scheduling in this thesis as low latency response times are a consistent priority across many streaming applications. The scheduling policy determines how tasks are distributed in the data stream processing system, which can have a significant impact on the performance metrics of the system such as tuple latency (the time taken to process a tuple) and system throughput (the number of tuples processed in a given time) (Chakravarthy and Jiang, 2009). A scheduling policy needs to strike a balance between system performance, the use of system resources and run-time overhead.

An efficient task scheduler will adapt to changes in the communication pattern of a streaming application, ensuring that the inter-node communication is minimised. Specifically, by placing highly communicating tasks on the same node, communication between compute nodes can be reduced. The term "highly communicating tasks", which is used throughout this thesis, refers to a pair or group of tasks which exchange a larger amount of data than other neighbouring tasks. Additionally, prioritising the use of higher capacity compute nodes allows more highly communicating tasks to be co-located within a compute node, requiring fewer nodes to be used, which helps to further reduce inter-node communication. To achieve this, the scheduler monitors the run-time communication of a streaming application, logging the communication rates between tasks and tasks load, which is then used when rescheduling.

Existing work on task schedulers which aim to minimise the inter-node communication have a number of limitations. Firstly, the compute nodes might be underutilised which results in using more compute nodes than required. Secondly, many of the schedulers are not designed for heterogeneous clusters, which is an important requirement for many deployments, where clusters grow over time as new hardware is added. Further, a multi-user homogeneous cluster where not all of the system resources are available can be viewed as a heterogeneous system. Finally, offline schedulers are incapable of adapting to the run-time changes in the traffic patterns of streaming applications. In this thesis, we aim to address these limitations in the design of our proposed schedulers.

1.3 Contributions

In this thesis, we address the problem of task scheduling in DSPSs and propose three heuristic schedulers: P-Scheduler, T3-Scheduler, I-Scheduler. A streaming application is represented as a Directed Acyclic Graph (DAG), where each vertex represents a task and the edges show the dataflow between the tasks. Task allocation can be defined as the assignment of the vertices in the DAG to the physical compute nodes such that the communication between the nodes is minimised. Each of the three schedulers adopt a different approach for partitioning the DAG in order to minimise the communication between each part, such that inter-node communication is minimised when each part is assigned to a compute node. Further, all three approaches use a two-level scheduler, where the first level determines the tasks to be assigned to each compute node, with the second level assigning tasks to worker processes within each node. Such a two-level approach is required when the compute nodes run multiple workers for fault

tolerance and stability in large deployments. In the following, we provide a more detailed description of each proposed scheduler.

- We propose P-Scheduler, which finds the number of homogeneous nodes required for an application, and partitions the application graph, based upon this number, using K-way partitioning, minimising the inter-node communication. P-Scheduler also minimises inter-worker communication within each node by grouping highly communicating tasks within a worker.

- We propose T3-Scheduler, which finds partitions of highly communicating tasks where each partition has a size relative to the capacity of a node in the heterogeneous cluster. By considering the shape and connectivity of tasks in the application graph when sizing each partition relative to node capacity, T3-Scheduler can ensure nodes are fully utilised, minimising inter-node communication.

- We propose I-Scheduler, for heterogeneous DSPSs, which reduces the size of the task graph by fusing highly communicating tasks, allowing mathematical optimisation software to be used to find an efficient task assignment. A fallback heuristic is also proposed for cases where the optimisation software cannot be used, which iteratively partitions the application graph based on the capacity of the heterogeneous nodes and assigns each partition to a node with relative capacity.

- We evaluate the communication cost of each of the proposed schedulers by comparing them with a theoretically optimal scheduler. Our evaluation shows that our proposed schedulers can achieve results close to optimal. Further, we implement the proposed schedulers in Apache Storm 1.1.1 and run experiments using three micro-benchmarks and two real-world applications to evaluate the proposed schedulers and compare them with state of the art R-Storm (Peng *et al.*, 2015) and Aniello et al.'s 'Online scheduler' (Aniello *et al.*, 2013). The results show that our schedulers can outperform OLS by 12–86% and R-Storm by up to 32%.

Throughout this thesis, we present the design, implementation and evaluation for each of our three proposed schedulers, where the evaluation criteria are the achieved latency and system throughput which is compared to two state of the art schedulers.

1.4 Publications

Some of the material presented in this thesis has previously been published, which are listed below:

- Eskandari, L., Huang, Z., and Eyers, D. (2016). P-Scheduler: Adaptive hierarchical scheduling in Apache Storm. In *Proceedings of the Australasian Computer Science Week Multiconference,* 26. ACM.

- Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2017). A Topology and Traffic Aware Two-Level scheduler for stream processing systems in a heterogeneous cluster. In *Proceedings of the 23rd European Conference on Parallel Processing,* 68–79. Springer.

- Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2018). Iterative scheduling for distributed stream processing systems. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems,* 234–237. ACM.

- Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2018). T3-Scheduler: A Topology and Traffic aware Two-level scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems*, 89, 617–632. Elsevier.

1.5 Thesis structure

The remainder of this thesis is structured as follows.

Chapter 2 provides background on big data, its requirements, and processing frameworks including batch and data stream processing.

Chapter 3 provides the state-of-the-art on scheduling in data stream processing systems. Further, we outline how scheduling in data stream processing systems is different with scheduling in batch processing frameworks and workflow data processing systems.

Chapter 4 provides a system model and formulation of the scheduling problem which is solved in this thesis.

Chapter 5 provides a comprehensive introduction of the Apache Storm framework, a distributed, reliable, fault tolerant and open source data stream processing system which is used in this thesis for implementation and experiments.

Chapter 6 provides all the settings used in the experiments and a description of the applications, which are common in all of the proposed schedulers.

Chapter 7 presents P-Scheduler, our graph partitioning-based scheduler for homogeneous clusters.

Chapter 8 presents T3-Scheduler, which proposes an algorithm for finding groups of highly communicating tasks to be scheduled on a heterogeneous cluster.

Chapter 9 presents I-Scheduler, an iterative graph partitioning algorithm for reducing the task graph size so optimisation software can be used to perform the task assignment.

Chapter 10 concludes this thesis with a summary of the key contributions and highlights some future work.

Chapter 2

Background

We begin this chapter by defining the 4 Vs of big data, before outlining required analysis techniques and the respective limitations of existing systems for big data analysis. We then describe two popular batch processing frameworks and the three generations of stream processing systems, highlighting the advances made in each generation. This is then followed by a detailed description of the Lambda architecture, which benefits from both batch and stream processing systems. Finally, we discuss how event stream processing is different to complex event processing, before concluding the chapter.

2.1 Big data description

Big data is defined by having at least one of the following characteristics, which are referred to as the 4 Vs:

- **Volume:** The data volume is huge, on the order of petabytes and a large storage space is required. For example, social media generates large volumes of data with Facebook users having uploaded 240 billion photos, continuing at a rate of 7 petabytes per month, while 12 terabytes of Tweets are sent each day.[1] IDC estimated the size of the digital universe as 4.4 zetabytes in 2013, and have forecast it will grow to 44 zetabytes by 2020, representing a tenfold increase in only seven years.[1]

- **Velocity:** The data is created quickly and requires low latency, high velocity access within distributed deployments (Zikopoulos and Eaton, 2011). For example, Facebook has to handle more than 900 million user photo uploads every day.[1]

[1] `https://www.emc.com/leadership/digital-universe/2014iview/index.htm`

Stock trading is a common example, where 5 million trade events are created each day, which require fast processing and responses as the information loses value over time. Velocity is the measure of how quickly data arrives and needs to be processed.

- **Variety:** Data resides in a variety of file formats and can be structured, semi-structured and unstructured. Traditionally, information has been stored in a structured way, such as within relational databases, but many new sources of data do not have such rigid structures. That is, photos, video, email and documents, among others, are either semi-structured or unstructured, and cannot efficiently be stored or processed by traditional systems.

- **Veracity:** As data can come from a number of different sources, it is important to ensure that the data is accurate and trustworthy. It is possible that the data contains bias, duplication, inconsistencies and abnormalities which can affect the data quality and the results of the data processing and analysis.

It is these characteristics of big data which pose the challenge when analysis is performed with generalised data processing frameworks. This has led to the development of tailor made solutions, that are designed to handle the volume or velocity of data in a manageable way. These 4 Vs pose a number of challenges for analysing big data, which has led to the development of tailor made data processing frameworks.

2.2 Big data analysis techniques

To process big data, a number of analysis techniques are used:

- **Batch analysis** is where a batch or queue of jobs are performed infrequently on blocks of data that have been stored in the system over a period of time. The operations are processed non-interactively, over potentially large volumes of data. For example, generating business reports is a common batch operation, where the financial records of a firm for the past week or month are analysed.

- **Real-time analysis or stream processing** provides a solution for analysing data as it is generated and flows through the system. Real-time analysis is useful for tasks like fraud detection, allowing anomalies in financial transactions to be detected before they are completed.

- ***Ad hoc* analysis or NoSQL** is performed on large data stores, containing collections of structured, semi-structured and unstructured data. Such queries can provide results from business records, documents, and graphs, among others.
- **Graph analysis** can be used to analyse the relationships between entities, where the data forms a graph, such as social graphs which show the relationships between people. By analysing the network structure of graph data, the impact or spread of changes on the graph can be evaluated.
- **Iterative analysis** allows an operation to be repeatedly performed on a large dataset to extract value or find patterns in the data. Such analysis is used by machine learning techniques, which perform a large number of iterative operations to gain insights and understanding from these huge growing datasets.

While these analysis techniques are not new, the characteristics of the data handle by each analysis technique has changed.

2.3 The emergence of big data processing frameworks

While some existing frameworks such as Grid computing and HPC are able to provide high compute performance or store large volumes of data, they are not suitable for big data processing (White, 2012). Big data systems need to be capable of scaling out across multiple nodes with reliable file systems, in order to meet the compute and storage requirements (Zaharia *et al.*, 2016), which traditional systems were not designed for. In this section, we outline some existing distributed computing and storage solutions, and highlight their limitations for processing big data.

Relational Database Management Systems (RDBMS) and NewSQL

Database systems are commonly used to store large volumes of data such as financial records, and logistic information among others. These large volumes of data can be processed with scalable relational NewSQL(Grolinger *et al.*, 2013; Aslett, 2011) systems, which are capable of scaling out processing. However, this data is stored in a structured way, with a defined format and data type for rows and columns. This makes RDBMS and NewSQL unsuitable for storing the variety of data produced, which is often semi-structured or unstructured (Marz and Warren, 2015). Moreover, the desire for processing database information as it arrives led to the development of continuous query processing systems, which run a query on streams of incoming data on the fly.

High performance computing and grid computing

High Performance Computing (HPC) and grid computing have long been used for running compute-intensive, MPI workloads. These systems are designed to maximise the processing power of each node, which can limit the scalability of such systems. Instead, a more scalable system will often tradeoff some per-node performance for a more scalable architecture. While HPC systems can process large datasets, it can be expensive to scale up these systems, as they use more specialised hardware that is optimised for performance. In comparison, a scalable system uses less optimised hardware, which can more readily be deployed to scale out the system, in response to growing user demand.

Volunteer computing

In this computing framework, volunteers from around the world donate spare compute resources by running a compute-intensive application when the volunteer node is idle. This allows users to contribute towards a collective goal by performing computations, before sending the results back to a centralised server. Two popular examples are SETI@Home[2] and Folding@Home.[3] SETI is designed to distribute small chunks of radio telescope data to the volunteer systems for processing. Each system performs a number of compute-intensive operations on the received data, searching for any signals from any extraterrestrial intelligence. Figure 2.1 shows the flow of data through the SETI infrastructure. The distributed and varied nature of the compute resources and bandwidth is not designed for accessing large volumes of data, making it unsuitable for big data processing.

Each of these systems was designed to serve a specific computing need, such as RDBMSs for processing business analytics and HPC for performing compute-intensive operations, making them unsuitable to perform big data analysis. This necessitated the development of tailored processing frameworks to meet the challenges of big data. In the remainder of this chapter, we will present and discuss some of the most prominent processing frameworks for batch processing to handle large volumes of data. Further, we provide an extensive state-of-the-art on different generations of stream processing frameworks designed to analyse high velocity data, which is the focus of this thesis.

[2] `https://setiathome.berkeley.edu/`

[3] `http://folding.stanford.edu/`

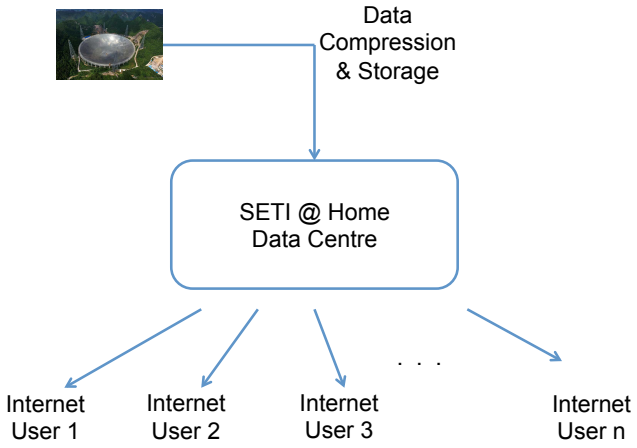


Figure 2.1: The flow of data through the SETI infrastructure

2.4 Big data processing frameworks

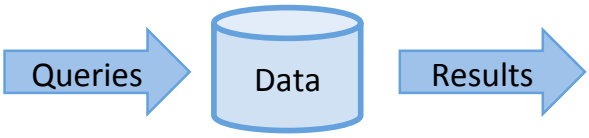
Big data processing can be classified into two primary classes of computation depending on whether the data is static or dynamic. In the former, the data is stored and queries are passed to the data whereas in the latter case queries are stored and data is passed to the queries (InfoSphere, 2018). The differences between these two categories are shown in Figure 2.2.

Batch processing frameworks, such as Hadoop,[4] are commonly used to process large volumes of static data offline. However, when the data is unbounded in size and streaming, it needs to be processed on the fly, as it arrives in near real-time, and thus batch processing approaches cannot be used. A DSPS is designed to manage continuous and unending data streams which execute continuous queries without needing to store the data.

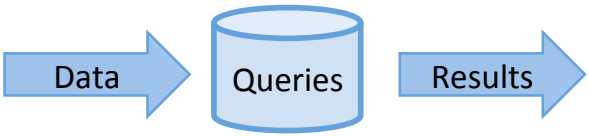
Each of these big data frameworks address particular challenges in analysing big data by handling different characteristics of the data, making them suitable for particular use-cases. While batch processing systems can view all of the data stored during analysis, providing accurate results, stream processing systems have a limited view of online and recent data as it arrives, ensuring low latency analysis. To balance latency and accuracy, Lambda architecture can be used which takes advantage of both batch processing and stream processing to provide low latency and accurate results over batch data.

The focus of this thesis is on data stream processing systems. However, as big data processing is often synonymous with large volumes of data and batch processing,

[4] `http://hadoop.apache.org/`



(a) Static data computation



(b) Streaming data computation

Figure 2.2: Static data computation versus dynamic data computation, reproduced based on (InfoSphere, 2018)

we first present common batch processing frameworks, from which a comparison with DSPSs is based. We then provide a comprehensive background on different generations of DSPSs. We also provide a detailed description of the lambda architecture to present how batch and stream can work together in particular use cases.

2.4.1 Batch processing systems

In this section, we describe two of the most popular batch processing systems, Apache Hadoop and Apache Spark.

Apache Hadoop is an open source software framework for processing and storing large datasets on scalable clusters of commodity hardware. It was developed based upon the Google File System (GFS) (Ghemawat *et al.*, 2003) and MapReduce (Guo *et al.*, 2004) research papers from Google. Hadoop is made up of two primary components: the Hadoop Distributed File System (HDFS) for storing the large volumes of data and MapReduce (Dean and Ghemawat, 2008) which is the programming model used to process the data.

HDFS stores large files by breaking them up into smaller chunks which are stored on different cluster nodes. The files are further replicated, and placed on different nodes, providing reliable storage. This method of storage has the further benefit of allowing parallel processing to occur on multiple nodes as the required data is already in distributed storage.

The distributed data is processed by the MapReduce programming model, which performs the computations in the nodes where each chunk of data is stored. This makes the processing parallel, providing good performance on large datasets.

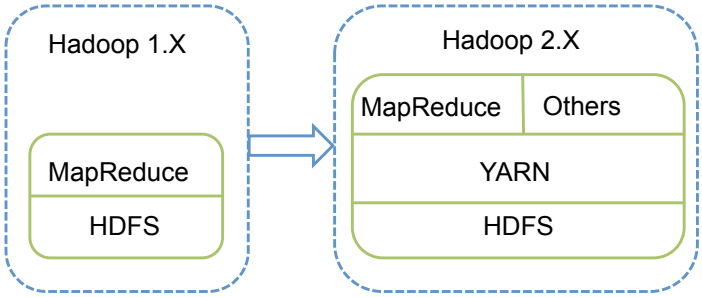


Figure 2.3: Hadoop 1.X vs. Hadoop 2.X

In the original Hadoop 1.X, MapReduce was responsible for resource management and processing. The YARN resource manager was introduced in Hadoop 2.X, which separated the resource management from the MapReduce, programming model, which all sits on top of HDFS. YARN contains a global Resource Manager (RM) for managing the cluster resources and a per-application Application Master (AM) for managing the resources and execution of each application.

While Hadoop began as a MapReduce framework, it has become more general with the development of YARN, which, along with HDFS, has allowed Hadoop to grow and support many other big data frameworks, implemented on top of this platform.

Apache Spark is a distributed data processing framework originally developed in the AMPLab at UC Berkeley and uses a programming model similar to MapReduce. It is extended with Resilient Distributed Datasets or RDDs (Zaharia *et al.*, 2012), which provide a distributed memory abstraction of data. RDDs allow data to be held in memory during execution, improving run-time performance. RDDs are able to recompute and recreate datasets, which makes Spark fault-tolerance. Spark can be deployed as a stand-alone server or it can be deployed on a distributed framework such as YARN. By deploying on YARN, Spark is able to use HDFS for storing large datasets on disk.

2.4.2 Data stream processing systems

Over the last few years, a broad range of research has advanced stream processing systems. These advances can be characterised by three distinct system generations (Heinze *et al.*, 2014) as follows.

First generation stream processing systems

The first generation of stream processing systems were developed to be standalone, with a specific application in mind. The development of these systems was very database-centric, adopting query languages similar to SQL, which performed continuous functions such as aggregate and filter, on data streams (Stonebraker *et al.*, 2005). Examples of first generation include: Tribeca, Aurora, TelegraphCQ and STREAM, described in more detail as follows.

Tribeca

Tribeca (Sullivan and Heybey, 1998) was developed in Bell Communications Research for the purpose of network traffic monitoring. It takes a single information flow as input and produces one or more output flows. An imperative language is used to express the sequence of operators applied to the information flow (Cugola and Margara, 2012), where the operators are selection, projection and aggregate, which supports count and sliding windows. Further, multiplexing and demultiplexing operators can be used to split and merge flows. Queries are optimised by ensuring that the intermediate results are able to fit within main memory, minimising query execution time (Panigati *et al.*, 2015).

Aurora

Aurora (Carney *et al.*, 2003) is an early single site stream processing system developed by Brown University and MIT, which was motivated by the analysis of sensor data streams. An application in Aurora is a graph of 'boxes' and 'arrows' where each box denotes a processing element and each arrow is a stream connecting boxes. Tuples arrive at the input queues of each box and a scheduler decides which box to run. The tuples are sent to the input queues of the next box after processing is done at each box. Aurora has seven primitive operators. Map, Aggregate, Resample, Filter, Union, BSort, Join and supports additional custom operators. Shared queues are used to support shared storage for sliding windows on streams.

TelegraphCQ

TelegraphCQ (Chandrasekaran *et al.*, 2003; Madden *et al.*, 2002) is a continuous query processing system, based on a declarative SQL language, for networking applications, developed at UC Berkeley. It derives all the relational operators from SQL, which is an extension of PostgreSQL. It processes unbounded streams

of data as window operators, allowing users to set their own policies for sliding windows. As an optimisation, TelegraphCQ inspects sets of queries for commonalities, using shared query evaluation to provide fast query processing and avoid wasting resources. This adaptive shared query processing is robust to the addition or removal of queries.

STREAM

The STanford stREam datA Manager (STREAM) project (Arasu *et al.*, 2003, 2006) is a relation-based system, developed at Stanford. The system focuses on memory management and approximate query answering. STREAM merges common query plans whenever possible to improve the sharing of computation and memory resources. Continuous queries are performed on data streams using one of three sets of operators: 1) stream-to-relation, which includes window operators that compute time-based, tuple-based, and partitioned windows over streams, 2) relation-to-relation, which includes all the operators derived from traditional relational queries and 3) relation-to-stream, which includes operators that produce a stream from a relation.

Each of these systems is able to perform continuous SQL-based queries on streams of data as it flows through the system. However, these are standalone, single machine systems which are not capable of scaling up, limiting their use for large dataflows. Further, the design of the systems is not very robust or configurable, limiting the applications for which they can be used. The database-centric development results in SQL-based applications, which is not very general or applicable to many streaming applications, where users require a general programming model for more expressive and configurable use-cases.

Second generation stream processing systems

The second generation systems were built upon the characteristics of the first generation systems, but advanced in robustness and fault tolerance. Examples of the second generation include: Aurora* , Medusa, Borealis, System S and SPADE, described in more detail as follows.

Aurora* and Medusa

The Aurora* and Medusa projects (Cetintemel, 2003; Cherniack *et al.*, 2003) extend Aurora to create a distributed workflow by connecting together multiple

Aurora workflows, using an overlay network, in a distributed environment. Aurora* is designed to connect hosts within the same organisation which have a shared QoS objective, while Medusa works in environments where the hosts belong to different organisations which may have different QoS targets. The Aurora optimiser is extended for these distributed environments, enabling cross-site optimisation and load balancing. In Aurora*, hosts freely share the load with each other, preventing a single host from becoming over-burdened. Medusa adopts a bounded-price mechanism for load balancing, where offline pairwise contracts specify the price for migrating each task, and the set of tasks a host is willing to execute on behalf of its partner. At run-time a host will compare the cost of processing a task locally and the price it would have to pay a partner host to execute the same task, opting for the lowest cost solution (Balazinska *et al.*, 2004). For reliability, Aurora* and Medusa use an upstream backup server to hold a copy of each tuple until the main server confirms the tuple was processed successfully (Hwang *et al.*, 2003).

Borealis

Borealis (Abadi *et al.*, 2005; Cetintemel *et al.*, 2016) builds on previous work by using Aurora as the core stream processing engine on each node, and Medusa as the distribution framework. This functionality is extended with the ability to distribute query processing across multiple nodes, and an improved load balance which can move operators across nodes and drop tuples if a node becomes overloaded. Borealis runs redundant query replicas on each node in order to be fault tolerant and achieve high availability. Further, a revision processing mechanism allows the system to handle errors in the input tuples by recovering previous query results.

System S

System S (Jain *et al.*, 2006; Amini *et al.*, 2006; Wu *et al.*, 2007) is a large-scale distributed data stream processing middleware developed at the IBM T. J. Watson Research Center. It supports both structured and unstructured data streams and can scale up to a large number of nodes. Processing is performed by basic operators, called Processing Elements (PEs), which are connected in a graph (Babcock *et al.*, 2002). Ports are streams which connect PEs using one of two types of connectivity, hard-coded or implicit links. Hard-coded links explicitly define which PE input port is connected to which output port of another

PE. Implicit links dynamically link PE, which provides support for incremental application development and deployment.

SPADE

SPADE (Gedik *et al.*, 2008; Andrade *et al.*, 2011) is the declarative stream processing engine for System S, as well as the name of the language used to program applications for the system. It provides an intermediate language for constructing parallel and distributed dataflow graphs, which is incorporated into an application development front-end for System S. SPADE supports the basic relational operators and windowing, which is able to be extended to process list types and vectorised operations. User defined operators can also be used, which has allowed a number of adapters to be developed for ingesting data from outside sources.

While these systems have made some advances over the first generation systems, they are still heavily SQL-based which restrict the potential use-cases and types of analysis they are able to perform. Further, the limited robustness and scalability mean these streaming applications cannot support large-scale, distributed deployments.

Third generation stream processing systems (Cloud-based)

The most recent, third generation systems operate within a cloud computing environment, which are not only robust, but more scalable than the previous generations (Dinsmore, 2016). They are also much more configurable, allowing users to develop and deploy their own operators, allowing the streaming applications to work across more varied domains. Examples of third generation stream processing systems include: Apache Storm, Apache S4, Apache Samza, Apache Apex, Apache Flink and Apache Spark Streaming, described in more detail as follows.

Apache Storm

Apache Storm[5] is an open source, distributed, scalable, low-latency computing system, originally developed at BackType. The PEs in Storm are user defined operators, written in programming languages such as Java. This provides the user with fine controls over the operation and processing performed in each PE. Multiple concurrent instances of each PE can be run, providing parallel executions. Each PE is either a source or compute unit, which are connected in a DAG, where stream grouping defines how each task within a PE communicates

[5] `https://storm.apache.org/`

with the connected PEs tasks. Scalability is achieved by running a master node, which distributes tasks to each worker node for execution. The master node is also responsible for monitoring and restarting any worker nodes which fail, providing fault tolerance. Reliability is achieved by replaying the tuples when failures occur. In this thesis, we have implemented our scheduling algorithms in Storm and provide further details about Storm in Chapter 5.

Apache S4

Apache S4 (Simple Scalable Streaming System) (Neumeyer *et al.*, 2010) is a fully distributed real-time stream processing framework, developed by Yahoo. It is inspired by the key, value pair based programming model of MapReduce. PEs are user defined operators specifying how to process events, which are connected together in a dynamic network, forming a DAG. Each PE is assigned a key attribute which determines the keys processed by the PE, where each event in the system is identified by a key. These keys are used to route the events to PEs by applying a hash function to each event, ensuring all events for a specific key are routed to that exact PE each time. S4 uses checkpointing for fault tolerance, and failure recovery. When a node fails, the tasks will be restarted on a new node based on the state saved in the latest checkpoint, meaning that any events processed since the last checkpoint will be lost.

Apache Samza

Apache Samza[6] is a stream processing system, originally developed by LinkedIn. It is run on top of YARN and Kafka[7] where YARN is used for the execution environment and Kafka for message brokering. Samza allows users to define PEs, which YARN distributes and executes across the multi-node cluster. YARN is responsible for managing CPU and memory usage across the multi-tenant cluster, as well as providing system monitoring, logging and fault tolerance. Kafka is used to distribute messages between PEs, where the topic partitioning in Kafka controls the flow of topics to specific PEs. Samza is able to recover failed tasks, by leveraging the messages stored on disk by Kafka. Further, YARN is used to manage fault tolerance, logging, resource isolation, security, and locality. Such a deployment configuration is similar to Hadoop MapReduce, which provides a processing engine, while relying on YARN for the underlying system management.

[6] `http://samza.apache.org/`

[7] `http://kafka.apache.org/`

Apache Apex

Apache Apex[8] is a streaming and batch processing engine originally developed by DataTorrent. It is based upon the idea of leveraging existing distributed frameworks, and not building yet another system. As such, it runs on top of YARN and HDFS, where YARN is used for resource scheduling and management, while HDFS provides a distributed file system. Apex provides the processing engine, which specifies how the user defined PEs are run, while YARN handles the distribution of the PEs to nodes. Failure recovery is performed by checkpointing, allowing a previous state to be recovered in the event of a node crashing. To further improve the speed of application development, Apex comes with Malhar, a library of pre-built operators for data sources and destinations of popular message buses, file systems, and databases.

Apache Flink

Apache Flink[9] is an open source stream processing framework, developed at TU Berlin. It is also able to perform batch processing by treating the streams as a special bounded stream with a finite number of elements. The PEs are user defined operators, which can contain a number of operator sub-tasks for parallelism. The operator sub-tasks run independently of each other in separate threads, possibly on different nodes. Flink is designed to run on large-scale, distributed clusters, with support for a standalone cluster mode, YARN and Mesos resource managers. Fault tolerance is implemented by a combination of stream replays and checkpointing, where checkpoints provide a point for failure recovery in the event of a node crashing. Stream replays allow the events for a dataflow to be replayed while ensuring consistency is maintained with exactly-once processing.

Spark Streaming

Spark Streaming (Zaharia *et al.*, 2012) is an extension of Spark which provides scalable, high-throughput, stream processing. It receives live data streams, which are split into micro-batches and processed by the Spark engine. The main abstraction provided by Spark Streaming are Discretized Streams or DStreams, which represent a continuous stream of data. Internally, DStreams are made up of a series of RDDs, which is the immutable data store used by Spark. This means many of the same RDD operations from Spark can be performed on DStreams,

[8] `https://apex.apache.org/`

[9] `https://flink.apache.org/`

Table 2.1: A comparison of DSPSs generations

Generation	Features	Examples
First	Standalone Designed for a specific application Database-centric and SQL-based	Tribeca Aurora TelegraphCQ STREAM
Second	Still SQL-based Limited scalability Limited robustness	Aurora* and Medusa Borealis System S SPADE
Third	Operate within cloud Very robust Highly scalable Custom operators, developed in a programming language	Apache Storm Apache S4 Apache Samza Apache Apex Apache Flink Spark Streaming

as well as new time-based operations, such as sliding windows. Spark Streaming can leverage Spark RDDs recovery mechanism for fault tolerance. Further, it includes checkpointing to improve recovery performance, reducing the operations to b replayed during recovery.

A comparison of the three different generations of DSPSs is presented in Table 2.1. It is worth noting that SQL-based DSPSs cannot process semi-structured or unstructured data.

2.4.3 Lambda architecture

Lambda architecture (Marz and Warren, 2015) takes advantage of both batch and stream processing frameworks to provide comprehensive results with a low latency for *ad hoc* queries over the complete dataset. This is achieved by using the batch processing system to provide accurate results for the time window covered by the last batch operation, while the stream processing systems correct this lag, by processing the data received since the last batch run.

The Lambda architecture is split into three layers, the batch layer, the serving layer, and the speed layer, as shown in Figure 2.4. The batch layer performs the long-running batch processing jobs, precomputing views which are given to the serving layer. The speed layer runs a stream processing engine to provide up-to-date views of the data as it arrives. The serving layer takes these views from both the batch and speed layers, and presents them to the user. This architecture allows users to receive comprehensive results by running *ad hoc* queries over the views. Each layer is described as follows:

- Batch layer is used to store the immutable, large and growing dataset. This is then made available to the batch processing engine for generating views based on all of the data that has been stored up to the time of processing. This can be a slow and time consuming process, depending on the volume of data and amount of processing required. After each batch processing iteration, the existing views in the serving layer are overwritten, ensuring comprehensive results. Apache Hadoop is commonly used to implement the batch layer, using MapReduce as the batch processing framework, and HDFS as the store for all of the data.

- Speed layer is used to compensate for the lag in data processing of the batch layer by generating real-time views of the most recent data. These views only contain data which supplements the views of the batch layer. The speed layer incrementally generates views for the serving layer, as new real-time data is continuously received. Once the data is processed by the batch layer and received by the serving layer, the corresponding data in the speed layer views will be discarded. Stream processing engines, such as Apache Storm or Spark Streaming are used to implement the speed layer.

- Serving layer stores the output from the batch and speed layers, which are indexed and merged before being presented to the users as views for performing *ad hoc* queries.

2.4.4 Event stream processing vs. complex event processing

So far we have looked at Event Stream Processing (ESP), but there are other approaches to even processing, such as Complex Event Processing (CEP). However, they differ in a number of ways, including, background, intended use-cases, and the problems they handle. Event stream processing is focused on running high speed, continuous

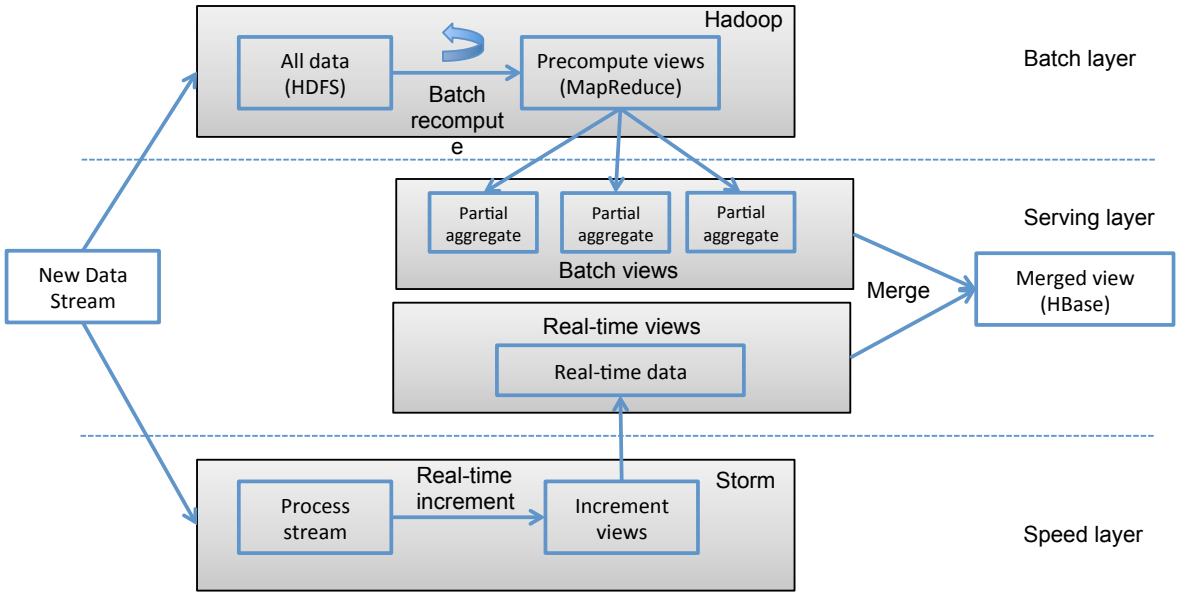


Figure 2.4: Lambda architecture (Marz and Warren, 2015)

queries on streams of incoming data to apply analytical functions to the event data (Etzion *et al.*, 2011). Its applications include network monitoring and real-time search. In comparison, complex event processing is used to extract information from a set of event sources, from enterprise IT and business systems. By searching for patterns within multiple event sources, the relationship and complex interactions between events can be extracted (Cugola and Margara, 2012). Its applications include monitoring service level agreements and anomaly detection.[10]

Currently, few ESPs are able to use timing between events or Boolean combinations. However, they are unable to evaluate more complex relationships between events, such as causality or independence. Many of the current commercial applications of ESPs do not require these more complex patterns. Going forward, as the uses of ESPs become more sophisticated, it is expected that more of these CEP events will begin to be incorporated into ESPs, creating more overlap between the two. Figure 2.5 shows the relationship between ESP and CEP and their overlap. Further, ESP and CEP are different as follows:

- Scalability—ESPs are designed to operate on parallel and distributed systems, with hundreds of nodes, as opposed to CEPs which operate on two or fewer nodes.

- Queries—ESPs allow user to implement queries using programming languages. In comparison, CEPs are often based on SQL like languages, which provide native

[10] `http://www.complexevents.com/2006/08/01/what's-the-difference-between-esp-and-cep/`

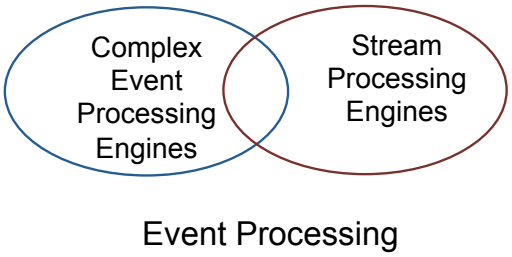


Figure 2.5: Event processing approaches: ESP and CEP

support for many operators.

- Reliability—ESPs focus on reliably processing messages, replaying messages in cases of failure. CEPs may discard events when failures occur, and continue processing new events.

2.5 Conclusion

In this chapter, we have described the characteristics and requirements of big data, and how new processing systems are required to address the big data challenges. We have also provided background on batch processing and three generations of stream processing frameworks. In this thesis, we focus on task scheduling on data stream processing systems. In the next chapter, we will discuss some existing work on scheduling algorithms for batch and stream processing systems, outlining the different approaches and optimisation goals used.

Chapter 3

Related work

As described in Chapter 2, the two main big data processing frameworks are batch processing and stream processing, which process static and dynamic data respectively. Batch processing systems analyse static data, which is stored before being processed, meaning it can handle large volumes of data. As the data is stored, it makes data locality an important consideration for task scheduling, where tasks need to be placed near the data in order to minimise the data transfers during analysis. In comparison, DSPSs process dynamic data, on the fly, as it arrives in the system, without first being stored. This in turn requires that the data is passed to the stored query. As the data is constantly moving, it is important to locate related queries together to minimise the data transfers during processing, to reduce latency.

To help highlight the differences in the scheduling approaches of each of these two big data processing frameworks, we begin by presenting some of the main schedulers for the most common batch processing system, Apache Hadoop. With this context, we then outline the three main categories that stream processing schedulers fall within, mathematical, graph-based approximation and heuristic, before describing the state-of-the-art scheduling algorithms which adopt each of these approaches. After that, we compare scheduling in other DAG-based systems with DSPSs. Finally, we discuss the limitations of existing approaches and how we address them in this thesis, before concluding the chapter.

3.1 Task scheduling in batch processing frameworks

One of the most important factors when scheduling tasks in a batch processing framework is data locality, defined as the distance between the computation (task) and the

input data (stored on a compute node). As these systems process large volumes of data, distributed across clusters of compute nodes, the distance between the tasks and data can impact performance. Therefore, by placing the computation near the input data, preferably on the same compute node, fewer data transfers are required, improving performance (Dean and Ghemawat, 2008; Rao and Reddy, 2012; Yoo and Sim, 2011; Gautam *et al.*, 2015). Scheduling algorithms that prioritise the assignment of tasks near the data address the issue of *data locality*.

However, this is not the only consideration of scheduling policies. As the clusters of compute nodes host multiple applications, there is a need to ensure that system resources are shared between the running applications, such that no single application will unfairly impact the execution of another application. System resources may be shared equally among applications, or based on other metrics, such as priority, size, or required execution time. Schedulers that prioritise the equitable distribution of system resources ensure a 'fair share of resources' are assigned to each application. These schedulers may also consider synchronisation, availability and resource utilisation. In the following, we describe some of the schedulers used in Apache Hadoop, a popular batch processing framework, which take data locality or fairness into account.

The default scheduler for Apache Hadoop is FIFO (First in First out) (Zaharia *et al.*, 2010), which assigns task in the order in which they arrive. When a map or reduce slot becomes free, a heartbeat is sent to the scheduler indicating this. Then, the scheduler picks the next job in the queue to run. Fair Scheduler (Zaharia, 2009a), developed by Facebook, assigns resources to each job such that on average, each job gets an equal share of the available resources. This policy lets short jobs finish in a reasonable time while not leaving long-lived jobs to starve in the queue. Capacity Scheduler (Zaharia, 2009b), developed by Yahoo, is designed for multiple organisations sharing a large cluster. It provides capacity guarantees for each queue and also elasticity which means that unused capacity within a queue can be allocated to overloaded queues. This results in an increased running efficiency and cluster utilisation.

LATE scheduler (Zaharia *et al.*, 2008) addresses the problem of how to robustly perform speculative execution to maximise performance. It detects a slow running task and launches an equivalent task as backup. This improves the overall performance by minimising the response time of a job when the backup task is able to finish earlier than the original task. LATE uses a metric that computes the estimated time remaining, which gives a clearer assessment of a straggling tasks' impact on the overall job response time.

Table 3.1: A comparison of Hadoop schedulers

Hadoop scheduler	**Data locality**	**Fair share of resources**	**Highlights**
FIFO Scheduler	×	×	Simple to use and configure
Capacity Scheduler	×	✓	Dedicated queues for each organisation
Fair Scheduler	×	✓	Short wait time for small jobs
LATE Scheduler	×	✓	Speculative execution
Delay Scheduler	✓	✓	Relaxing fairness for locality enhancement
Matchmaker Scheduler	✓	✓	Prioritises local tasks

The above schedulers address fairness in Apache Hadoop and not data locality. Delay scheduling (Zaharia *et al.*, 2010) achieves data locality with a minimal impact on fairness. When a job is scheduled to run next to satisfy fairness, the data locality condition is checked. If there is no local task, i.e., a task which its input data is present, it waits for a small amount of time, running other tasks. The idea in delay scheduling is that by waiting, a slot with a local task might become free after a few seconds and the data locality can improve performance.

Matchmaking scheduler (He *et al.*, 2011) enhances the data locality by giving every node a fair chance to pick local tasks before any non-local tasks are assigned to them, using a locality marker. The scheduler relaxes the strict job ordering and keeps searching until it finds a local task in the succeeding jobs. However, if a node cannot find any local tasks, a non-local task is launched to avoid wasting resources in a heartbeat interval, resulting in high resource utilisation. Table 3.1 provides a comparison of the Hadoop schedulers, discussed in this section.

3.2 Task scheduling in data stream processing frameworks

There is an extensive amount of research on scheduling in data stream processing systems with different optimisation goals. One important goal is providing communication awareness, where a scheduler takes the communication between tasks into account. By

placing highly communicating tasks on the same node, the overall inter-node communication can be reduced. Another goal is providing Quality of Service (QoS) where specific tasks are prioritised. Other goals include load balancing, reliability through message replication, maximising the resource utilisation of each node and optimising the parallelism degree for each application. Generally, there are three main approaches to tackle each optimisation problem (Chu *et al.*, 1980): mathematical programming, graph-based approximation and heuristics. In the following, we discuss a number of scheduling algorithms which use different approaches to meet specific optimisation goals.

Mathematical programming

In this approach, the scheduling problem is formulated as an optimisation problem and solved using mathematical programming techniques. A number of different optimisation techniques have been proposed for scheduling, which includes integer linear programming, and max-min fairness, among others. We present a selection of papers which adopt different techniques in solving different optimisation goals.

Cardellini *et al.* (2016) formulate the optimal schedule using integer linear programming. It considers the heterogeneity of the computing and networking resources, finding the optimal solution for a small number of tasks. The resolution time of this approach grows exponentially with the problem size, making it impractical for larger problems. The same authors propose a model in (Cardellini *et al.*, 2017) to find the optimal number of replicas for an operator in a DSPS, which has the same scalability problem.

Eidenbenz and Locher (2016) provide a theoretical analysis of task allocation and prove its NP-hardness. They propose an approach to compute optimal resource assignments for each task, where the application graph is from the class of series-parallel decomposable graphs. Their algorithm assumes that the computational cost of the tasks dominates the communication cost, and achieve an approximate solution for resources with uniform capacities and bandwidths.

SODA (Wolf *et al.*, 2008) is an optimised scheduler designed for System S (Amini *et al.*, 2006). It models the placement of Processing Elements (PEs) on the processing nodes for admitted jobs as a mixed integer programming problem. A heuristic approach is used as backup, in case the optimal solution fails.

SQPR (Kalyvianaki *et al.*, 2011) presents an optimisation model for query admission, allocation and reuse. If an optimal solution is not able to be found within a given

time, the best solution up to that point will be used. While it does not guarantee an optimal solution, the scheduling algorithm can find a schedule in a fixed amount of time.

Jiang *et al.* (2016) propose a max-min fairness approach for scheduling multiple streaming applications on a heterogeneous cluster, by formulating the problem as a mixed 0-1 integer program. As this problem is NP-hard, they propose an approximate approach by converting the non-convex constraint into linear constraints using linearisation and reformulation techniques.

DRS (Fu *et al.*, 2015) finds the optimised number of tasks for each component, minimising the processing time for the input data. DRS models the relationship between the number of tasks and response time, based on the theory of Jackson open queuing networks. After finding the optimised number of tasks, a rebalance is run to distribute the new number of tasks per operator across the cluster.

Wang *et al.* (2017) model the system as G/G/M priority queues using the Allen-Cunneen formula to estimate the average end-to-end response times. They formulate the resource allocation as an optimisation problem, with three cost functions: reducing QoS violation, maximising CPU utilisation and minimising overhead. The state of the underlying platform is continuously monitored and the allocation of resources is periodically adjusted to achieve the optimisation goals.

Each of these mathematical techniques attempt to solve a different optimisation goal. However, as the problem is NP-hard, many of these techniques rely on heuristic methods as a fallback, in the event of being unable to find a solution within a feasible amount of time. This is a key limitation of such approaches, where the intended method of finding an optimal solution is not practical.

Graph-based approximation

In graph-based approximation approaches, the application graph is considered as a DAG where vertices and edges represent tasks and the dataflow between tasks respectively. Then, graph-based approximation techniques are used to tackle the scheduling problem. For example, to reduce the inter-node communication, the vertices are assigned to compute nodes using graph algorithms such that the communication cost is minimised. Quincy (Isard *et al.*, 2009) for Dryad (Isard *et al.*, 2007) maps the problem of task to worker assignment into a graph over which a min-cost flow algorithm is used. It then minimises the cost of a model which includes data locality, fairness, and starvation freedom.

A commonly used method for finding groups of highly communicating tasks is to partition the DAG, where each partition can then be assigned to a compute node. COLA (Khandekar *et al.*, 2009) in System S uses a graph partitioner to fuse multiple PEs of a stream processing graph into bigger PEs in order to decrease inter-process communication. Fischer and Bernstein (2015) propose a mapping of workload scheduling to graph partitioning problem. Ghaderi *et al.* (2016) develop a dynamic stochastic graph partitioning and packing method for allocating resources of graph-based streaming applications. It also provides a tradeoff among average partitioning cost and average queue lengths.

Caneill *et al.* (2016) take a more fine grained approach to graph partitioning, where the aim is to partition the bipartite graph such that pairs of keys that appear together frequently are in the same group. They analyse the correlation between keys used in successive fields grouping, and explicitly map correlated keys to operator instances executed on the same server. Hence, data tuples containing these keys can be passed from one operator to the next through an address in memory, instead of copying the data over the network. This has the advantage of being faster, and avoids the saturation of the network infrastructure.

An alternative method of improving the performance of a streaming application is to reduce the critical path when assigning tasks to nodes, thereby reducing the response time. Such an approach is used in Re-Stream (Sun *et al.*, 2015) where the critical path of a data stream graph is identified, allowing the critical vertices on the critical path to be reallocated, minimising the response time. It also consolidates the non-critical vertices on the non-critical path to maximise energy efficiency.

Another approach to reduce the critical path is proposed in (Sun and Huang, 2016) where an online scheduler achieves system stability with a makespan guarantee. First, some relevant dynamic information, such as, the input rate of the data steam, and the current available capacity of each compute node, needs to be collected. The new task placement scheme is obtained through the dynamic critical path based on an earliest finish time priority strategy.

In summary, by treating streaming applications as a graph, existing graph analysis techniques can be applied to the problem of scheduling. Such techniques include graph partitioning and critical path analysis, which mainly aim at reducing inter-node communication. Having a global view of the application graph can help to better locate partitions of highly communicating tasks, when compared to some of the heuristic approaches discussed in the next section.

Heuristics

To work effectively, heuristic approaches make a number of simplifying assumptions that can result in sub-optimal solutions. Despite this, they are useful when an optimal solution is not essential, or when determining such an optimal solution would be too time consuming (Abrams and Liu, 2006). A large number of heuristic approaches have been proposed in the literature that are able to find efficient solutions in a timely manner (Lakshmanan *et al.*, 2008). In this section, we present a selection of heuristic scheduling policies, and discuss the different approaches taken by each method.

Since older DSPSs were centralised with limited scalability, their optimisation goals focused on reducing memory consumption and latency. In comparison, the third generation DSPS are highly scalable, operating across distributed nodes, which shifts the focus of the optimisation goals towards node selection and assignment. For the sake of completeness, we discuss the schedulers within three early DSPSs Aurora, STREAM and Borealis, before focusing on the scheduling algorithms proposed for current cloud based DSPSs.

The global scheduler in STREAM (Arasu *et al.*, 2003) controls the execution of query plans using a simple round-robin heuristic to select an operator to execute. Then, an operator specific procedure is called which passes the maximum number of input tuples, able to be processed, to the operator, returning control to the scheduler to select the next operator.

Aurora (Abadi *et al.*, 2003) uses a heuristic schedule to select which operator to execute by finding the first downstream operator whose queue is in memory. It will then consider other upstream operators, continuing until reaching an operator with no queue in memory, or no operators remain. This results in a sequence of operators, able to be scheduled one after the other.

Borealis uses a heuristic algorithm (Xing *et al.*, 2005) for load balancing by minimising the variation in load across the nodes. Each node and operator store load statistics in time-series, which is used to score the pairing of operator and node. Based on these scores, a greedy approach is used to select the pair with the highest score for assignment. A problem with this algorithm is that it assumes a high level of network bandwidth and that any delays in network transfers are negligible.

These heuristic scheduling methods were designed for centralised traditional stream processing systems. Current cloud-based DSPS schedulers should distribute the tasks across a large cluster while achieving specific optimisation goals such as minimising inter-node communication, minimising latency, load balancing, etc.

EvenScheduler is the native scheduler in Apache Storm[1], a popular cloud based DSPS, which uses a simple round-robin strategy to evenly allocate tasks to compute nodes. EvenScheduler has two phases. In the first phase, the tasks are assigned to worker processes using a simple round-robin heuristic. In the second phase, the workers are evenly assigned to worker nodes. This has the advantage of being easy to implement, and is load balanced as tasks are evenly allocated to compute nodes. However, it is offline and ignores the communication patterns between tasks.

One of the most commonly adopted optimisation goals is the minimisation of inter-node communication. That is, by assigning highly communicating tasks to the same node, the communication between nodes will be reduced, improving performance. This is especially important on current, cloud-based DSPSs which are highly scalable, where a large number of nodes may be used. In the following, we discuss a number of proposed scheduling policies with this as the optimisation goal.

Aniello *et al.* (2013) propose a static, offline graph analysis method that derives a partial order of the components of a streaming application based on their connectivity. It then places the tasks of each operator within the same node as those of the adjacent operator. However, such a linear set is incapable of representing the entire streaming application communication pattern. Additionally, offline analysis cannot identify the amount of communication between tasks.

To address this inability to use run-time values, Aniello *et al.* (2013) also propose an online scheduler (that we refer to as OLS in this thesis). OLS monitors the data transfer rate between communicating tasks, CPU load and memory usage at the beginning of execution. These values are used to minimise the inter-worker and inter-node traffic, through a best fit greedy approach. First, communicating pairs of tasks are iterated over in descending order of tuple transfer rate. If none of the tasks in the pair have been assigned, they are placed in the least-loaded worker. If one or both tasks have been assigned before, a set is built to evaluate the inter-worker traffic of assigning to either one of the hosting or least loaded worker. The worker with minimum inter-worker traffic is selected. The same procedure is applied when assigning workers to worker nodes. Communicating workers are iterated over in descending order of the tuple transfer rate between them. A set is built from the least-loaded node and the nodes hosting the workers, which is evaluated to find the node assignment which minimises the inter-node traffic. However, this approach inspects each task pair in isolation and cannot see all the communications between tasks, therefore some communicating task

[1] `https://storm.apache.org/`

pairs might end up in different nodes. Additionally, the nodes are not fully utilised as the algorithm tends to use all of the nodes in the cluster.

The limitations of a task pair view is improved in (Chatzistergiou and Viglas, 2014) which uses a different heuristic approach to inspect communicating groups of tasks instead of task pairs. It defines a metric for two communicating tasks, called relative significance, which is calculated as the data transfer rate between the tasks over the sum of tasks' load. The nodes are sorted based on capacity and the communicating groups of tasks are sorted based on relative significance. Using a greedy approach, a group pair is selected to be assigned to the node with the highest remaining capacity. However, the nodes are not fully utilised, because this approach has a tendency to spread the groups of tasks across the cluster as a result of inspecting each group pair in isolation.

To better consolidate the cluster nodes and fully utilise the resources, Peng *et al.* (2015) propose R-Storm,[2] an offline resource-aware scheduler, which considers the CPU, memory and bandwidth requirements of each task, as specified by the user. It then deals with the scheduling as a 3D knapsack problem. However, the need for users to provide such detailed task requirements beforehand might not always be a practical or convenient solution.

Similarly, T-Storm (Xu *et al.*, 2014) fully utilises each node by filling it to capacity. The algorithm sorts all of the tasks in descending order of their incoming and outgoing load. Then, it assigns tasks to available nodes using a bin packing approach, where nodes are sorted by capacity. Each node in T-Storm can only have one worker in order to remove inter-process traffic between workers. However, when assigning tasks to the node, each task's total load is considered separately and the traffic rate between tasks is ignored.

Similar to T-Storm, D-Storm (Liu and Buyya, 2017) schedules tasks using bin packing, where tasks are prioritised based on a formula which weights tasks by their newly introduced and potential intra-node communication. Tasks are then assigned, in descending order of priority, using First Fit Decreasing (FFD) heuristic , which places them in the first node with sufficient capacity.

For a more dynamic scheduler, which adapts to each workload type, Rychlý *et al.* (2015) profile each application and assign tasks based upon the dominant resource, i.e., tasks that heavily use the CPU are assigned to nodes with the largest CPU capacity, while graphic-intensive tasks are placed on nodes with powerful GPUs.

[2]Available in the last release of Apache Storm

Another consideration during scheduling is to optimise the parallelism degree, that is, the number of parallel tasks used for each operator. Such an approach is taken by Li *et al.* (2017) who propose a dynamic optimisation algorithm based on the theory of constraints which adjusts the number of tasks for each operator to eliminate performance bottlenecks at run-time. They first calculate the capacity of each operator and analyse the message processing queue congestion degree to determine an application's performance bottleneck. Then, they change the number of tasks per operator accordingly to mitigate the bottleneck, improving performance.

Mayer *et al.* (2015) ensure a high degree of parallelism by using a pattern-sensitive stream partitioning model. This model predicts the workload distribution and its parameters using a mix of matching the distribution moments and the maximum likelihood methods. The parallelism degree is dynamically adapted utilising queuing theory.

Shukla and Simmhan (2018) break the problem of task scheduling into two parts. First, they determine the degree of parallelism for each task, where a task contains threads, such that enough threads are allocated to handle the task's peak throughput. Second, the threads are grouped and assigned together, which maximises the peak event rate, while minimising interference from other tasks, ensuring predictable performance.

The optimisation goal of (Pietzuch *et al.*, 2006) and (Rizou *et al.*, 2010) is to minimise the network usage. They both use a latency space as the search space to find the optimal placements in a distributed way. The former is based on the physical model of springs where a spring relaxation technique is used to find operator placement in the virtual nodes in the cost space and then the closest physical nodes to these solutions are found. The latter iteratively finds an optimal virtual node for each operator in the latency space, based on the Weber problem, which is then mapped to a physical node. Similar to (Pietzuch *et al.*, 2006), Cardellini *et al.* (2015) optimise scheduling by using a spring based formulation to search a 4-dimensional cost-space including latency, throughput, utilisation and availability.

In systems with limited resources, it can be desirable to prioritise specific tasks or operators. For example, Bellavista *et al.* (2014) propose priority-based resource scheduling for flow-graph based distributed stream processing systems. In their approach, application developers augment flow-graphs with priority metadata. Then, the way stream tuples are delivered to operators input queues is dynamically modified in order to dispatch data destined to important operators with higher priority than those destined to less important ones.

Similarly, Chakraborty and Majumdar (2017) propose a method to schedule mul-

tiple streaming applications in a resource constrained cluster such that no application suffers from starvation. First, all the applications are assigned the same default priority value. The priority of an application may change at run-time if predefined trigger conditions occur. In this case, a new schedule is generated where the higher priority applications get a higher proportion of resources.

Many of the scheduling techniques only respond to changes in the resource utilisation during execution. This can put them at a disadvantage for systems which have high degrees of variation. One method of addressing this challenge is to predict future resource utilisation and requirements, which allows the scheduler to be more proactive in assigning system resources.

For example, Buddhika *et al.* (2017) propose a method for predicting the expected arrival time and resource utilisation of a stream computation. These predictions are derived from time-series analysis and are stored in a proposed data structure, called a prediction ring. The predicted values are then used to calculate an interference score, which identifies the computations that are most likely to be impacted by interference. This can guide scheduling by evaluating which machines are best suited to run each computation.

Balkesen *et al.* (2013) assign input streams to nodes based on a prediction of the workload's future behaviour and meta-data, provided by the user and stream partitioning. Holt-Winters forecasting technique is used in the model which is a type of exponential smoothing. Then, heuristic Best Fit Decreasing (BFD) bin-packing is used to partition the input streams across as few nodes as possible.

Li *et al.* (2015, 2016) use Support Vector Regression with feature selections of CPU, memory, thread and machine workload and co-location to predict latency. Then, their greedy scheduling algorithm uses the predicted latency, to minimise the average tuple processing time. It starts with an initial solution and keeps improving it by adjusting task assignment to achieve its goal.

Hidalgo *et al.* (2017) proposes an adaptive stream processing system that reacts to short-term and mid-term changes in the processing graph. In the short-term, the algorithm responds to peaks in traffic, while in the mid-term, statistical information from the operators is used in a markov chain model to predict future traffic.

To respond to changes in load on-demand, some schedulers integrate elasticity methods, allowing them to dynamically adjust the scale of the deployment. Stela (Xu *et al.*, 2016) proposes a method for seamless on-demand elasticity. It defines a new metric called ETP per operator/component or machine. To scale-out, Stela calculates ETP

per operator, which identifies the operators that are congested or affect throughput because of reaching a large number of sink operators. Then, the level of parallelism for that operator is increased in order to give more resources to that operator. To scale-in, the user specifies the number of machines to be removed and Stela calculates ETP per machine, before selecting the best machines to remove and deciding where to migrate the operators.

Sun *et al.* (2018) propose an elastic online scheduling framework for multiple online applications, which minimises system response time, guarantees application fairness, and achieves high elasticity in big data stream computing environments. Data stream graph is scheduled with a priority-based earliest finish time first elastic online scheduling strategy to minimise system response time. Multiple graphs are scheduled with a max–min fairness-based multiple DAGs scheduling strategy to guarantee fairness subject to the constraint of response time.

Additionally, techniques such as meta heuristics can be used to improve task placement. For example, Smirnov *et al.* (2017) use a genetic algorithm to model the assignment as an evolutionary process. Performance statistics from each assignment are used as the measure of fitness for the evolution. The results show an improvement when compared to greedy heuristics.

TCEP (Luthra *et al.*, 2018) proposes a scheduler which adapts to changes in the environment and desired QoS by choosing between multiple placement mechanisms. As a single placement mechanism may not handle these changes, a lightweight genetic algorithm model is used to determine the best operator placement, improving QoS. The algorithm allows for seamless transitions between the operator placement mechanism at run-time.

Table 3.2: A comparison of DSPSs schedulers

Scheduling algorithm	**Solution approach**	**Optimisation goal**	**Dynamic**	**Heterogeneity**
Aniello *et al.* (2013)	Best fit	Inter-node communication	✓	✓
R-Storm (Peng *et al.*, 2015)	3D knapsack	Resource utilisation, Inter-node communication	×	✓
T-Storm (Xu *et al.*, 2014)	Bin packing	Resource utilisation, Inter-node communication	✓	✓
Cardellini *et al.* (2016)	Integer programming	Resource utilisation, Inter-node communication, QoS Metrics	✓	✓
Chatzistergiou and Viglas (2014)	Task grouping	Inter-node communication	✓	✓
Fischer and Bernstein (2015)	Graph partitioning	Inter-node communication	✓	×
Ghaderi *et al.* (2016)	Dynamic graph partitioning	Resource utilisation	✓	✓
Rychlý *et al.* (2015)	Benchmarking applications	Workload balancing	✓	×
DRS (Fu *et al.*, 2015)	Jackson open queuing networks	Resource utilisation	✓	✓
Cardellini *et al.* (2015)	Searching a cost space	QoS Metrics	✓	✓

Re-Stream (Sun *et al.*, 2015)	Critical path in graph	Energy consumption Response time	✓	✓
Caneill *et al.* (2016)	Finding correlations between the keys	Inter-node communication	✓	✓
D-Storm (Liu and Buyya, 2017)	Bin packing	Inter-node communication	✓	✓
Shukla and Simmhan (2018)	Peak throughput prediction	Parallelism degree	✓	✓
Li *et al.* (2017)	Queue congestion analysis	Parallelism degree	✓	✓
Buddhika *et al.* (2017)	Time-series analysis	Resource utilisation	✓	✓
Balkesen *et al.* (2013)	Holt-Winters forecasting	Inter-node communication Resource utilisation	✓	✓
TCEP (Luthra *et al.*, 2018)	Hybrid with a lightweight genetic algorithm	QoS metrics	✓	✓
Smirnov *et al.* (2017)	Genetic algorithm	Resource utilisation	✓	✓

Table 3.2 provides a comparison of some of the schedulers presented above. For each of the related works, the *optimisation goal* is what the algorithm is trying to achieve and the *solution approach* is the method used to reach this goal. The *dynamic* column refers to whether the scheduling algorithm dynamically adapts to the changes in the traffic or operates statically. The *heterogeneity* indicates if the algorithm works on heterogeneous or homogeneous clusters.

3.3 Workflow scheduling in cloud and grid computing

While workflows may initially appear similar to DAGs in DSPSs, there are some important differences. Workflows are often used to model a wide range of distributed applications, which are described by a DAG, where computational tasks are vertices and the data or control dependency between tasks are the directed edges. This is different to DAG-based streaming application as the precedence in a streaming DAG is not of concern as they deal with pipelined processing and unbounded streams of data. Further, how each system handles data locality is different: workflow schedulers may place tasks near any data stored on disk, whereas streaming applications do not store data, and assign tasks based on their communication pattern, i.e., highly communicating tasks are place on the same node.

Workflow scheduling policies for clusters, grids and the cloud, distribute tasks across the available resources, such that the execution satisfies specific requirements (Yu *et al.*, 2008). The selection of a schedule has a significant impact on the performance of the system including CPU utilisation, load balancing, system usage, queuing time and throughput (Xhafa and Abraham, 2008).

There are two main types of workflow scheduling policy: heuristic and QoS-based. Heuristic based workflow scheduling attempts to minimise the makespan of the workflow application, where the makespan determines the time taken to execute the application. QoS-based scheduling policies assign the workflow to resources such that a required service level is met, satisfying user constraints based on either time or cost.

Although the optimisation goals, such as increasing throughput and load balancing, may appear to be the same in workflows and streaming applications, the heuristic approaches taken to achieve each goal are different, meaning that they are not directly comparable.

3.4 Discussion

So far, we have described different scheduling approaches for stream processing systems, where each algorithm is based upon varying sets of assumptions and optimisation goals. In this research, we design schedulers that reduce latency and improve system throughput, which can be achieved with an optimisation goal of reducing inter-node communication and by fully utilising compute nodes. To reduce the inter-node communication, highly communicating tasks should be assigned to the same node, which will reduce the amount of data needing to be moved across the nodes. This can be further helped by prioritising the use of higher capacity nodes within a heterogeneous cluster as this will further reduce the number of nodes required.

Existing task schedulers have a number of limitations. Firstly, by not fully utilising nodes, more nodes will be used than necessary, which will increase the inter-node communication. Secondly, many of the schedulers are not designed for heterogeneous clusters meaning they may not take advantage of higher capacity nodes or adequately share system resources when not all resources are available. Thirdly, by operating offline, they are not capable of responding to dynamic changes in the workload and traffic patterns. Finally, many schedulers take a task-pair view, where they only consider pairs of tasks during assignment, which can result in highly communicating groups of tasks being distributed across nodes, increasing inter-node communication.

In the following chapters, we describe our three algorithms which take a broader view of the application graph to find groups of highly communicating tasks, which are then assigned to the same compute node, reducing inter-node communication. By sizing each group relative to the capacity of each node, we are further able to ensure each node is fully utilised. We compare our three schedulers to two of the most cited, open source schedulers, R-Storm (Peng *et al.*, 2015) and OLS (Aniello *et al.*, 2013), which have the same optimisation goal of reducing inter-node communication. R-Storm uses offline, 3D knapsack algorithm to consolidate the number of compute nodes used, improving resource utilisation, and reducing inter-node communication. OLS, assigns task pairs to nodes in descending order of communication rate using a best fit approach, which places highly communicating task pairs together in order to reduce inter-node communication. Our experiments will evaluate the latency and throughput of each scheduling algorithm, and will demonstrate the improvements that can be made by taking a broader view of the task graph and fully utilising compute nodes.

3.5 Conclusion

In this chapter, we presented the state-of-the-art scheduling algorithms for stream processing systems, which take one of three main approaches: mathematical, graph-based approximation and heuristic. We further highlighted the main optimisation goals of each scheduler, before outlining the main limitations of these approaches. Finally, we outlined what our optimisation goals are and how we intend to address some of the limitations in existing heuristic approaches.

Chapter 4

System model and problem statement

In this chapter, we first present a system model for a DAG-based data stream processing system. Then, we describe our optimisation goal in the scheduling problem and formulate it as an optimisation problem. Further, we discuss the limitations of such formulations in practice and conclude the chapter.

4.1 System model

Data stream processing systems such as Apache S4 (Neumeyer *et al.*, 2010), Flink[1], Storm[2] and Twitter Heron (Kulkarni *et al.*, 2015) are designed to process large volumes of data, at high velocity, as it flows through the system.

Streaming applications provide real-time analysis of data streams, with a common example being the top trending topics on Twitter, previously introduced in Chapter 1, which provides insights into what is being tweeted about at a given time. Each of the streaming frameworks represent applications as a DAG, shown as $G = (V, E)$. Each vertex v_i in V represents an operator, and each edge e_j in E represents the dataflow between two operators communicating with each other, where data flows through the system without being stored. The weight on each edge represents the communication rate, where variations in the communication rate are shown by corresponding changes in the edge weight.

An operator can either be a source or a computation unit. A source operator reads in external data and emits it as a stream into the streaming application. Computational units process the stream, received from either a source or another computational

[1] `https://flink.apache.org`

[2] `https://storm.apache.org/`

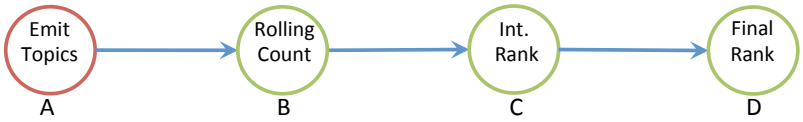


Figure 4.1: Operator-view of a streaming application

unit, before emitting the results to another computational unit. Figure 4.1 shows the operator-view graph of the top trending topics streaming application. From this figure, it can be seen that it has a linear layout with one source and three computational operators, where an edge connects two operators communicating with each other. In the top trending tweets application, the *Emit topics* operator reads tweets from the Twitter API, before sending them to the *Rolling count* operator, which counts the number of occurrences of each tweet. The counts are then sent to the *Intermediate Rank* operator, where each task finds the top 10 trending tweets for a specified time window. These intermediate rankings are then sent to the *Final Rank* operator, which aggregates the values from all of the tasks into a final rank of the top tweets.

Each operator in a streaming application has one or more tasks, where each task is an executing instance of the operator's code. This allows each task within an operator to perform the same computation on different data streams, providing parallelism. The number of tasks can be specified by the developer of the streaming application and changed during run-time to respond to variations in the workload from changes in the input rate, if required.

When more than one task is used in an operator, *stream grouping*,[3] indicates how the tasks of the emitting operator are connected to the tasks of the receiving operator, and how the stream is partitioned among the receiving tasks. In a communicating operator pair (A, B), the communication between a task in operator A and a task in operator B can be shown as a connected edge, resembling a bipartite graph.

For example, Figure 4.2 shows the task-view graph of the top trending topics streaming application, previously shown in Figure 4.1. From the figure, it can be seen that operators A, B, C and D have 5, 9, 4 and 1 tasks respectively. It can also be seen that the tasks of every two communicating operators are fully connected due to the parallelism and stream groupings.

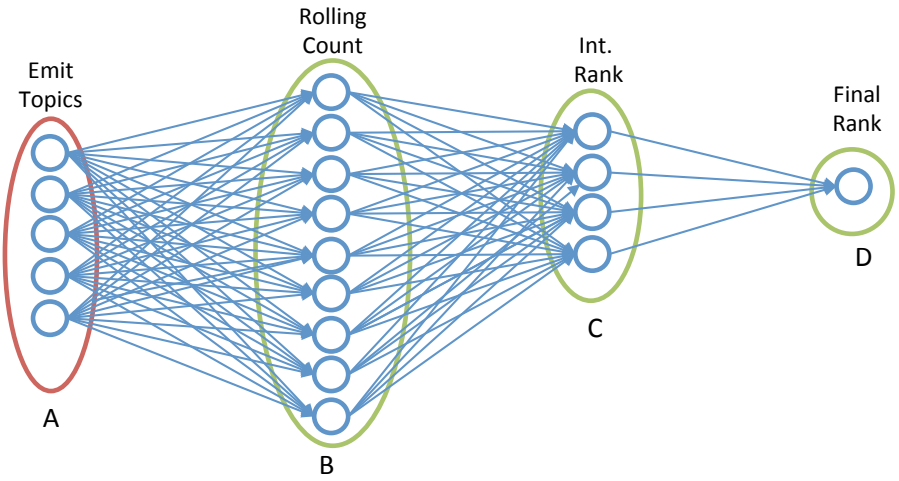
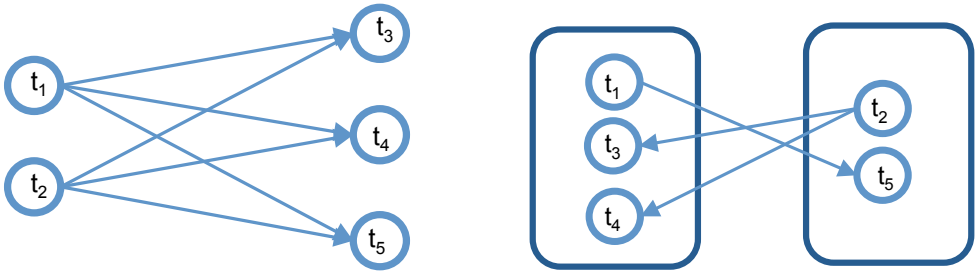


Figure 4.2: Task-view of a streaming application



(a) A simple DAG application (b) One possible optimal task placement

Figure 4.3: A simple DAG application and one possible optimal task placement

4.2 Problem statement

Each vertex and edge in the task-view graph is weighted by the task load and data transfer rate between the two communicating tasks respectively. We define the scheduling problem as mapping the vertices in the task-view graph to a set of compute nodes with different capacities, such that the sum of the inter-node edge weights is minimised and the sum of the vertex weights does not exceed the capacity of a given node. In other words, the DAG should be divided into a number of parts, where each part is sized relative to the capacity of the respective node, to fully utilise each node, while minimising the sum of edge weights between compute nodes, which reduces inter-node communication. This problem has been proven to be NP-hard (Gary and Johnson, 1983; Srivastava *et al.*, 2005; Eidenbenz and Locher, 2016) due to the large search space and computational complexity.

[3] `http://storm.apache.org/releases/current/Concepts.html`

Table 4.1: Notation for problem formulation

Symbol	Description
t_i	Task i
T	Task set
$R(t_i, t_j)$	data transfer rate between t_i and t_j
$L(t_i)$	load of t_i
n_i	node i
N	Node set
$C(n_i)$	available capacity of n_i
$x_{i,u}$	Placement of t_i on n_u
T'_i	All of the tasks assigned to n_i
$y_{i,j}$	Inter-node communication between t_i and t_j
$Cost(i, j)$	Communication cost between t_i and t_j

Similar to (Chatzistergiou and Viglas, 2014; Cardellini *et al.*, 2016), the optimal scheduling problem is formulated as follows. It is assumed that t_i and t_j are two communicating tasks in the task set T and $R(t_i, t_j)$ is the data transfer rate between these two tasks. $L(t_i)$ indicates the load of task t_i, n_i represents node i in the node set N, and $C(n_i)$ is the available capacity of n_i. Table 4.1 shows the notation used in the problem formulation.

In order to represent the allocation of t_i on n_u, a placement matrix x is defined such that:

$$x_{i,u} = \begin{cases} 1 & \text{if task } t_i \text{ is assigned to } n_u \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

In order to formulate whether two tasks, t_i and t_j are on the same node or different nodes, an inter-node communication matrix y is defined such that:

$$y_{i,j} = \sum_{u=1}^{|N|} \sum_{v=1}^{|N|} x_{i,u} \times x_{j,v} \qquad \text{for } u \neq v \tag{4.2}$$

The problem of minimising data movement between the communicating tasks in an application can be expressed as minimising the following sum:

$$Cost(i,j) = \sum_{i=1,j=1}^{|T|} R(t_i,t_j) \times y_{i,j} \qquad \forall t_i, t_j \in T \text{ and } i < j \tag{4.3}$$

subject to:

$$\sum_{j=1}^{|T_i'|} L(t_j) < C(n_i) \qquad \forall n_i \in N \tag{4.4}$$

$$\sum_{u=1}^{|N|} x_{i,u} = 1 \qquad \forall i \in T \tag{4.5}$$

The limitations in 4.4 and 4.5 ensure that the capacity of a node is not exceeded and each task is assigned to only one node. It is worth noting that matrix y is symmetric and only $i < j$ is considered in Formula 4.3.

In the following, we provide an example to demonstrate the optimisation problem formulated above. Figure 4.3a shows an example DAG application, with 5 tasks that communicate with each other. We assume a task load of 1 for each task and a communication rate of 1 to simplify this example. These tasks are to be assigned to 2 nodes, each with a capacity of 3 tasks. The placement matrix x, given by Formula 4.1, is:

$$x = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \\ x_{51} & x_{52} \end{bmatrix}$$

where each row represents a task and each column shows a node. An entry x_{iu} shows the placement of task i on node u.

The Inter-node communication matrix y for all 5 taks, found by Formula 4.2, shows all of the possible task placements as follows:

$$y = \begin{bmatrix} 0 & x_{11}{*}x_{22}{+}x_{21}{*}x_{12} & x_{11}{*}x_{32}{+}x_{31}{*}x_{12} & x_{11}{*}x_{42}{+}x_{41}{*}x_{12} & x_{11}{*}x_{52}{+}x_{51}{*}x_{12} \\ x_{11}{*}x_{22}{+}x_{21}{*}x_{12} & 0 & x_{21}{*}x_{32}{+}x_{31}{*}x_{22} & x_{21}{*}x_{42}{+}x_{41}{*}x_{22} & x_{21}{*}x_{52}{+}x_{51}{*}x_{22} \\ x_{11}{*}x_{32}{+}x_{31}{*}x_{12} & x_{21}{*}x_{32}{+}x_{31}{*}x_{22} & 0 & x_{31}{*}x_{42}{+}x_{41}{*}x_{32} & x_{31}{*}x_{52}{+}x_{51}{*}x_{32} \\ x_{11}{*}x_{42}{+}x_{41}{*}x_{12} & x_{21}{*}x_{42}{+}x_{41}{*}x_{22} & x_{31}{*}x_{42}{+}x_{41}{*}x_{32} & 0 & x_{41}{*}x_{52}{+}x_{51}{*}x_{42} \\ x_{11}{*}x_{52}{+}x_{51}{*}x_{12} & x_{21}{*}x_{52}{+}x_{51}{*}x_{22} & x_{31}{*}x_{52}{+}x_{51}{*}x_{32} & x_{41}{*}x_{52}{+}x_{51}{*}x_{42} & 0 \end{bmatrix}$$

For example, entry y_{12} shows the inter-node communication between task 1 and task 2, which is a result of either, task 1 being placed on node 1 and task 2 on node 2 or task 1 on node 2 and task 2 on node 1. Note that if both tasks are placed on the same node, there is no inter-node communication between them. As the data transfer rate is 1, the communication cost matrix, previously formulated in Formula 4.3 is the same as y. This problem can be given to the optimisation software with the limitations, given in Formulas 4.4 and 4.5, that the capacity of 3 tasks per node is not exceeded and that a task is only assigned to one node. Figure 4.3b shows one possible optimal solution, with an inter-node communication cost of 3, illustrated by the arrows.

4.3 Conclusion

While it is possible to formulate the problem of optimally scheduling tasks onto a set of nodes, it has been proven to be NP-hard, which means it is only feasible to find a solution for small problem sizes. However, as the problem size grows, it becomes impractical to find the optimal solution because of the increased computational complexity. To overcome this practical limitation of an optimal scheduler, there is a need for a heuristic approach to find a near optimal schedule in real-time. In this thesis, we propose three heuristic scheduling algorithms, described in later chapters, which can find near optimal task placements in real-time.

Chapter 5

Apache Storm

Apache Storm (Toshniwal *et al.*, 2014) is an open-source, real-time, distributed, scalable and reliable data stream processing framework. Because of these characteristics, it has been of interest in industry for processing streams of unbounded data and an area of research in academia. A significant amount of work on scheduling in data stream processing systems has been implemented in Storm, because the framework provides a plugin scheduler, allowing users to write their own custom scheduler. This work includes two popular and most cited DSPS schedulers that have been chosen for comparison in this thesis.

It is for these reasons that we use the Storm framework throughout this thesis to implement our proposed schedulers and perform our experiments. For the sake of completeness, we provide a detailed background on the Storm architecture, its components and the default scheduler.

5.1 A Storm cluster

A Storm cluster consists of three types of nodes: the master node, worker/compute nodes and ZooKeeper nodes.

- **The master node** has a daemon called *Nimbus* that is responsible for scheduling the tasks, assigning them to worker nodes and also monitoring tasks' execution.
- **A worker node** has a daemon named *Supervisor*, which starts and stops the tasks assigned by Nimbus on the worker node. Each worker node is configured with a number of ports or *slots*, where each slot can host a worker process, and each such worker process is an independent JVM. Each JVM is used to run a

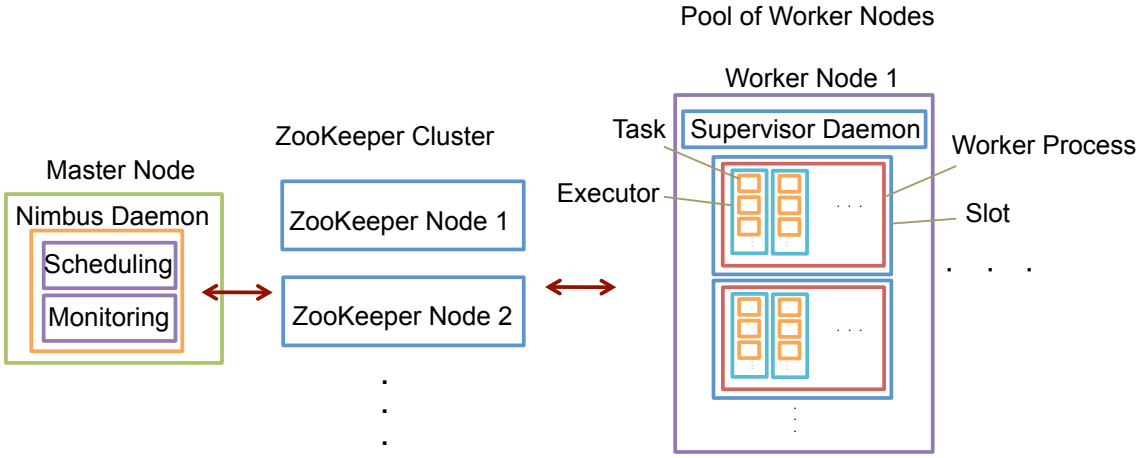


Figure 5.1: Storm architecture

number of tasks. Worker nodes contain more than one JVM so as to ensure reliable execution, as the failure of one task within a JVM will cause the entire JVM to restart, disrupting the execution of all the tasks running on that JVM.

- **ZooKeeper nodes** coordinate the communication between the master and worker nodes, with the number of ZooKeeper nodes depending on the scale of the cluster.

Figure 5.1 presents an overview of the Storm architecture.

5.2 A Storm application and its components

A streaming application in Storm is called a *topology*. It is a DAG where the vertices represent operators or *components* and the edges indicate the data flow or *streams*. There are two types of components in Storm: *spouts* and *bolts*.

A spout is the source of a data stream and emits data, while a bolt is the computational unit used to process the data, before emitting new data to the next bolt in the DAG. A stream is defined as an unbounded sequence of tuples where a tuple is a named list of values. Each component in Storm consists of a number of *executors* that can be run in parallel. In other words, each executor is an executing instance of the component's code that can be run in parallel with other executors of the same component. A Storm user specifies the number of executors per component and tasks per executor when developing the Storm topology. By default, there is one task per executor. As the number of tasks remains fixed during execution, it could be useful to have more than one task per executor, so that when new nodes are added to a clus-

ter, new executors can be created which are then assigned some of the existing tasks, increasing the parallelism degree for the running topology.

An executor is run inside a worker process, where multiple executors can run in a worker process and there can be multiple worker processes per node. The relationship between task, executor, worker process, and worker node can be seen in Figure 5.1. When a topology, consisting of spouts and bolts, is submitted to a Storm cluster, the tasks are grouped into a number of worker processes, where each node runs multiple worker processes.

5.3 Stream grouping

For every two communicating components, each having multiple tasks, Storm needs to determine which tasks of the receiving component should receive which stream coming from the emitting component. This is called *stream grouping*,[1] which controls the flow of the stream between tasks. The most common stream groupings are discussed in the following list. More stream groupings can be found in the Storm documentation.

- **Shuffle grouping:** The target task in the component receiving a tuple is selected randomly such that each task of this component receives on average the same number of tuples emitted by the source component.
- **Fields grouping:** The target task in the component receiving a tuple is selected based on a specific field in the tuple such that the tuples with the same value for that specific field go to the same task.
- **Global grouping:** The entire stream goes to the task with the lowest ID in the component receiving a tuple.
- **Direct grouping:** It is decided by the emitting component, the producer of the stream, which task of the receiving component should receive the stream.
- **All grouping:** All tasks in the receiving component receive the tuples.

To explain the concept of stream grouping, we use the word count topology in Figure 5.2 as an example. This topology consists of three components: $C1$, $C2$ and $C3$. Each component has two executors, each of which has only one task. $C1$ emits the sentences to $C2$ which splits the sentences into words. $C3$ receives the words from

[1] `http://storm.apache.org/releases/current/Concepts.html`

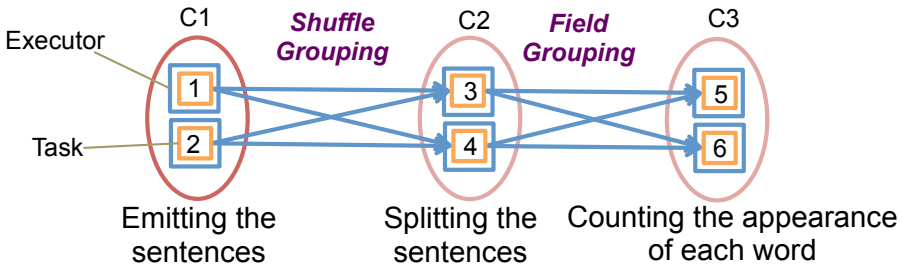


Figure 5.2: A Storm word count topology

$C2$ and counts the number of occurrences of each word. The stream grouping for $C1$ and $C2$ uses a shuffle grouping in order to partition the streams equally among $C2$'s tasks. But for $C2$ and $C3$, a fields grouping is used so that each word goes to a specific task based on its content. This ensures a complete view of the number of occurrences of each word at any time.

5.4 Storm default scheduler

The default Storm scheduler, EventScheduler, uses a round-robin algorithm to distribute the tasks across the cluster. The default scheduler iterates over the list of tasks ordered by their task number and assigns them in a round-robin fashion to the available slots such that slot 1 of each node is selected first, then slot 2 of each node and so on. Storm assigns numbers to the tasks by first sorting the components by their name alphabetically. It then assigns a number to each task, starting from one, within each component in the sorted list. For example, in Figure 5.2, the sorted components list is $\{C1, C2, C3\}$ and the tasks are numbered 1–6, as shown in the figure.

We explore how the default scheduler works by assigning the word count topology's tasks shown in Figure 5.2 to the Storm cluster shown in Figure 5.3. The average task load is defined as the average CPU load generated by each task, denoted as α. We assume that our Storm cluster has four worker nodes where slot i, node j and slot i on node j are denoted as s_i, n_j and $s_i : n_j$, respectively. n_1 , n_2 , n_3 and n_4 have the capacity of 6α, 6α, 3α and 3α, respectively. n_1 and n_2 are configured with two slots whereas n_3 and n_4 are configured with one slot. This configuration is selected for the purpose of simplifying the explanation.

The slots list, sorted by Storm, is as follows: $s_1 : n_1$, $s_1 : n_2$, $s_1 : n_3$, $s_1 : n_4$, $s_2 : n_1$, $s_2 : n_2$. The task numbers are shown in Figure 5.2. The default scheduler starts task assignment by assigning task 1, task 2, task 3 and task 4 to s_1 on n_1, n_2, n_3 and n_4

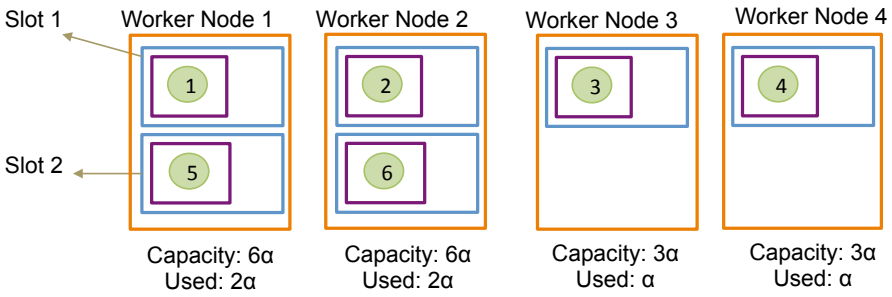


Figure 5.3: Task assignment by default scheduler of Storm

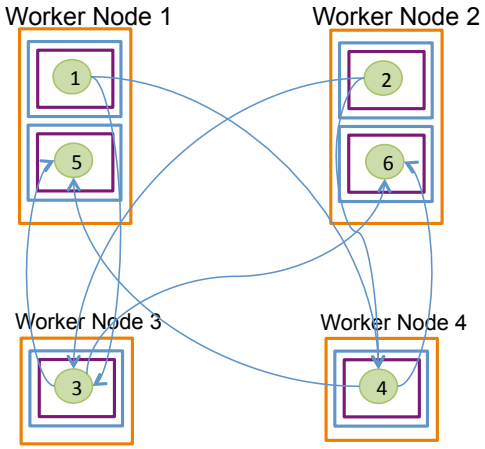


Figure 5.4: An illustration of the inter-node communication for a task assignment by default scheduler of Storm. Ideally, the number of node-crossing arrows would be as small as possible, as it represents expensive communication.

respectively. The assignment is continued by assigning task 5 and task 6 to s_2 on n_1 and n_2 respectively. Figure 5.3 shows the final assignment of the tasks to the worker nodes and Figure 5.4 illustrates the inter-node communication in our cluster.

The default scheduler is a simple and load-balanced algorithm which evenly distributes the tasks among worker nodes. However, it has a number of drawbacks.

- It does not consider the extent of heterogeneity within the cluster. As shown in Figure 5.3, n_1 and n_2 have higher capacities and could have hosted more tasks, reducing inter-node communication.

- It does not fully utilise the capacity of each node, where all of the tasks could have easily been allocated to either n_1 or n_2 in our example.

- The communication pattern between the tasks is ignored, as shown in Figure 5.4.

- The assignment of tasks to workers is static and does not respond to changes in the data transfer rate or load.

5.5 Conclusion

We have implemented our proposed schedulers, described in the later chapters, in the Apache Storm framework, an open-source, real-time, distributed, scalable and reliable data stream processing system. In this chapter, we have provided some background on Storm for the sake of completeness, helping the reader better understand our implementations and experiments.

Chapter 6

Experimental settings

In this chapter, we describe the experimental setup and applications used for evaluating our three proposed schedulers, discussed in later chapters. As each of the schedulers uses the same experimental settings, we present all of the details here. It may be useful to return to this chapter when reading the experiments for each scheduler to clarify the details regarding any setting. The remainder of this chapter is outlined as follows. We present the micro-benchmarks which are used to evaluate the communication cost of each scheduler, comparing it to the optimal scheduler formulated in Chapter 4, which is referred to as a theoretically optimal scheduler throughout the rest of this thesis. We further discuss how these micro-benchmarks can be configured to represent I/O- and CPU-intensive workloads. Finally, we describe two real-world applications used to evaluate the system throughput and latency of each scheduler.

6.1 Experimental setup

We implement our proposed schedulers in Apache Storm 1.1.1, running on a Storm cluster, configured with one master node, one ZooKeeper node and eight worker nodes. Each node runs Ubuntu 16.04 LTS.

Typical cluster deployments begin as homogeneous clusters, but as they grow over time with the addition of new hardware, they may become heterogeneous. For this reason, we evaluate our schedulers on both homogeneous and heterogeneous configurations. To achieve this, we run a VirtualBox VM on each machine, which allow us to change the hardware configuration, with high and low capacity worker nodes, providing more configurability. We initially validated a homogeneous configuration by comparing a bare metal and VMs configuration, finding both configurations achieved

the same performance. The high capacity nodes are configured with 4 cores and 4GiB of RAM with four slots per node, whereas the low capacity nodes have 2 cores and 2GiB of RAM with two slots per node. This node configurability allows us to use both homogeneous and heterogeneous clusters. Although a heterogeneous environment typically refers to different models of hardware with different configurations, we argue that the same hardware, where not all resources are available, can also be considered a heterogeneous environment. Each node has a 2.7GHz Intel Core i5-3330S processor and is connected to a 1Gbps network.

To prevent any node from becoming overloaded, each scheduler will assign tasks such that average CPU usage does not exceed 80% for the node. T in Equation 7.3 (see Section 7.3.4) provides a balance between the desire for more tasks per worker, and the reliability of execution. The failure of any single task within a worker will cause all of the other tasks within that worker to be restarted. As a result, any failure will cause tuples to be lost, where having fewer tasks per worker will reduce these losses. Therefore, we set the user-defined parameter of T to 5, as this had less significant losses. It is also worth noting that the results presented in this thesis do not include any runs which had task failures.

We compare our schedulers with Aniello *et al.* (2013)'s 'Online scheduler' (referred to as 'OLS' in this thesis) and R-Storm (Peng *et al.*, 2015), previously discussed in Chapter 3. Unlike many other Storm schedulers, the OLS and R-Storm implementations are publicly available, allowing for a fair comparison. Furthermore, these two schedulers have the same optimisation goal as our schedulers, as presented in Chapter 3, which is to reduce the inter-node communication. Our evaluation criteria for the algorithms are the latency and system throughput. To measure these values, we record the average throughput, defined as the average number of tuples executed in each bolt's task per 10 second period, and average execute latency for each task as our performance metrics. We use three micro-benchmarks and two real-world applications, with real data, described in the following sections, for our evaluation.

Each experimental application is run ten times for 650 seconds. Applications are initially run with round-robin scheduling for 50-60s while the proposed schedulers monitor the execution. Rescheduling is then performed once for each experimental run, where Storm migrates the tasks to new cluster nodes. Currently, Storm uses a simple method of task migration, where the execution is stopped, allowing the tasks to be moved, before restarting execution with the new configuration. As a result of this delay, we present all experimental results starting at 150s, after rescheduling has completed. In

our future work, we will investigate methods for smooth task migration, which does not stop the entire execution, in order to reduce the overhead of rescheduling. Further, we will also continue work on run-time performance monitoring, investigating how workload characteristics change during execution and when rescheduling should be performed.

6.2 Micro-benchmarks

In this section, we describe three layouts that represent common shapes of a streaming application, first presented in (Peng *et al.*, 2015). Each of these layouts represents a different congestion pattern: linear, diamond and star, described as follows:

- **Linear:** This is one of the most common types of topologies, shown in Figure 6.1a, which consists of a number of operators, where tuples are passed from one operator to the next.

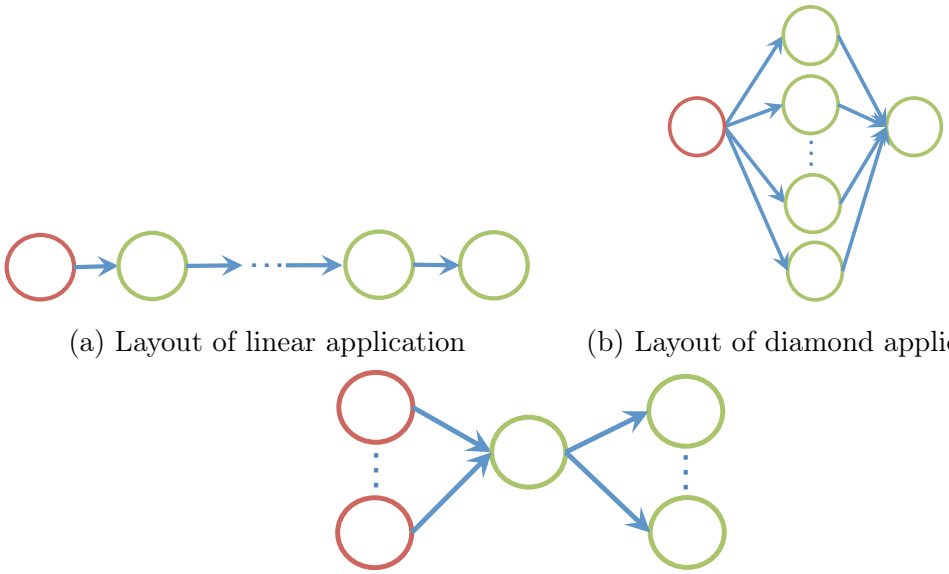
- **Diamond:** The layout of this micro-benchmark has a diamond shape, shown in Figure 6.1b, where one source emits tuples to multiple operators. Each operator then passes these tuples to a single sink operator.

- **Star:** This more complicated layout has a star shape, shown in Figure 6.1c, where multiple sources emit tuples to one operator. This operator then emits the received tuples to multiple sinks, passing them along.

6.2.1 Comparing our proposed schedulers with a theoretically optimal scheduler

We compare the communication cost of each scheduler with a theoretically optimal scheduler using the three micro-benchmarks. The linear micro-benchmark is configured with two tasks per operator, while in the diamond layout, the source and sink operators have four tasks, with all other operators having two tasks. In the star layout, each of the source and sink operators has two tasks and the middle operator has four tasks.

We evaluate different problem sizes by increasing the number of operators in each layout, with each new operator containing two tasks. For the linear layout, each new operator is added to the end of the layout, whereas for the diamond, new operators are



(a) Layout of linear application

(b) Layout of diamond application

(c) Layout of star application

Figure 6.1: Micro-benchmark applications (Peng *et al.*, 2015)

Table 6.1: The node configuration for the homogeneous and heterogeneous clusters used in the evaluation of scheduler communication cost

Setting	**Number of nodes**	**Node capacity**
Homogeneous	10 nodes	10 nodes with capacity of 4
Heterogeneous	10 nodes	3 nodes with capacity of 6 3 nodes with capacity of 4 4 nodes with capacity of 2

added to the middle of the layout which communicate with the source and sink operators. Finally, for the star layout, operators are added as either a source or sink, with each new operator alternating between the two, ensuring the layout remains balanced. We configure each of the micro-benchmarks with a problem size of 10 tasks to begin with, increasing the problem size to 32. To simplify the experiments, for illustrative purposes, we assume that the data transfer rate between every two communicating tasks is 1 normalised unit per second, and the load is uniform across all tasks. The simple configuration of these examples is for illustrative purposes, and to ensure the corresponding discussion is clear and easy to follow. It is also worth noting that the schedulers are capable of working with more complex configurations, which do not have a uniform task load or communication.

We run our experiments on both a homogeneous and heterogeneous cluster config-

uration, described in Table 6.1. The homogeneous cluster consists of 10 nodes, each with a capacity of 4, where the capacity of a node is defined as the average number of tasks that can be assigned to the node. The heterogeneous cluster contains 10 nodes, made up of three different node types, with capacities of 6, 4 and 2; there are 3, 3 and 4 nodes of each capacity respectively.

We use IBM CPLEX Studio 12.7[1] as our optimisation software as it provides state-of-the-art implementations of linear, integer and mixed-integer linear optimisations (Kalyvianaki *et al.*, 2011). We run our experiments to find an optimal solution on a 64-core server with four 16-core AMD Opteron 6276 processors, running at 2.3GHz, with 512GiB of RAM. We require such a large server as it becomes increasingly computationally expensive to find the optimal solution for larger problem sizes. In comparison, to find the communication cost for our proposed schedulers, we can use any commodity hardware as they do not need any specific hardware requirements. We run our experiments for our proposed schedulers on a system with 8GiB of RAM which has a 3.2GHz Intel Core i5-4570 processor with four cores.

In each of the experiments, we present the cost, which is a measure of the communication cost between nodes, previously seen in Equation 4.3. We refer to the solutions found by CPLEX as "optimal" throughout this thesis as done in other work (Wolf *et al.*, 2008; Cardellini *et al.*, 2016). Further, based on a limited brute force evaluation of small problem sizes, we found that CPLEX results were indeed optimal for those cases.[2]

6.2.2 I/O-intensive and CPU-intensive

For this evaluation, we configure each of the spouts and bolts in the linear and diamond micro-benchmarks to have eight tasks. The star micro-benchmark is configured with four and eight tasks for the spout and bolts respectively. Each micro-benchmark is run in two different configurations: I/O-intensive and CPU-intensive, described as follows.

In I/O-intensive configuration, the throughput of the system is limited by the amount of communication between the nodes. We reduce the workload of each bolt by slowing the rate of the spout, so that each bolt has little processing to do, causing processing time to be limited by the network latency. In CPU-Intensive configuration, the throughput of the system is limited by the CPU utilisation of each node. We increase

[1] `https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`

[2] A more extensive brute force evaluation is infeasible due to the time complexity of the problem.

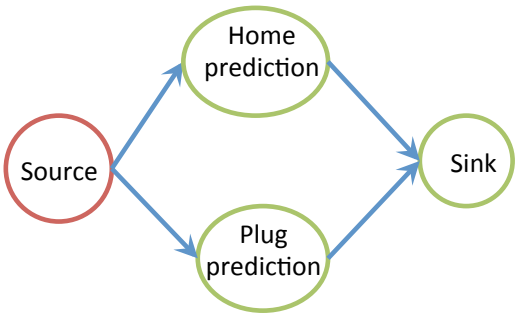


Figure 6.2: Load prediction application

the workload of each bolt by supplying tuples at a faster rate, ensuring the bolts are fully loaded, resulting in a high CPU load.

6.3 Real-world workloads

In this section, we describe two real-world applications, which were originally used in the DEBS Grand Challenge contests.[3] The DEBS Grand Challenge is a series of competitions where researchers solve a real-world problem, using real data, with a distributed and event-based system.

6.3.1 Load prediction application in smart homes

This topology is based on the first query of the DEBS 2014 Grand Challenge.[4] The data is collected from 2,125 smart plugs, deployed across 40 houses in Germany, during September 2013. The smart plugs are connected between a device/appliance and the wall outlet, and contains two sensors for measuring the power of each device, roughly once a second. Each smart plug is identified in a hierarchical structure such that the top entity is the house with a unique ID. Each house can have a number of households with unique household IDs within that house. Also, each household has a number of smart plugs with a unique ID within the household.

Each record in the dataset contains 7 fields: id, timestamp, value, property, plug_id, household_id, house_id. id is the unique identifier for each record in the dataset, while the timestamp is the timestamp of the measurement or event. All the events are sorted by timestamp and the events with the same timestamp are sorted randomly. The value field is the measurement value and property indicates the type of value field: work or

[3] http://debs.org/grand-challenges/

[4] http://debs.org/debs-2014-smart-homes/

load, where load is the current load measured in Watts, and the work is the accumulated work since reset, measured in KWh. However, because work values are cumulative and a change is not visible for small amounts of energy, first query of the Grand Challenge uses load and discards the work values.

The query predicts the future load for each house and each individual smart plug based on current load measurements and a slice-based prediction model built on historical data. More specifically, the load is predicted for the two time slices from the current time slice. The model is based on the average load of the current time slice and the median of the average load for the same time slices over the past days.

In the model presented for the Grand Challenge, the whole period of time for which data was collected is divided into N slices, $s_i, i >= 0$ with each having a size of $|s|$ seconds. The start of each slice s_i is calculated as $t\_start + i \times |s|$ where t_start is the very first timestamp in the dataset. The load value for slice s_{i+2} is predicted as: $L(s_{i+2}) = (L(s_i) + median(L(s_j)|j = i + 2 - n \times k))/2$, where k is the number of slices in a 24 hour period, n is a natural number with values between 1 and $floor((i+2)/k)$. The slice size in our application is 15 minutes. The streaming application implementing this query has a diamond shape, shown in Figure 6.2, with one spout, two intermediate bolts and one sink bolt described as follows:

- Source spout: The spout splits the comma separated records and sends the data to each of the load prediction bolts with shuffle grouping.
- Plug Prediction bolt: At the start of each slice, the sum of the load for all the events with a timestamp within the slice is calculated. The average load for the current time slice is calculated at the end of the slice and kept in a data structure to calculate the median for future predictions. Then, the predicted load for the two time slices from the current time slice is calculated, using the current average load and median of the average load for the same time slices over the past days. The output is sent to the sink bolt with fields grouping.
- House Prediction bolt: For each house, the predicted load is calculated as the sum of the predicted load for all of the smart plugs within a house. The output is sent to the sink bolt with fields grouping.
- Sink bolt: A sink bolt formats the output such that depending on the tuples received by the bolt, it either emits the house prediction value along with house ID and predicted time or the plug prediction value with hierarchical identifier for the plug and predicted time.

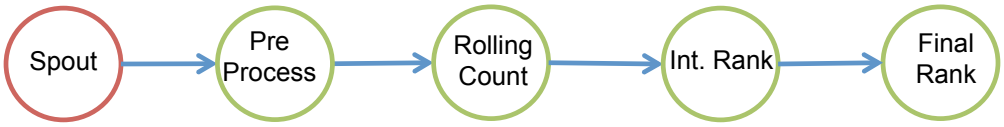


Figure 6.3: Top frequent routes application for NYC taxi data

Our application is configured with a time slice of 15 minutes, with the spout and each of the bolts having 10 tasks. We use a Redis server to store the smart plug records.

6.3.2 Top frequent routes application for NYC taxi data

This topology is based on the first query of the DEBS 2015 Grand Challenge.[5] The query is to find the top 10 most frequent routes of New York taxis for the last 30 minutes using the 2013 dataset. This is a similar problem to the Twitter top trending topics, previously described in Chapters 1 and 4, where the top 10 tweets from the last 30 minutes are tracked. As the Twitter API does not provide full access to the Twitter firehose, only a limited stream of tweets was available. The NYC taxi problem has the same topology, and is used as a substitute implementation for the Twitter problem used for illustrative purposes earlier in this thesis.

The NYC taxi dataset was released under the FOIL (Freedom Of Information Law), and has been made publicly available by Chris Whong.[6] The taxi trip records include pickup and drop off time and location, trip distance, duration, fare and tip amount. New York city is mapped to a 300×300 grid where each cell represents a $500\text{m} \times 500\text{m}$ location in New York. The top left cell is numbered 1.1, where the centre of this cell is located at 41.474937, -74.913585 (in Barryville) (latitude and longitude coordinates). The cell numbers increase towards east and south which means the bottom right cell is numbered 300.300. All the trips starting or ending outside the area covered by the grid are considered outliers and are not processed. A route is represented by the starting and ending cell numbers, where two trips with the same start and end locations are considered to have the same route. The layout of this topology has a linear shape with one spout and four bolts, shown in Figure 6.3, described as follows:

- Source spout: The spout reads the records of each taxi trip from the dataset and sends the data to the Preprocess bolt with shuffle grouping.

- Preprocess bolt: The Preprocess bolt processes each trip record in order to find

[5] `http://www.debs2015.org/call-grand-challenge.html`
[6] `http://chriswhong.com/open-data/foil_nyc_taxi/`

the start and end cell numbers based on longitude and latitude coordinates of the pickup and drop off locations. This bolt then emits the routes to Rolling Count bolt with shuffle grouping.

- Rolling Count bolt: This bolt counts the occurrence of each route using a rolling counter, implemented with a sliding window. The data is then passed to the Intermediate Rank bolt with fields grouping.
- Intermediate Rank bolt: This bolt has multiple tasks and is used to distribute the load coming from the Rolling Count bolt and reduce it before emitting the results to Final Rank bolt with global grouping. Each task of the Intermediate Rank bolt finds 10 top frequent routes for a specified window.
- Final Rank bolt: This bolt has only one task, aggregating the incoming intermediate rankings from the Intermediate Rank bolt into a final rank.

The numbers of tasks for each of the Source, Preprocess, Rolling Count, Intermediate Rank and Final Rank bolts are 16, 16, 8, 4 and 1 respectively. We use a Redis server to store the trip records. We simulate a replay of the taxi data, by setting a simulation time that is a constant ratio with real time. This ratio is set so that 1 minute is equal to 0.1 second of time in our experiment—thus the sliding window in Rolling Count bolt is 3 seconds.

6.4 Conclusion

In this chapter, we have described the settings used in the evaluation of our proposed schedulers, which will be presented in later chapters. By outlining the experimental settings in this single chapter, we are able to avoid unnecessary repetition of these details in each chapter.

Chapter 7

P-Scheduler

In this chapter, we present P-Scheduler, a partition-based scheduler for DAG-based data stream processing systems. This chapter is organised as follows. Section 7.1 provides an overview of P-Scheduler. In Section 7.2, a brief background on graph partitioning is presented. Then, the P-Scheduler algorithm is described in Section 7.3, followed by an example in Section 7.4. Section 7.5 presents a comparison with an optimal scheduler using three micro-benchmarks, previously described in Section 6.2. Section 7.6 evaluates P-Scheduler with two real-world applications, previously presented in Section 6.3, with a detailed discussion in Section 7.7. Finally, Section 7.8 concludes the chapter.

7.1 Introduction

P-Scheduler is an algorithm for DAG-based homogeneous stream processing systems that employs graph partitioning algorithms in a hierarchical manner. Each edge in the streaming application graph is weighted with the data transfer rate between the communicating tasks and each vertex is weighted with the load for the corresponding task. P-Scheduler has two levels of scheduling: In the first level, the number of compute nodes required to run the streaming application is calculated based on the total application load. Then, the entire application graph is partitioned into the number of required compute nodes. This ensures that P-Scheduler utilises each compute node, helping reduce the inter-node communication. In the second level of scheduling, P-Scheduler assigns highly communicating tasks to the same worker process in order to minimise the communication between the workers within a compute node. Our contributions in this chapter are summarised as follows:

- We use a method to reduce the inter-node traffic by first calculating the number of nodes that are required to run a streaming application in a data stream processing system.

- We present P-Scheduler, an online hierarchical scheduling algorithm for streaming applications, which partitions the graph based on the number of required nodes, before further partitioning each sub-graph into a number of workers. This hierarchical approach maximises the communication efficiency by assigning highly communicating tasks to the same worker or node.

- The communication cost of P-Scheduler is evaluated by comparing it to a theoretically optimal scheduler, implemented in CPLEX, when run on three micro-benchmarks, each representing a different communication pattern, previously discussed in Section 6.2. The results show that P-Scheduler can achieve results that are close to optimal in a homogeneous cluster configuration.

- P-Scheduler is implemented in Apache Storm 1.1.1 and through experimental results, we show that P-Scheduler can outperform state-of-the-art R-Storm (Peng *et al.*, 2015) and OLS (Aniello *et al.*, 2013), previously described in Chapter 3. The results show that P-Scheduler outperforms OLS, increasing throughput by 12–86% and R-Storm by 32% for the real-world applications.

7.2 Graph partitioning

Graph partitioning has many applications such as in VLSI design, spatial databases, transportation management, data mining and task scheduling. The K-way graph partitioning problem is to decompose the vertices of a graph G into k roughly equal parts, such that the number of interconnecting edges is minimised. This problem can be extended to a graph with weighted edges and vertices where the goal is to partition the vertices into k disjoint subsets such that the sum of the vertex-weights in each subset is the same, and the total weight of the edges whose incident vertices belong to different subsets is minimised.

While the problem of graph partitioning is NP-complete, there are many algorithms that use heuristics and approximation to find a reasonably good partition (Karypis and Kumar, 1998).

An efficient heuristic approach for graph partitioning is K-way multilevel partitioning, which consists of three phases: graph coarsening, initial partitioning into k parts,

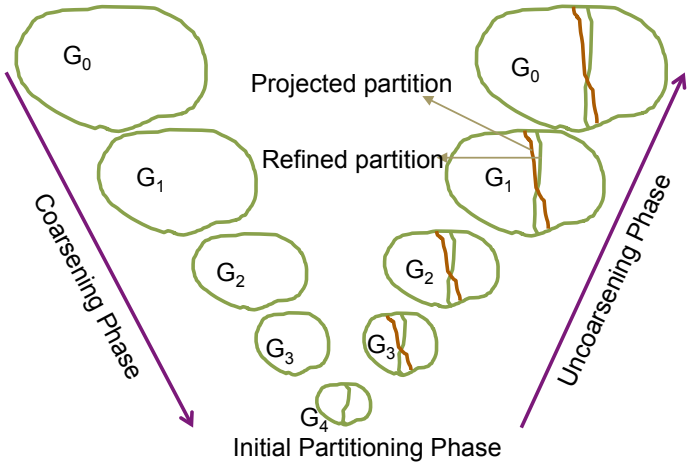


Figure 7.1: Illustration of multilevel graph partitioning (Karypis and Kumar, 1998)

and uncoarsening. During the coarsening phase, the size of the graph is decreased in a series of steps by collapsing together a maximal size set of adjacent pairs of vertices. In this phase, the output of each step is presented to the next step, so the size of the graph is gradually decreased. Then, the smallest graph is fed into the initial partitioning phase where the K-way partitioning algorithm is applied to find a set of k partitions. Finally, uncoarsening is performed on the graph to obtain the original graph. It iteratively projects the smallest graph to a larger graph by recovering the pairs of vertices that were collapsed (Karypis and Kumar, 1998). The phases for the multilevel graph partitioning described above, are shown in Figure 7.1.

METIS[1] is an open source software package developed in the Karypis Lab at the University of Minnesota. It can partition large irregular graphs and large meshes, in addition to computing the fill-reducing orderings of sparse matrices. METIS includes an implementation of the K-way multilevel partitioning proposed in (Karypis and Kumar, 1998), which can produce high quality partitions. We use METIS in our research due to its efficiency in finding good partitions.

7.3 P-Scheduler algorithm

In this section, we describe the P-Scheduler algorithm. The goal is to assign the highly communicating tasks of a streaming application to the same worker process or node. To achieve this, a two-level scheduling approach is applied to assign the tasks to compute nodes. At the first level, it is decided which tasks should be assigned to the

[1] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

same compute node. The second level of the scheduler decides which tasks within the same node should be in the same worker process. Using a two-level graph partitioning method, we can minimise the communication between nodes and worker processes, improving performance.

P-Scheduler consists of four main steps as follows.

1. **Monitoring:** To facilitate the task scheduling, P-Scheduler needs to initially profile the task graph execution, measuring the tasks' load and data transfer rate between the communicating tasks.

2. **Constructing a weighted graph:** The online profile from the previous step is then used to build a weighted graph. This graph is updated with new weights when rescheduling.

3. **First level of scheduling:** At this level, P-Scheduler determines the number of compute nodes required to run the application, n, (see Equation 7.1 shortly) and then partitions the weighted graph, from the previous step, into n parts. By assigning each partition to a compute node, the highly communicating tasks are co-located within the same node.

4. **Second level of scheduling:** At this level, P-Scheduler finds the number of worker processes required for each node, w, (See Equation 7.3 shortly), before dividing the sub-graph, (found by the first-level of scheduling), into w such that highly communicating tasks are grouped together into a worker, reducing inter-worker communication.

The formal description of the two levels of scheduling are presented as follows.

- **First level of scheduling:** Given the dataflow graph of the application $G = (V, E)$, where V is the set of vertices (tasks) and E is the set of weighted edges (dataflow between tasks with data transfer rate), find n first-level subsets $V_1, V_2, ..., V_n$ of V such that:
  - $\bigcup_{i=1}^{n} V_i = V$
  - $|V_i| \approx \frac{|V|}{n}$, $i \in 1, 2, ..., n$
  - $V_i \bigcap V_j = \emptyset$ for $i \neq j$
  - The sum of weights of edges crossing between first-level subsets V_i and V_j, $i \neq j$ is minimised.

Table 7.1: Notation for P-Scheduler algorithm

Symbol	Meaning
op	Online profile
g	Weighted graph
sbg	A sub-graph, consisting of highly communicating tasks
t	Number of tasks within sbg
T	Max number of tasks per worker
w	Required number of workers
n	Required number of nodes

- **Second level of scheduling:** For each first-level subset V_i, find w second-level subsets $V_{i1}, V_{i2}, ..., V_{iw}$, using a similar approach to the first-level, such that:
  - $\bigcup_{j=1}^{w} V_{ij} = V_i$
  - $|V_{ij}| \approx \frac{|V_i|}{w}$, $j \in 1, 2, ..., w$
  - $V_{ij} \bigcap V_{ik} = \emptyset$ for $j \neq k$
  - The sum of weights of edges crossing between second-level subsets V_{ij} and V_{ik}, $j \neq k$ is minimised.

The pseudo-code for P-Scheduler is given in Algorithm 1, which shows the two-level partitioning algorithm used. Table 7.1 shows the notation used in this chapter for the P-Scheduler algorithm. Figure 7.2 shows the interaction between P-Scheduler, METIS, monitoring log and worker nodes in a Storm cluster. In the following, each step is discussed in detail.

7.3.1 Monitoring

As the first step, when a streaming application is first submitted, it is initially run for a period of time using round-robin scheduling, while P-Scheduler monitors the execution. This involves measuring the data transfer rate between each of the task pairs, providing a profile of all communications and also the task loads. The collected values are stored regularly in a monitoring log which P-Scheduler can read periodically when rescheduling.

Algorithm 1 Pseudo-code for P-Scheduler Algorithm

```
1: op = MONITORING(Streaming_Application)   ▷ Online profile for the streaming application
2: g = CONSTRUCT_WEIGHTED_GRAPH(op) ▷ Constructing a weighted streaming application graph
3: SCHEDULE(g, nodes)
4:
5: function SCHEDULE(g, nodes)
6:     sbg_list ← FIRST_LEVEL(g, nodes)
7:     for each sbg in sbg_list do
8:         SECOND_LEVEL(sbg, T, nodes )
9:     end for
10: end function
11:
12: function FIRST_LEVEL(g, nodes)
13:     U ← CALCULATE_TOTAL_LOAD(g)
14:     C ← CALCULATE_CAPACITY(nodes)
15:     n ← ⌈U/C⌉
16:     sbg_list ← GRAPH_PARTITION(g, n)   ▷ Partition g into n sub-graphs using METIS
17:     return sbg_list
18: end function
19:
20: function SECOND_LEVEL(sbg, T, nodes)
21:     t ← FIND_TASK_SIZE(sbg)
22:     w ← ⌈t/T⌉
23:     parts ← GRAPH_PARTITION(sbg,w)   ▷ Partition sbg into w parts using METIS
24:     node ← SELECT_NODE(nodes)
25:     ASSIGN(parts, node)   ▷ Assign each part in parts to one worker process on node
26: end function
```

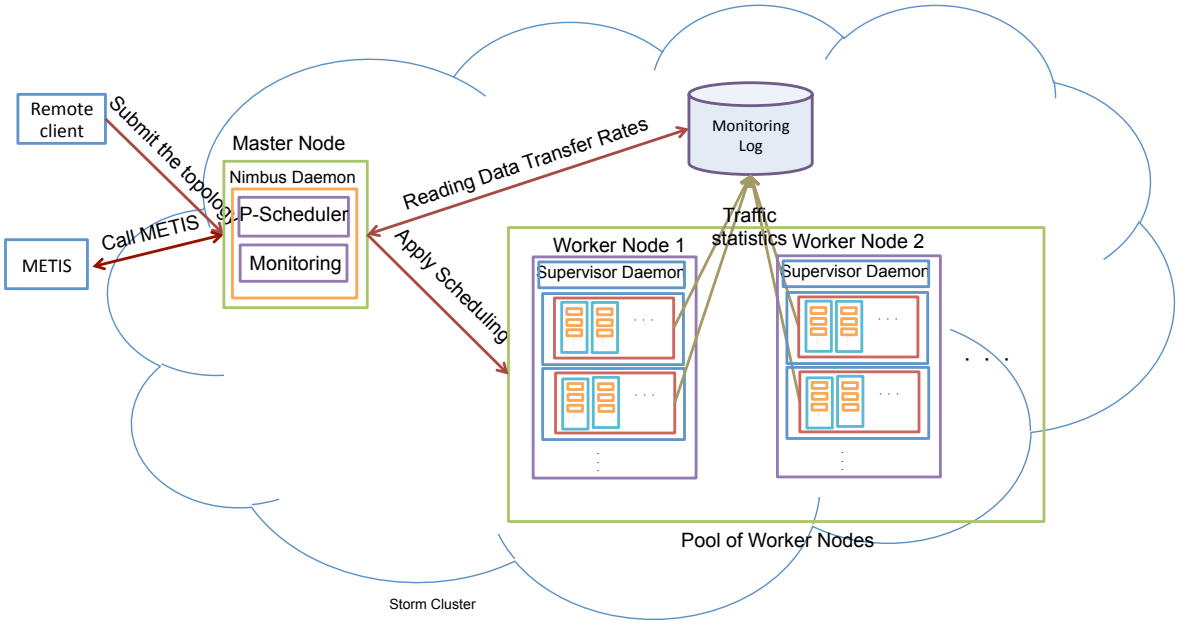


Figure 7.2: The interaction between P-Scheduler, METIS, monitoring log and worker nodes in a Storm cluster

7.3.2 Constructing a weighted graph

P-Scheduler constructs a weighted graph using the online profile collected from the monitoring step. Each vertex represents a task and each edge represents the dataflow between two communicating tasks. The weight of each vertex is the load of each task. Each edge is weighted with the data transfer rate of the communicating tasks. The weighted graph has the advantage of having a view of the amount of communication for all of the tasks. Therefore, highly communicating tasks can be found and assigned to the same compute node. When rescheduling, vertices and edges are updated using the online profile from the monitoring step.

7.3.3 First level of scheduling

The first step in reducing the inter-node communication of an application is to reduce the number of nodes required to execute the streaming application. Therefore, we use cluster consolidation to reduce the number of nodes, and fully utilise each node. By assuming that the cluster is homogeneous and all of the nodes have the same capacity, P-Scheduler calculates the number of required nodes, n, based on the following equation:

$$n = \left\lceil \frac{U}{C} \right\rceil \tag{7.1}$$

where U is the sum of CPU usage for the tasks of the streaming application and C is the capacity threshold that each node can effectively operate at without becoming overloaded. C is calculated as follows.

$$C = S_c \times \alpha, \text{ for } 0 < \alpha < 1 \tag{7.2}$$

where S_c is the overall CPU speed of each node considering all its cores and α is a system dependent parameter defined by the DSPS administrator who decides the parameter according to the usage of the node.

The parameter n is updated in each scheduling period with the changing load of the streaming application, collected from the monitoring log.

At this point, P-Scheduler needs to group highly communicating tasks into n nodes such that inter-node communication is minimised. This problem can be viewed as partitioning the weighted application graph into n sub-graphs. To achieve this, P-Scheduler uses well-studied and performant K-way partitioning proposed in (Karypis and Kumar, 1998), which has commonly been used for graph partitioning (Tanaka and Tatebe, 2012; Wang *et al.*, 2014; Tian *et al.*, 2013; Huang *et al.*, 2011; Low *et al.*, 2012). As mentioned in Section 7.2, K-way partitioning divides the vertices of the weighted graph g into n roughly equal parts, such that the number of edge cuts between the compute nodes is minimised. Finally, each of the n sub-graph partitions are assigned to a compute node, before performing the second level of scheduling.

7.3.4 Second level of scheduling

In data stream processing systems, such as Apache Storm, each of the compute nodes contains one or more worker processes, where each of these workers run one or more tasks. The scheduler needs to determine how tasks should be assigned to each worker, such that inter-worker communication is minimised. Therefore, P-Scheduler needs to assign the tasks to workers within each node based upon the task communication for the sub-graph. Therefore, the second level of scheduling determines which tasks should be assigned to the same worker process.

Finding the number of workers per node is hard to know *a priori*. In determining the number of workers per compute node, a balance needs to be struck between performance and reliability. That is, assigning all of the tasks within a compute node to a single

worker will be less reliable, as a single task failure will cause all tasks within the worker to be interrupted and restarted. Whereas, assigning fewer tasks per worker will require more workers, increasing the inter-worker communication.

Therefore, P-Scheduler uses a simple heuristic by introducing a threshold T, for the maximum number of tasks per worker, which sets a balance between better tolerating failures, and reducing inter-worker communication. The value of T is empirically determined, by observing which value gives reliable performance results.

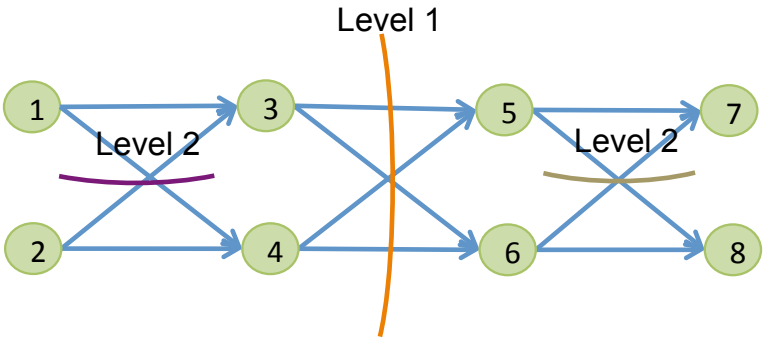
To find the number of workers, denoted as w, for each compute node, the number of tasks to be assigned to a compute node, denoted as t, is divided by T, for the given streaming application:

$$w = \left\lceil \frac{t}{T} \right\rceil \tag{7.3}$$

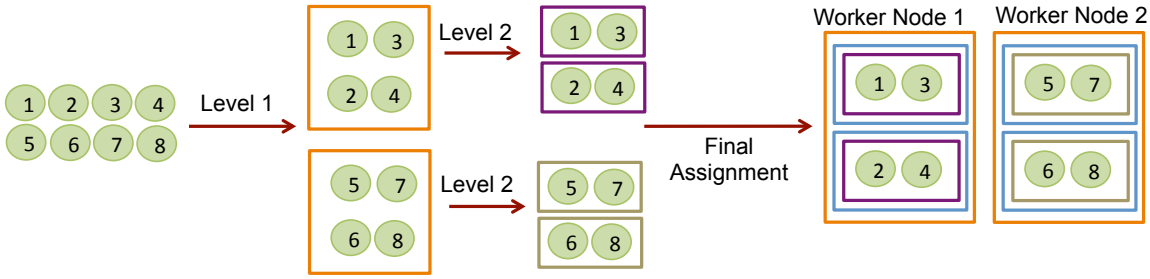
After finding w for each compute node, P-Scheduler finds highly communicating tasks to be assigned to each worker by employing K-way partitioning. The partitions are of roughly equal size, which means each part may contain fewer than T tasks. The number of tasks will remain close to T, but will not exceed this threshold value. Each part is then assigned to a worker, where the most highly communicating tasks are grouped together, minimising inter-worker communication.

7.4 An example of task assignment by P-Scheduler

Figure 7.3 demonstrates the two-level graph partitioning of P-Scheduler with an example. For simplicity, we assume that the vertices and edges have a weight of 1. We also assume that the number of required nodes n has been calculated to be two, based on Formula 7.1. We further assume that the number of tasks per worker process, T, described in Section 7.3.4, is set to two. First, the application graph is partitioned into two sub-graphs where one sub-graph contains the tasks $1, 2, 3, 4$ and the other one contains the tasks $5, 6, 7, 8$. The number of required worker processes per node is calculated to be two, based on Formula 7.3. Therefore, in the second level of scheduling each first-level sub-graph is further partitioned into two parts. For example, tasks $1, 3$ are grouped together to be assigned to the same worker. The final assignment is shown in Figure 7.3b. From the figure, it can be seen that, by using P-Scheduler, the task pairs with the most communications are in the same worker process or the same node.



(a) A partitioned streaming application. We assume that the number of required number of nodes and also workers per nodes have been calculated to be 2.



(b) Final task assignment by P-Scheduler

Figure 7.3: An example of task assignment by P-Scheduler

7.5 P-Scheduler vs. optimal scheduler

In this section, the communication cost and resolution time of P-Scheduler are compared with a theoretically optimal scheduler. The communication cost for P-Scheduler and the optimal scheduler running the linear, diamond and star micro-benchmarks, which were previously discussed in Section 6.2, are shown in Figures 7.4, 7.5 and 7.6 respectively. While we are able to determine an optimal schedule in this evaluation, this is only possible for small problem sizes, as the computational complexity increases with the problem size. Therefore, it become impractical to find the optimal solution for larger, more complex problems. As we are evaluating how close to optimal the results of P-Scheduler are, the resolution time was not a concern.

Figure 7.4 shows the communication cost for the linear layout, where P-Scheduler is able to match the results of the optimal scheduler in the homogeneous configuration. This is due to both schedulers employing a similar strategy, where tasks from multiple operators are assigned to the same node.

Figure 7.5 shows the communication cost for diamond layout, where it can be seen

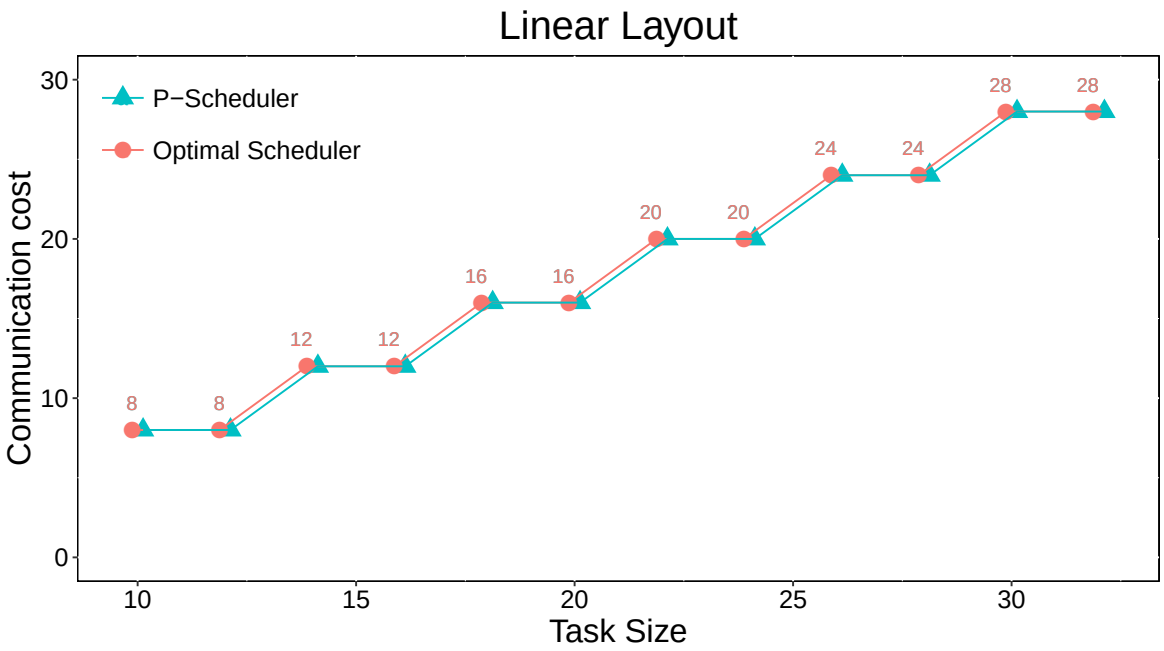


Figure 7.4: P-Scheduler vs. optimal scheduler for linear layout

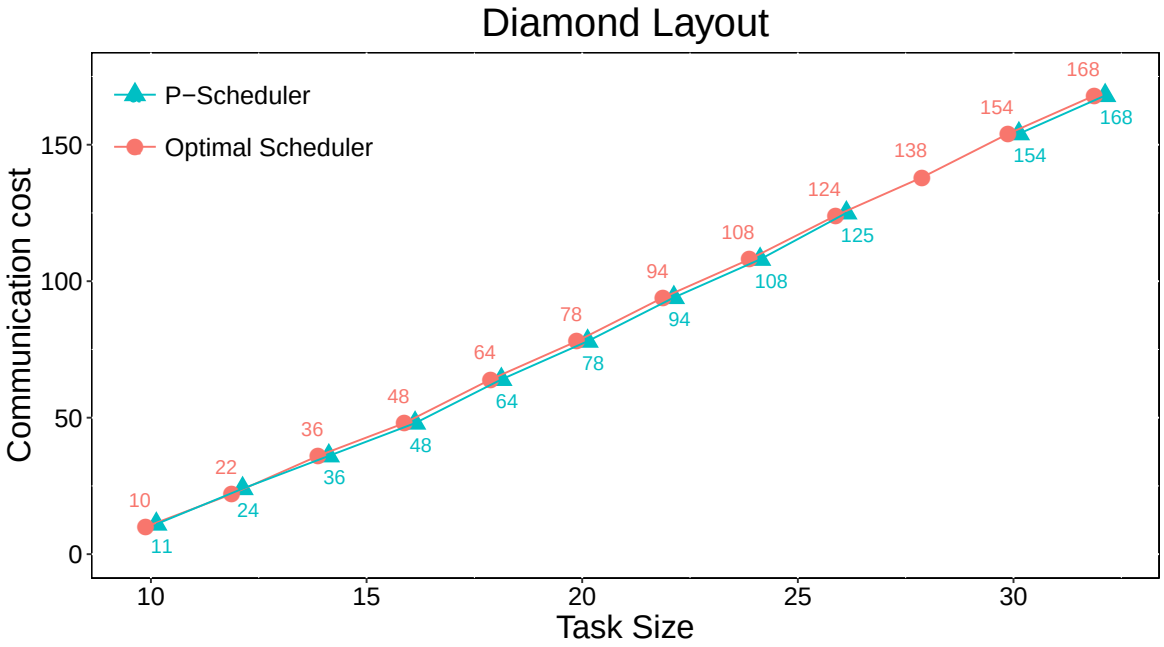


Figure 7.5: P-Scheduler vs. optimal scheduler for diamond layout

that P-Scheduler can match the optimal scheduler results for all but three task sizes. For a task sizes of 10 and 22, P-Scheduler has a communication cost of 11 and 125,

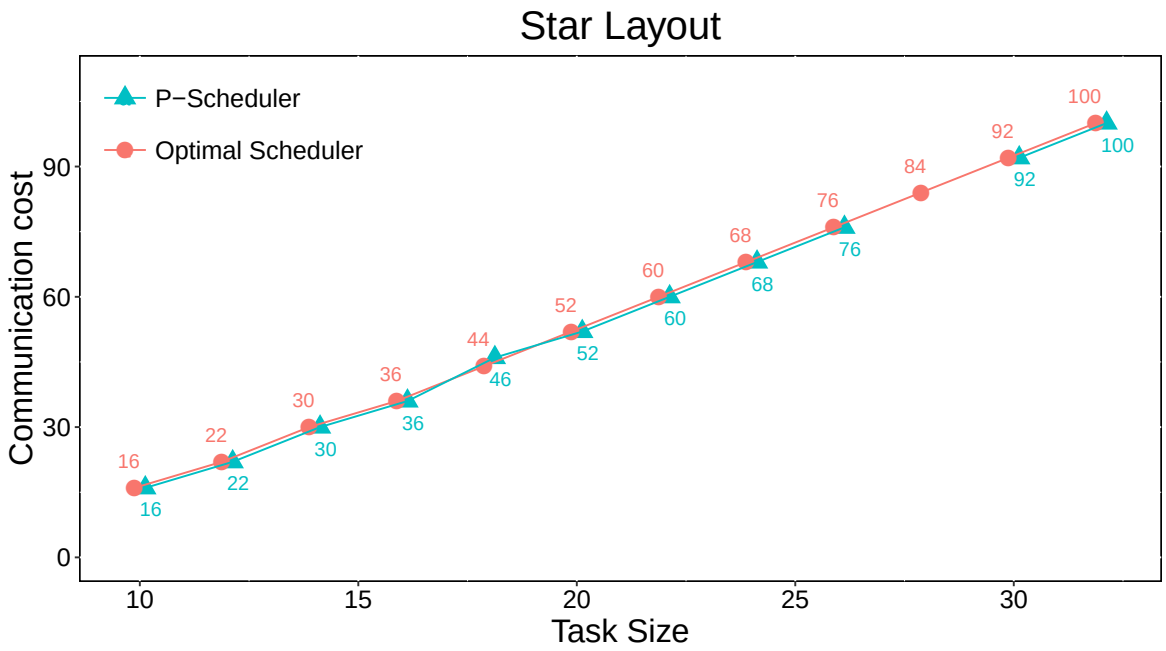


Figure 7.6: P-Scheduler vs. optimal scheduler for star layout

while the optimal scheduler is slight lower with 10 and 124 respectively. For a task size of 28, P-Scheduler is unable to find a solution, as not all of the partitions were of size 4. This sometimes occurs with K-way partitioning, as it is not always able to find partitions of the size we want, with the resulting partitions being either slightly larger or smaller. In this case, the partition size found by K-way partitioning was 5, but by changing the capacity of a single node to size 5, a solution was found with a communication cost of 137 for optimal and 138 for P-Scheduler.

Similar results can be seen in Figure 7.6 for the star layout. P-Scheduler can match optimal for all task sizes except 18, where optimal is 44 and P-Scheduler is 46. For 28 tasks, P-Scheduler was unable to find a solution with the available node capacities, as one partition was of size 5. A solution was found by adjusting the capacity of a single node, where both P-Scheduler and optimal had a communication cost of 84.

It can also be seen from Figures 7.4, 7.5 and 7.6 that the communication costs of both schedulers for diamond and star layouts are bigger than linear. The reason is that in the linear layout, more communicating task can be grouped and placed in the same node because of the shape of the layout when compared to the other two layouts. In the diamond layout, the source and sink operators only have 4 tasks each, which means that the number of communicating tasks that are able to be placed in the same node is limited. As communicating tasks from the source or sink are selected and assigned, along with the tasks from a few of the middle operators, this leaves a number of the

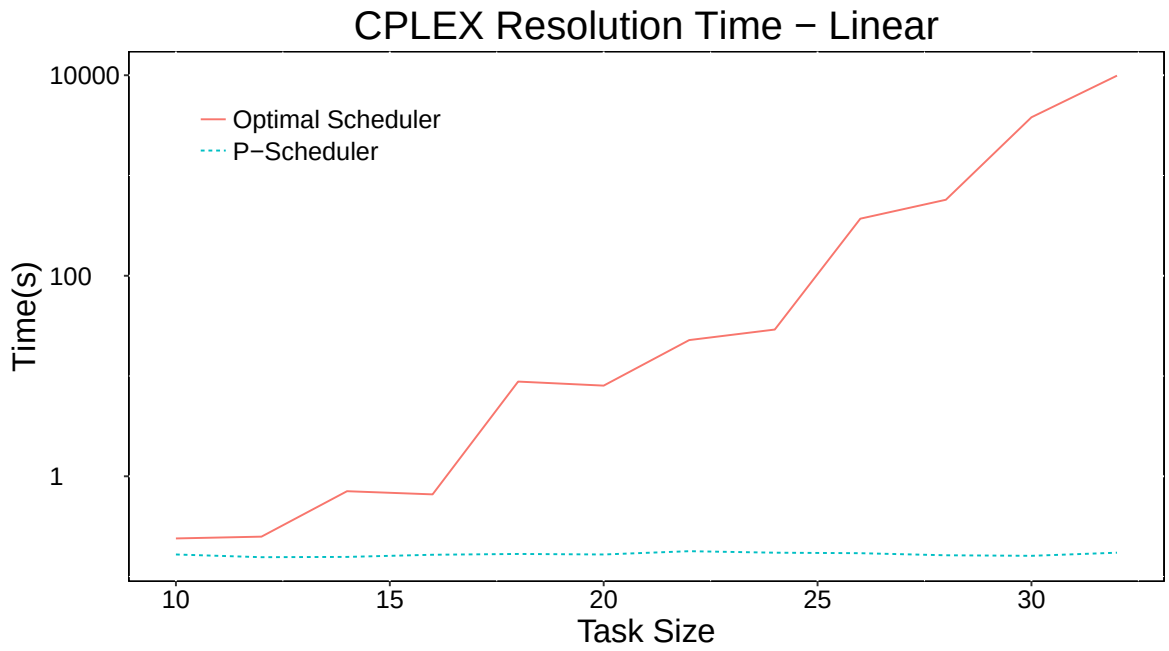


Figure 7.7: Resolution time for optimal scheduler and P-Scheduler for linear layout

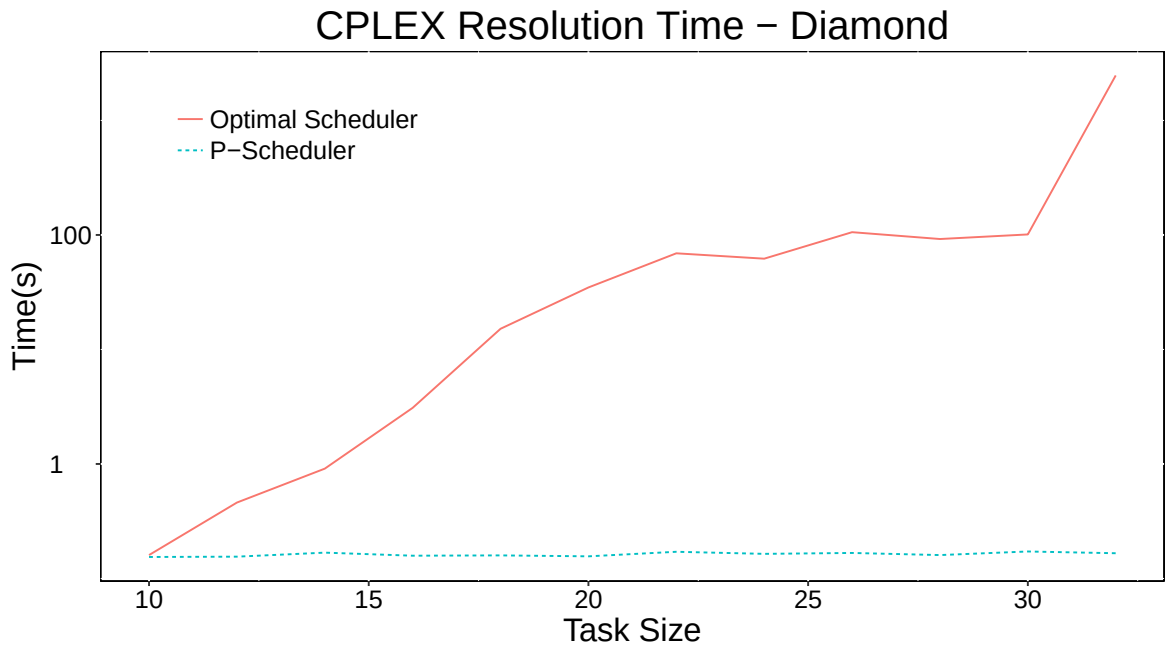


Figure 7.8: Resolution time for optimal scheduler and P-Scheduler for diamond layout

middle operators unassigned. In turn, when these operators are to be assigned, there are no neighbouring operators left for them to be grouped with, as the source and

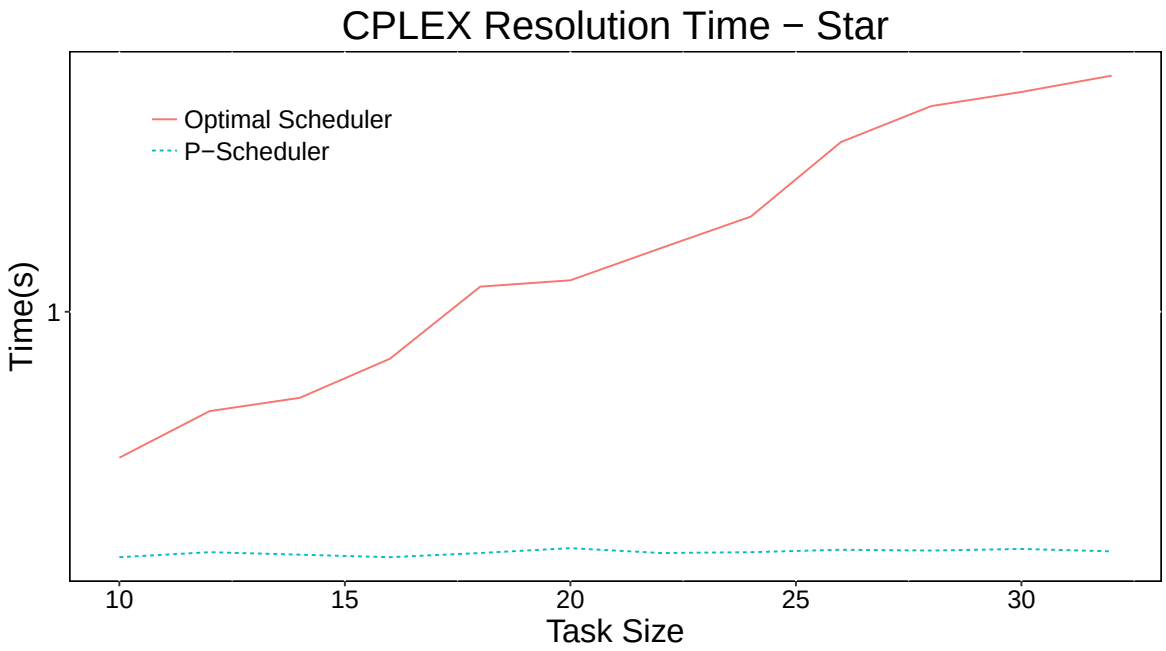


Figure 7.9: Resolution time for optimal scheduler and P-Scheduler for star layout

sink were previously assigned. The same happens in the star layout where the middle operator only has 4 tasks and can be placed with a few tasks from the sink or source operators in the same node.

While it is possible to use optimisation software to find an optimal schedule, the resolution time begins to significantly increase with the problem size. Figures 7.7, 7.8 and 7.9 show the resolution times for each micro-benchmark, where it can be seen that while the resolution time of optimal grows exponentially, P-Scheduler can always find near optimal or optimal schedules in near real-time.

Overall, P-Scheduler is able to efficiently allocate tasks to nodes within the homogeneous cluster, with near optimal results for the three micro-benchmarks. We evaluate the practical performance of P-Scheduler in the next section.

7.6 Experimental evaluation

In this section, we evaluate P-Scheduler using two real-world applications, previously described in Section 6.3.

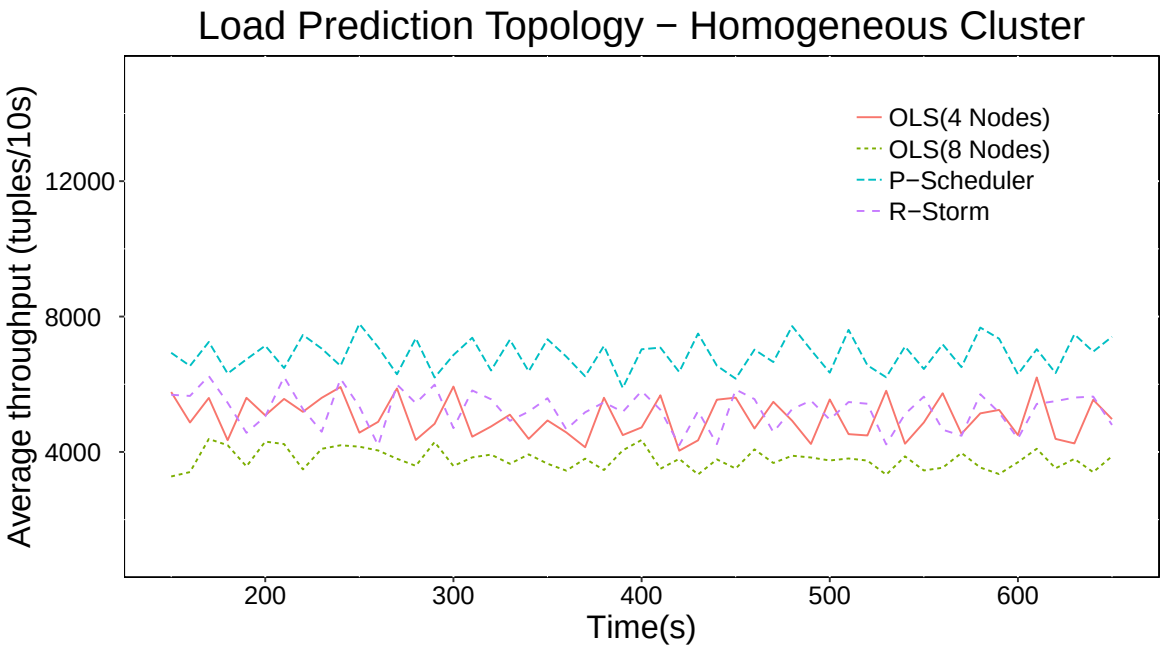


Figure 7.10: Throughput results of smart home load prediction topology, using P-Scheduler, R-Storm and OLS

7.6.1 Load prediction application for smart homes

We first run the load prediction application on a homogeneous cluster with 8 high capacity nodes using P-Scheduler, R-Storm and OLS. Figure 7.10 shows the experimental results, where it can be seen that OLS and R-Storm have an average throughput of 3,700 and 5,200 respectively. In comparison, P-Scheduler has a higher average throughput of 6,900, which represents an improvement of 32% and 86% over R-Storm and OLS respectively. The lower throughput of OLS is due to the best fit approach, which assigns task pairs to the least loaded node, spreading the tasks across all 8 nodes, while P-Scheduler and R-Storm only use 4 nodes on average. Therefore, to remove the impact of node consolidation, we restrict OLS to 4 nodes and rerun the application, shown in Figure 7.10. This increases the throughput for OLS to 5,000, but it is still unable to match either R-Storm or P-Scheduler. R-Storm is able to assign communicating tasks from most components together, but is unable to assign tasks for the Plug Load component with other tasks. It also fails to consider the data transfer rate between tasks due to its use of an offline schedule. Figures 7.11 and 7.13 show the execution latency for both topologies run by the three schedulers. From the figures, it can be seen that as the execution latency is reduced, we can achieve higher throughput.

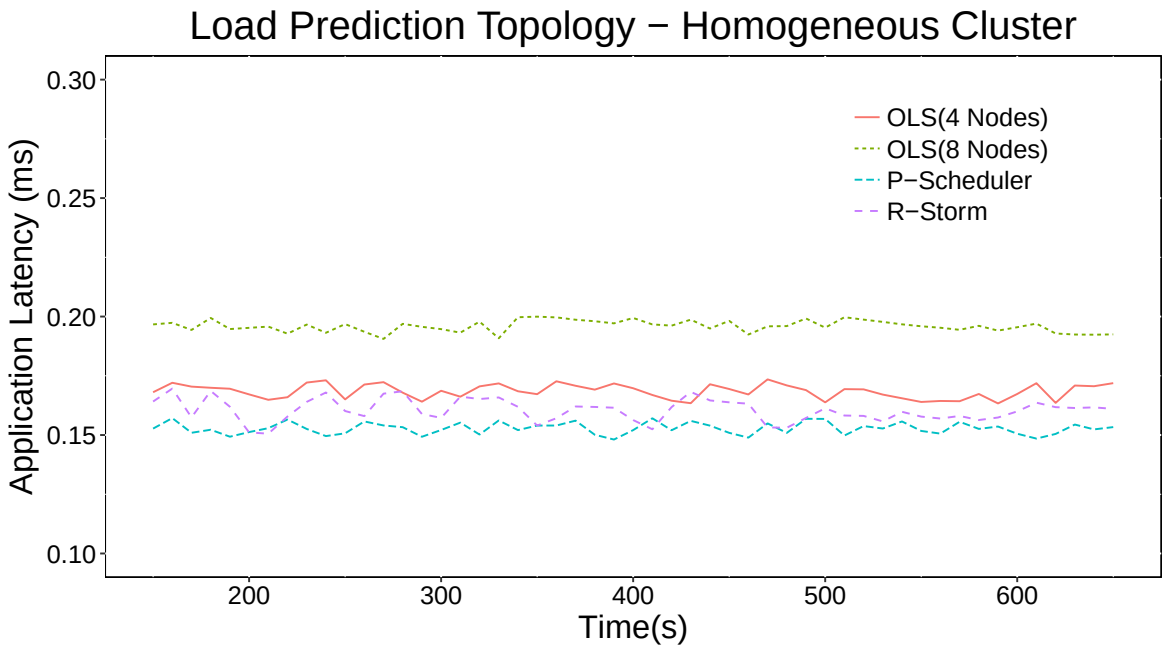


Figure 7.11: Latency results of smart home load prediction topology, using P-Scheduler, R-Storm and OLS

7.6.2 Top frequent routes in NYC taxi data

In this section, we evaluate the performance of P-Scheduler when run on the top frequent routes topology, previously described in Section 6.3. Figure 7.12 shows the throughput for this topology using P-Scheduler, R-Storm and OLS schedulers, run on a 8 node homogeneous cluster. As can be seen from the figure, the average throughput for P-Scheduler, R-Storm and OLS is 15,700, 15,500 and 9,800 respectively. Again, OLS spreads the tasks across all 8 nodes, increasing inter-node communication, while both P-Scheduler and R-Storm only use 3 nodes. When OLS is rerun with a 3-node cluster, the average throughput is 13,900, which is still lower than the other two schedulers. Finally, Figures 7.13 show the latency for each scheduler running on the homogeneous cluster configuration.

7.7 Discussion

Our experimental results have shown that P-Scheduler can more efficiently group highly communicating tasks, which are assigned to the same node. For the load prediction topology, P-Scheduler had a higher average throughput than both OLS and R-Storm. This demonstrated that P-Scheduler can better handle the different topology shapes,

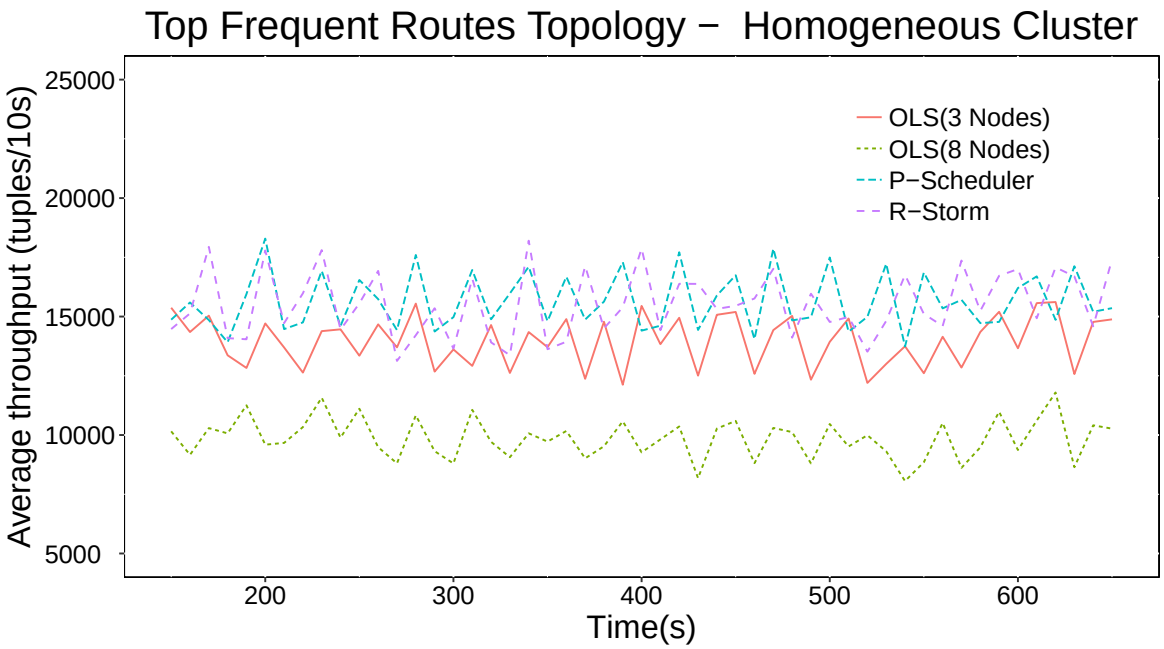


Figure 7.12: Throughput results of top frequent routes topology, using P-Scheduler, R-Storm and OLS

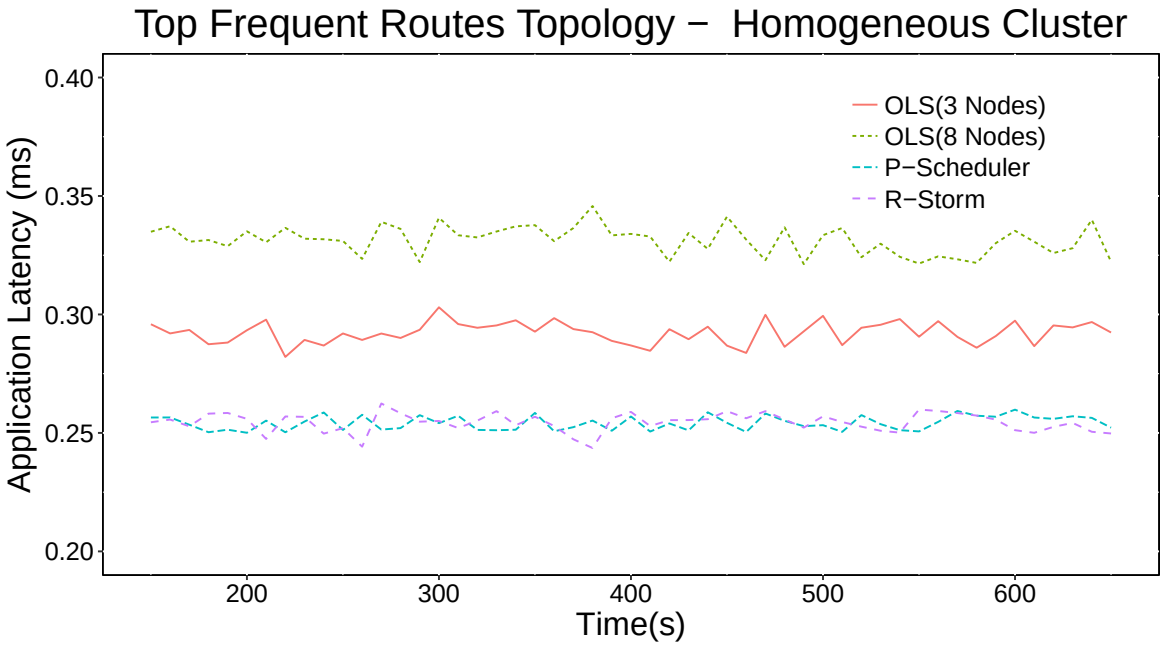


Figure 7.13: Latency results of top frequent routes topology, using P-Scheduler, R-Storm and OLS

where R-Storm was unable to reliably find the communicating tasks for the diamond shape of the Smart Home topology.

Although, P-Scheduler does not provide a significant improvement over R-Storm for the top frequent routes topology, it is worth noting that R-Storm operates offline and requires a considerable amount of tuning by the user. This tuning involves the user specifying the CPU and memory requirements. In comparison, P-Scheduler is online and is able to monitor the execution and find the CPU usage of each task, achieving the same throughput without requiring such extensive tuning.

The lower average throughput of OLS is due to a limited view of the communication for both topologies, meaning it is unable to co-locate the highly communicating tasks efficiently, because of its task pair view and greedy best fit approach.

Further, by using a static schedule, R-Storm does not have any rescheduling overhead, however, it is unable to dynamically react to changes in the communication pattern. In comparison, P-Scheduler and OLS are dynamic and incur an overhead when rescheduling, but they are able to react to the changes in the workload.

Overall, these results have shown that by monitoring the execution, P-Scheduler is able to reliably find the communication pattern between tasks. However, by employing K-way partitioning, P-Scheduler cannot be efficiently used in a heterogeneous configuration, as each partition is of a roughly equal size. To address this limitation we need to develop a partitioning-based heuristic which can find partitions of different sizes, relative to the capacity of each node. In the next two chapters, we present two heuristics which overcome this by operating on heterogeneous clusters.

7.8 Conclusion

In this chapter, we presented P-Scheduler, an adaptive hierarchical scheduling method, which uses K-way partitioning to divide up the task graph. By first partitioning the task graph into the required number of nodes, communicating tasks can be grouped together, reducing inter-node communication. The tasks assigned to each node are further partitioned into the number of workers in order to improve the inter-worker communication. We evaluated P-Scheduler using two real-world applications in a homogeneous cluster with 8 worker nodes. The experimental results show that P-Scheduler outperformed OLS, increasing throughput by 12–86% and R-Storm by up to 32%.

Chapter 8

T3-Scheduler

In this chapter, we present T3-Scheduler, for DAG-based data stream processing systems on homogeneous and heterogeneous clusters. This chapter is organised as follows. We begin by presenting an overview of T3-Scheduler in Section 8.1. A detailed description of the algorithm is given in Section 8.2, which is followed by an example in Section 8.3. Section 8.4 compares T3-Scheduler with an optimal scheduler using the three micro-benchmarks, previously described in Section 6.2. Section 8.5 evaluates T3-Scheduler with two real-world applications, previously presented in Section 6.3, with a detailed comparison discussion of T3-Scheduler with P-Scheduler in Section 8.6. Finally, Section 8.7 concludes the chapter.

8.1 Introduction

T3-Scheduler is a Topology and Traffic aware Two-level Scheduler for DAG-based DSPSs, which can find highly communicating tasks and assign them to the same compute node in homogeneous and heterogeneous clusters such that each node remains fully utilised. T3-Scheduler is *topology* aware, as it considers the shape and connectivity of the topology graph when finding highly communicating tasks. It is also *traffic* aware as it monitors the streaming application execution to find the data transfer rate between tasks communicating with each other. Finally, it is *two-level* as follows:

- First level: T3-Scheduler divides the application graph into multiple parts, where each part includes highly communicating tasks, with a size relative to the capacity of a compute node in the heterogeneous cluster. This allows T3-Scheduler to reduce inter-node communication by placing groups of highly communicating tasks together and to fully utilise each compute node, by filling it to capacity.

The novel technique used by T3-Scheduler to find groups of neighbouring tasks in the application graph provides a broad view of the communicating tasks, rather than the localised view of some previous approaches.

- Second level: Once tasks are assigned to a compute node, T3-Scheduler finds the best assignment of those tasks within the node by placing highly communicating tasks in the same worker process. This helps to minimise the communication between the worker processes within a compute node.

Our contributions in this chapter are summarised as follows:

- We present T3-Scheduler, a scheduling algorithm that searches the application graph to find groups of highly communicating tasks, which are grouped and assigned to the same compute node, based upon the available capacity of computing nodes within the cluster. This approach of searching the application graph to find neighbouring groups of communicating tasks and assigning them to the same node reduces the inter-node communication, while fully utilising each node.

- The communication cost of T3-Scheduler is evaluated by comparing it to a theoretically optimal scheduler, implemented in CPLEX, when run on three micro-benchmarks, each representing a different communication pattern, previously described in Section 6.2. The evaluation shows that T3-Scheduler can achieve results that are close to optimal in a number of different cluster configurations.

- T3-Scheduler is implemented in Apache Storm 1.1.1 and through experimental results it is shown that T3-Scheduler outperforms OLS (Aniello *et al.*, 2013) and R-Storm (Peng *et al.*, 2015) (previously described in Chapter 3) when run on three micro-benchmarks and two real-world applications. The results show that T3-Scheduler has an average throughput up to 26% higher than R-Storm and 12–78% higher than OLS for the two real-world applications.

8.2 T3-Scheduler algorithm

To efficiently assign tasks to compute nodes in a DAG-based DSPS, a scheduler needs to find groups of highly communicating tasks, which can then be assigned to the same node, reducing the inter-node communication cost. This can be seen as a graph partitioning problem where the application graph is partitioned into multiple parts such that the number of edge cuts between the partitions is minimised. Each part can then

be assigned to a compute node, such that the node's capacity is not exceeded. Usually, graph partitioning algorithms, such as K-way partitioning proposed by Karypis and Kumar (1998), are used to produce partitions of roughly equal parts, suitable for a homogeneous configuration as proposed in (Fischer and Bernstein, 2015; Eskandari *et al.*, 2016). However, to produce partitions of different sizes, graph partitioning algorithms are dependent upon *a priori* information. That is, they rely on knowing the number of tasks to be assigned to each node before the graph can be partitioned. This is a barrier to practical deployments as it is difficult to reliably know this information at scheduling time.

To address this limitation, T3-Scheduler uses a heuristic method to produce partitions of different sizes such that the inter-part communication is reduced. Each part can then be assigned to a node with a relative capacity. This is achieved without requiring any *a priori* information, such as the number of tasks to be assigned to each compute node.

T3-Scheduler consists of five main steps as follows.

1. **Monitoring:** To facilitate the task scheduling, T3-Scheduler needs to initially profile the task graph execution, measuring the tasks' load and data transfer rate between the communicating tasks.

2. **Constructing a simplified graph:** The profile from the previous step is then used to build a weighted simplified graph. This graph is initially similar to the operator-view graph, and is updated with new vertices and edges if necessary.

3. **Node selection:** T3-Scheduler selects a node by taking the capacity into account.

4. **First level of scheduling:** At this level, T3-Scheduler determines which groups of tasks should be co-located within each node by finding a sub-graph of highly communicating tasks from the simplified graph.

5. **Second level of scheduling:** At this level, T3-Scheduler finds the number of workers required for each node and then divides the sub-graph found by first-level of scheduling into the number of workers such that highly communicating tasks are grouped together into a worker in order to reduce inter-worker communication.

Algorithm 2 presents these steps and the following subsections provide a detailed discussion of each step. Table 8.1 shows the notation used in this chapter for the T3-

Algorithm 2 Pseudo-code for T3-Scheduler algorithm

```
1: op = MONITORING(Stream_Application)
2: sg = CONSTRUCT_SIMPLIFIED_GRAPH(op)
3: SCHEDULE(nodes, sg)
4:
5: function SCHEDULE(nodes, sg)
6:     while all g in sg are not assigned do
7:         n = NODE_SELECTION(nodes)
8:         sbg = FIRST_LEVEL(n, sg)
9:         SECOND_LEVEL(n, sbg, T)
10:    end while
11: end function
12: function FIRST_LEVEL(n, sg)
13:    sbg ← ∅
14:    while n is not full do
15:        g = GROUP_SELECTION(n, sg)
16:        Add g to sbg
17:        Mark g in sg as assigned
18:    end while
19:    return sbg
20: end function
21: function SECOND_LEVEL(n, sbg, T)
22:    t ← FIND_TASK_SIZE(sbg)
23:    w = ⌈t/T⌉
24:    parts ← Partition sbg into w parts using METIS
25:    Assign each part in parts to one worker on n
26: end function
```

Scheduler algorithm. Figure 8.1 shows the interaction between T3-Scheduler, METIS, monitoring log and worker nodes in a Storm cluster. In the following, each step is discussed in detail.

8.2.1 Monitoring

Similar to the monitoring step in P-Scheduler, T3-Scheduler monitors the execution of the streaming application. This involves measuring the load of each task and the

Table 8.1: Notation for T3-Scheduler algorithm

Symbol	Meaning
op	Online profile
sg	Simplified graph
g	A group of tasks, represented as a vertex in sg
sbg	A sub-graph, consisting of highly communicating tasks
t	Number of tasks within sbg
T	Max number of tasks per worker
w	Required number of workers
n	Current node

communication rate between each task pair, which is recorded in the monitoring log. This profile is then able to be read periodically by T3-Scheduler during rescheduling.

8.2.2 Constructing a simplified graph

T3-Scheduler constructs a weighted simplified graph, initially similar to the operator-view graph, using the online profile, collected from the monitoring step. T3-Scheduler initially aggregates all of the tasks within each operator into a single group, representing a vertex in the graph. The weight of the new group/vertex in the simplified graph is found by summing the load of each task within the group. Each edge in the graph, connecting two groups, is the aggregation of all the communications between two groups, weighted with the sum of the data transfer rate of their communicating tasks. This is made possible by the fact that the tasks within two communicating operators are fully connected in typical DSPSs because of stream grouping.

The simplified graph has the advantage of having a view of the connectivity between all the tasks. Therefore, a sub-graph, consisting of highly communicating tasks, can be found and assigned to the same compute node. Additionally, by considering communicating groups of tasks instead of communicating tasks, a slight change between some data transfer rates of communicating tasks within the communicating groups will not result in rescheduling, making the scheduling more stable. Vertices and edges are updated regularly if any vertex has to be partitioned in order to fit in a compute node.

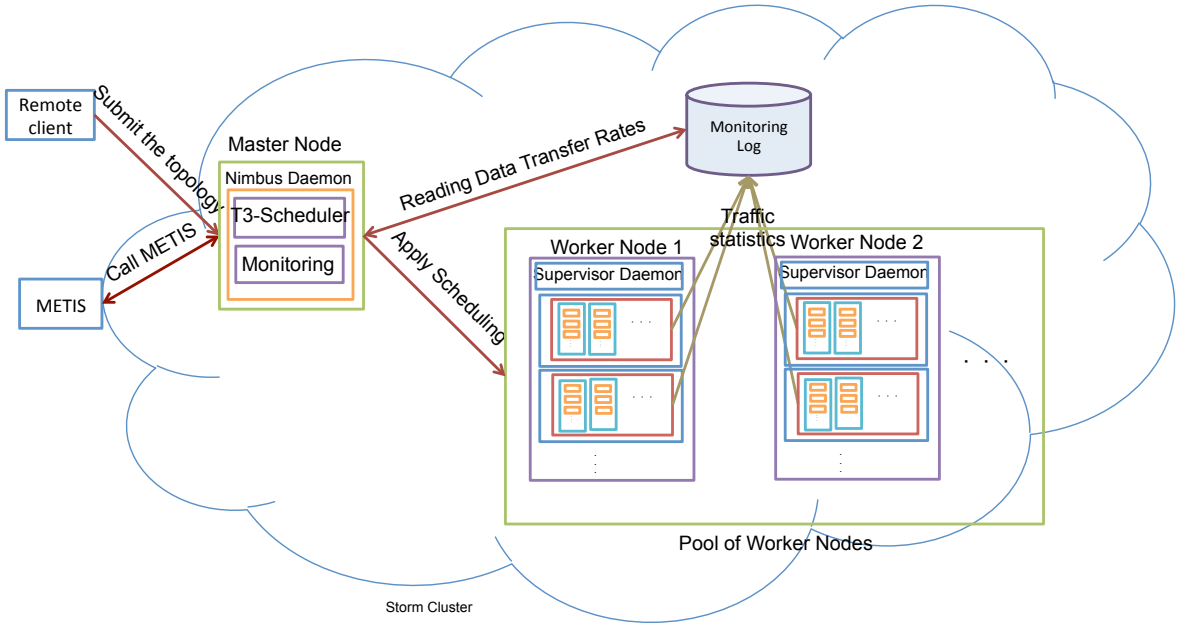


Figure 8.1: The interaction between T3-Scheduler, METIS, monitoring log and worker nodes in a Storm cluster

8.2.3 Node selection

T3-Scheduler considers the capacity and resource availability of each node, selecting the highest capacity node. This allows T3-Scheduler to take steps towards minimising the inter-node communication as a result of placing more communicating tasks in the higher capacity nodes. If multiple nodes have the same capacity, ties are broken by selecting a node randomly among the potential nodes. T3-Scheduler fills a node with as many communicating tasks as possible, up to its capacity, and then moves to the next highest capacity node.

8.2.4 First level of scheduling

The goal of the first level of scheduling in T3-Scheduler is to divide the simplified graph into multiple parts where each part consists of highly communicating groups of tasks. T3-Scheduler takes a heuristic approach to divide the graph into multiple parts which are sized according to the capacity of the heterogeneous node to be scheduled on. For every empty compute node, T3-Scheduler begins by finding the highest communicating group pair as a starting point in the simplified graph and expanding it by repeatedly selecting the most highly communicating neighbouring groups, forming a sub-graph, until the node is full. Once the node becomes full, a new compute node is selected for

Algorithm 3 Pseudo-code for Group Selection Algorithm

```
1: function GROUP_SELECTION(n, sg)
2:     if n is empty then
3:         p = FIND_STARTING_POINT(sg)
4:         if p can fit in n then
5:             return p
6:         else
7:             (p1, p2) = GROUP_PAIR_PARTITIONING(p)
8:             UPDATE_SIMPLIFIED_GRAPH(sg)
9:             return p1
10:        end if
11:    else
12:        g = FIND_NEIGHBOUR(sg)
13:        if g can fit in n then
14:            return g
15:        else
16:            (g1, g2) = GROUP_PARTITIONING(g)
17:            UPDATE_SIMPLIFIED_GRAPH(sg)
18:            return g1
19:        end if
20:    end if
21: end function
```

which a new starting point is found and the same procedure is applied. By following this approach, T3-Scheduler can find highly communicating tasks and place them in the same compute node. In the following, we provide further details of the first level of scheduling.

Forming a sub-graph

After locating the group pair with the highest weight as the starting point, we evaluate if the pair of groups is able to fit within the current compute node. This condition is checked by comparing the sum of the tasks' load of each group, and the capacity of the selected compute node. This will have two possible outcomes. If the compute node has sufficient capacity to accommodate the group pair, it is assigned to the compute node. However, in the event of the compute node having insufficient capacity, a fine-grained

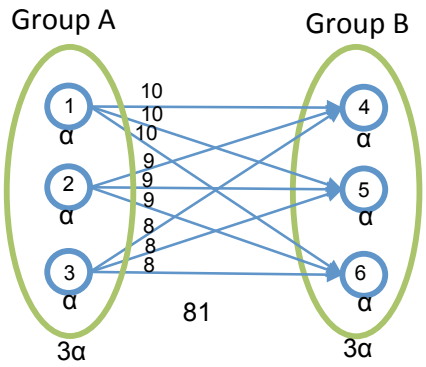


Figure 8.2: Group pair (A, B), with load 6α, to be assigned to a node, with capacity of 4α

partitioning will be performed on the group pair, where the number of tasks within one or both groups is reduced. This will enable a new, smaller group pair to be assigned to the current compute node which would not otherwise be possible. Partitioning a group pair will be explained in more detail shortly.

If the node still has some remaining capacity, we expand the starting point by finding the most highly communicating neighbours. Having located the immediate neighbour with the highest weight, an evaluation is performed to check if this group can fit within the compute node. If the node has sufficient capacity for the neighbouring group, it is added to the sub-graph and the next highest weighted neighbour will be evaluated. But, in the event that the highest weight neighbour is not able to fit within the compute node, an additional fine-grained partitioning will be performed on this group. Partitioning a single group will be explained in more detail shortly. After performing the single group partitioning, we expand the sub-graph to include the new group, which now has a size equivalent to the node's capacity, meaning it is fully utilised. Algorithm 3 presents the pseudo-code for the process of group selection in forming a sub-graph step.

Fine-grained group pair partitioning

To resolve the issue of insufficient capacity to accommodate the group pair selected as the starting point, we partition this group pair into two smaller group pairs, thereby allowing a subset of the initial group pair to be assigned to the compute node. For instance, assume that the group pair denoted as (A, B), shown in Figure 8.2, has a total size of 6α, which is unable to fit within the compute node with a total capacity of 4α,

where α is the average load for each task. Therefore, we have to partition the group pair, (A, B), into two smaller group pairs (A_1, B_1) and (A_2, B_2) such that (A_1, B_1) can fit. The aim of partitioning (A, B) is to minimise the edge cuts between (A_1, B_1) and (A_2, B_2) while maximising the number of tasks in (A_1, B_1). To achieve this, the task pairs with the highest rate from (A, B), are repeatedly selected and assigned to (A_1, B_1) until the node's capacity is reached.

However, when selecting a task pair (t_i, t_j) with the highest data transfer rate from (A, B), three scenarios are possible. To keep track of which tasks have been selected from (A, B), selected tasks are marked as 'selected' in (A, B) when they are assigned to (A_1, B_1).

1. Both tasks, t_i and t_j, are new and have not been selected before. In this case, both tasks will be assigned to (A_1, B_1) if the sum of the load of t_i and t_j is less than or equal to the compute node's remaining capacity.

2. One of the tasks is not new and is already marked as selected in A or B. To simplify the explanation, we assume that t_i in task pair (t_i, t_j) is already marked as 'selected' in A and is not new, but t_j is new. We first find all the task pairs connected to t_j from (A, B), denoted as (t_k, t_j), where t_k in A is not marked as 'selected'. Then, among these task pairs, we pick the task pair with the highest rate that can fit in the compute node and assign it to (A_1, B_1). By doing this, (t_i, t_j) is also included in (A_1, B_1) due to full connectivity of two communicating groups. This has the benefit of assigning two new tasks to (A_1, B_1) at each step with the highest data transfer rate and therefore minimising the edge cuts between (A_1, B_1) and (A_2, B_2).

3. Both tasks have already been assigned to (A_1, B_1) and are marked as 'selected' in (A, B). In this case, no further processing is done and we move to the task pair which has the next highest data transfer rate.

If one group is smaller than the other, for example A is smaller than B, we cannot always find a task pair with two new tasks. In this case, we just assign the task pair with the highest data transfer rate, that can fit in the compute node, to (A, B_1). After repeatedly selecting the task pairs and reaching the compute node's capacity, all the unmarked tasks in B are assigned to B_2, which will be inspected for assignment to another compute node later by T3-Scheduler. The simplified graph is updated with the new vertices, edges and their weights every time a partitioning is performed. A detailed example of partitioning a pair is presented in the next section.

An example of group pair partitioning

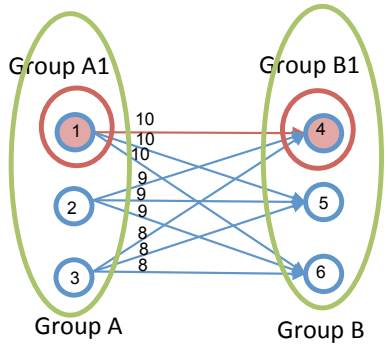
This section presents the step by step process of partitioning the group pair (A, B) in the example presented in Figure 8.2. We assume that the group pair (A, B) in the figure, which has an aggregated data transfer rate of 81 (sum of data transfer rate for task pairs) and 3α load for each group (α being the load for each task), has been selected to be assigned to a node with an available capacity of 4α. These numbers have been selected to simplify the explanation of the example. Since the sum of tasks' load for group A and group B is greater than the node's capacity, the group pair (A, B) needs to be partitioned. First, the task pair with the highest data transfer rate should be selected. In this example, there are three task pairs $(1, 4)$, $(1, 5)$ and $(1, 6)$ with a data transfer rate of 10, which is the highest. None of the tasks in these task pairs have previously been selected (scenario 1), therefore we select task pair $(1, 4)$ at random and assign it to (A_1, B_1), as shown in Figure 8.3a.

The next highest data transfer rate for the remaining task pairs belong to $(1, 5)$ and $(1, 6)$. However, one task of both task pairs, *task* 1, has been selected before and is already in (A_1, B_1) (scenario 2). Having the same condition for both task pairs, task pair $(1, 5)$ is selected at random for further processing. As we want to assign two new tasks to (A_1, B_1) which have not been marked as selected, we find all the task pairs in (A, B) that include *task* 5 and a task not marked as selected. There are two possible task pairs $(2, 5)$ and $(3, 5)$, which meet this criteria, shown in Figure 8.3b. Among all the potential task pairs, we pick the task pair with the highest rate, which can fit in the compute node. In this example, task pair $(2, 5)$ is selected, as shown in Figure 8.3c. After each selection the load and capacity is checked. The node is full at this point, so the rest of the tasks which are not marked as selected, form (A_2, B_2) group pair, shown in Figure 8.3d, which will be assigned to another node by following the T3-Scheduler algorithm.

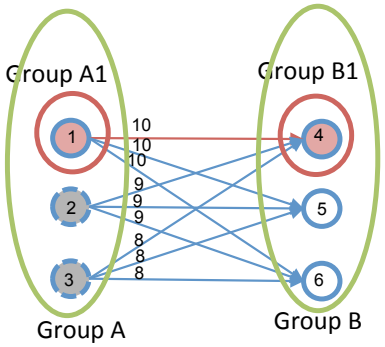
Fine-grained single group partitioning

When expanding the sub-graph, the situation can arise where an entire group, denoted as A, is unable to fit within the current compute node's remaining capacity, requiring A to be partitioned. The aim is to utilise the node by filling it with the most highly communicating tasks from A.

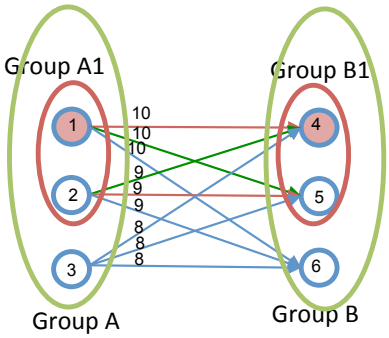
At each step, the task with the highest data transfer rate from A, which is connected to the sub-graph within the compute node, is found and assigned to a smaller group,



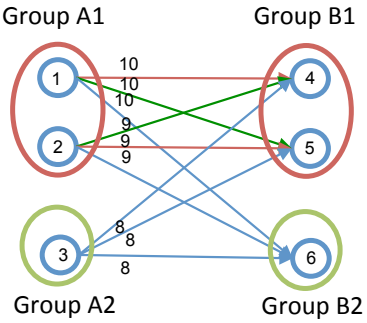
(a) Scenario 1: Selecting task pair $(1, 4)$



(b) Scenario 2: Potential task pairs are $(2, 5)$ and $(3, 5)$



(c) Selecting task pair $(2, 5)$



(d) Forming group pairs (A_1, B_1) and (A_2, B_2)

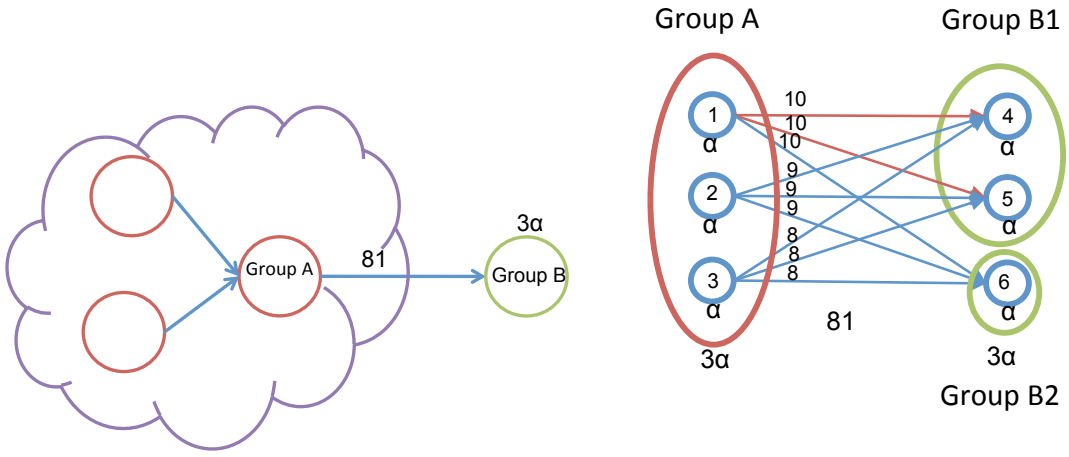
Figure 8.3: An example of group pair partitioning by T3-Scheduler

denoted as A_1, until the compute node is full. The selected task from A is marked as 'selected' in A after assignment to A_1. In the case that the selected task's load is higher than the available capacity, the task with the next highest data transfer rate is inspected for assignment. This process is repeated until we find a task that can fit in the node. Otherwise, the group partitioning process is complete.

The remaining tasks from A, which are not marked as selected, form a new group, denoted as A_2. Then, the simplified graph is updated with the new vertex, edges and their weights. The new group, A_2, has this chance to be placed in another compute node with the unassigned task groups in the simplified graph that it communicates with. A detailed example of partitioning a pair is presented in the next section.

An example of single group partitioning

This section presents the steps of applying the single group partitioning algorithm to Group B, from the example shown in Figure 8.2. We assume that the sub-graph



(a) Remaining capacity is 2α (b) Selecting tasks 4 and 5

Figure 8.4: An example of single group partitioning by T3-Scheduler

shown in red in Figure 8.4a has been selected for assignment to the current node, which already includes Group A. Since the node still has some remaining capacity, Group B is selected as the neighbour with the highest weight for assignment, expanding the sub-graph. We assume that (A, B) has an aggregated data transfer rate of 81. The remaining capacity of the node is 2α, but the size of the selected group is 3α. These numbers have been selected to simplify the explanation of the example. Therefore, Group B is too large for the remaining capacity and requires single group partitioning such that one group can fit in the remaining capacity.

To do this, we pick the tasks with the highest transfer rate which are connected to the sub-graph. In our example, the task pairs $(1, 4)$, $(1, 5)$ and $(1, 6)$ have the highest rate, so we pick $(1, 4)$ at random. The remaining capacity is now α, so we pick $(1, 5)$ next. The node is now full and by picking the task pairs with the highest transfer rates we have reduced the edge cuts between the sub-graphs which are to be assigned to the different nodes.

8.2.5 Second level of scheduling

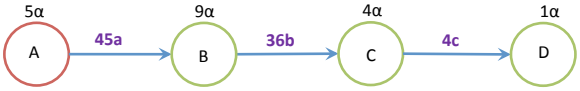
After placing the highly communicating tasks on the same node, T3-Scheduler determines which task should be assigned to the same worker process. Similar to the second level in P-Scheduler, this level of scheduling is used in data stream processing systems that have a number of worker processes per node. The number of tasks in the sub-graph assigned to each node is used to calculate the required number of workers,

w, based on Equation 7.3 (see Section 7.3.4). Unlike P-Scheduler, which calculates w once for all of the nodes in a homogeneous cluster, the second level scheduler in T3-Scheduler calculates w for each node as the node capacity varies in a heterogeneous cluster, where higher capacity nodes can have more workers than lower capacity nodes. K-way partitioning is applied to the sub-graph assigned to each node, dividing it into w equal sized parts, which are then assigned to a worker. By using the second level of scheduling to place highly communicating tasks within the same worker process, we are able to minimise the inter-worker communication.

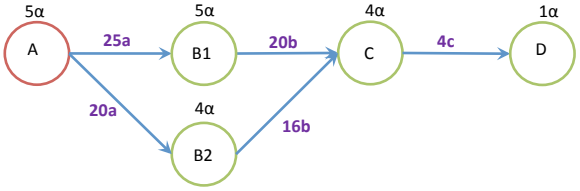
8.3 An example of task assignment by T3-Scheduler

In this section, we apply the T3-Scheduler algorithm to the top trending topics use-case presented in Chapter 4. In this example, we have a heterogeneous cluster, consisting of five compute nodes of varying sizes. It is assumed that the task load for all the tasks in the application are the same and equal to α. This is a simplifying assumption we make in this current example, where each task in the application is the same size, however, T3-Scheduler is not dependent upon this assumption and is capable of scheduling tasks of varying sizes, as previously discussed. Based on this assumption, the capacity of each node is defined as the number of tasks which are able to be allocated to each node multiplied by α. *Node* 1, *node* 2, *node* 3, *node* 4 and *node* 5 have capacities of 6α, 4α, 10α, 6α and 5α respectively. It is assumed that *node* 3 is configured to have up to four workers, whereas the rest of the nodes can have up to two workers.

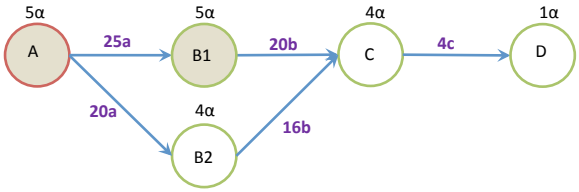
The first step performed by the T3-Scheduler is to monitor the execution and collect the tasks' load and data transfer rate between the tasks, previously described in Section 8.2.1. We further assume that for the top trending topics application in this example, the communication rates between communicating operators obtained from monitoring step reduce at each subsequent step. This is a valid assumption as less data progresses through each operator. In this example, we assume weights a, b and c are obtained for each task pair in (A, B), (B, C) and (C, D) group pairs respectively, assuming that $a > b > c$. Therefore, while the exact communication rates are not known, these weights allow us to present the way our algorithm works. To further simplify this current example, we assume that the data transfer rate between the task pairs of each group pair is the same. This assumption is based upon the lack of skew in the topics. However, our algorithm can handle different data transfer rates, as previously discussed. The assumptions made in this current example are strictly for illustrative



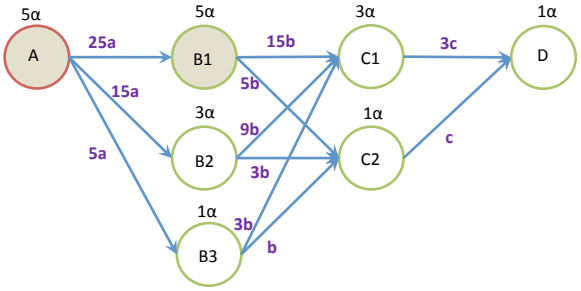
(a) Simplified Graph built by T3-Scheduler for top trending topics application



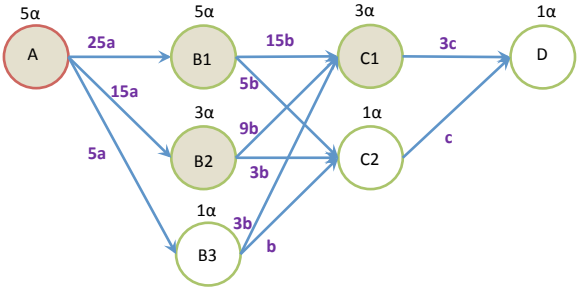
(b) Updated simplified graph after first group pair partitioning



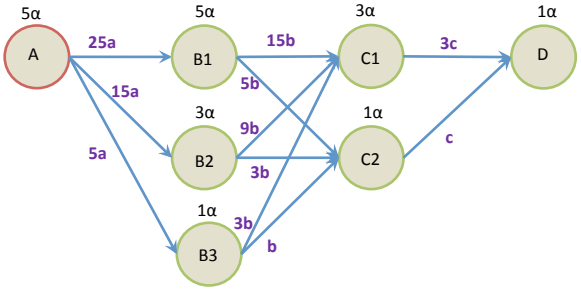
(c) Assigning (A, B_1) to *node* 3



(d) Updated simplified graph after second group pair partitioning



(e) Assigning (B_2, C_1) to *node* 1



(f) Assigning (B_3, C_2) and (C_2, D) to *node* 4

Figure 8.5: An example of task assignment by T3-Scheduler

purposes.

As the second step, a simplified graph is constructed based on these statistics such that each vertex weight is calculated by the summation of the task loads in each operator. Also, the weight for an edge connecting two groups is calculated by the aggregation of the data transfer rates of all the task pairs. The simplified graph can be seen in Figure 8.5a.

Node selection in T3-Scheduler begins by selecting the largest capacity node from the heterogeneous cluster. In this case, the selected node is *node* 3, with a capacity of 10α. Having selected a node, in the first level of scheduling, we then need to locate a starting point within the simplified graph. Group pair (A, B), consisting of the Emit Topics and Rolling Count groups, has the highest weight of $45a$, as shown in Figure 8.5a.

After evaluating this group pair, it can be seen that the group pair cannot fit in the selected node because the whole group pair has 14α, while *node* 3 has a capacity of only 10α, requiring a group pair partitioning. The results of performing fine-grained group pair partitioning, as explained in Section 8.2.4, are shown in Figure 8.5b. This then allows the smaller group pair, (A, B_1), to be assigned to *node* 3. Further, B_2 can be assigned to another node along with its communicating neighbours. Following this, the graph weights and edges are updated, as they are continually updated at each step after each partitioning. *Node* 3 is full at this point with (A, B_1), shown in Figure 8.5c, so we move to the next node.

The next node selected by the node selection algorithm is *node* 1, which has a capacity of 6α. The starting point for this node is (B_2, C) with the highest edge weight of $16b$ and including 8α, however this group pair is unable to fit within *node* 1 with a capacity of 6α. Therefore, the group pair is partitioned and the results are shown in Figure 8.5d. After group pair partitioning, (B_2, C_1), with an edge weight of $9b$, is assigned to *node* 1. The graph weights are then updated. The immediate neighbours for this pair includes B_3, C_2 and D. However, *node* 1 is now full, Figure 8.5e, and there is no further remaining capacity on this node.

The next available node to be selected is *node* 4 with a capacity of 6α. The starting point of (B_3, C_2) with the weight of b is found for *node* 4. As this node still has some remaining capacity, we find and process the immediate neighbours to find the group with the highest weight which is connected to the groups already assigned to this node. The next group to be picked is D, which is the only immediate neighbouring group. After assigning D to the node, the first-level scheduling finishes as all the groups are assigned, as shown in Figure 8.5f.

For the second level of scheduling, T3-Scheduler partitions the sub-graph assigned to each node into the number of required workers. We assume that T is set to 6 for this example. The number of workers required for *node* 3, *node* 1 and *node* 4 is 2, 1 and 1 respectively. Therefore, there is a need to further partition the sub-graph assigned to *node* 3. After performing the partitioning by METIS, each part is assigned to one worker and T3-Scheduler finishes here.

8.4 T3-Scheduler vs. optimal scheduler

In this section, we compare the communication cost and resolution time of T3-Scheduler with a theoretically optimal scheduler using three micro-benchmarks that represent

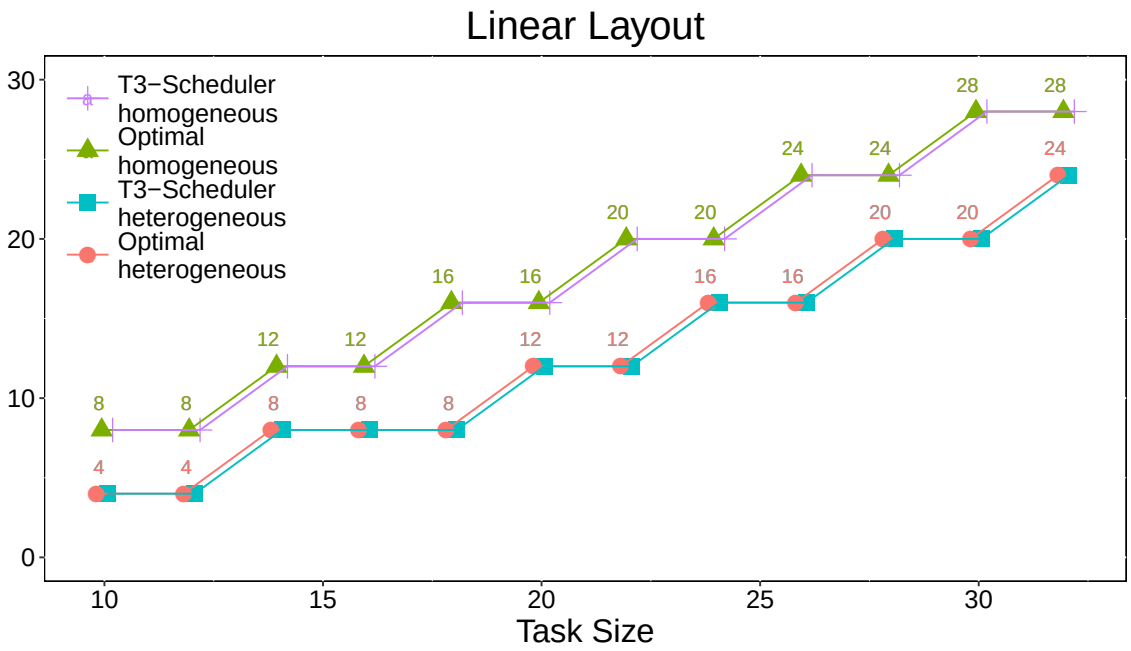


Figure 8.6: T3-Scheduler vs. optimal scheduler for linear layout

common shapes of streaming applications. These micro-benchmarks are described in Section 6.2, along with detail about the shape of each micro-benchmark, the number of tasks per operator for each problem size and the cluster settings.

The communication costs found by the optimal scheduler and T3-Scheduler for the micro-benchmarks are shown in Figures 8.6, 8.7 and 8.8. It is worth noting that while the optimal scheduler can be used to solve small problem sizes in a feasible amount of time, this does not remain the case as the problem size increases due to the increased computational complexity. For the problem sizes greater than 20, the solutions were found in minutes or hours which is not practical. Here, our intended purpose for determining the optimal communication cost is to evaluate how close to optimal the near real-time results of T3-Scheduler are.

From Figure 8.6, it can be seen that for both the homogeneous and heterogeneous configurations, T3-Scheduler is able to match the results of the optimal scheduler for the linear layout. There is a notable gap in the communication cost between the homogeneous and heterogeneous configurations. This is a result of the node capacities, where the heterogeneous cluster configuration has some larger nodes, 3 with capacity 6, which allows more of the tasks to be placed in fewer nodes, thereby reducing the communication cost when compared to the homogeneous configuration which uses smaller nodes with a capacity of 4.

Figures 8.7 and 8.8 show the results for diamond and star layouts. As can be seen

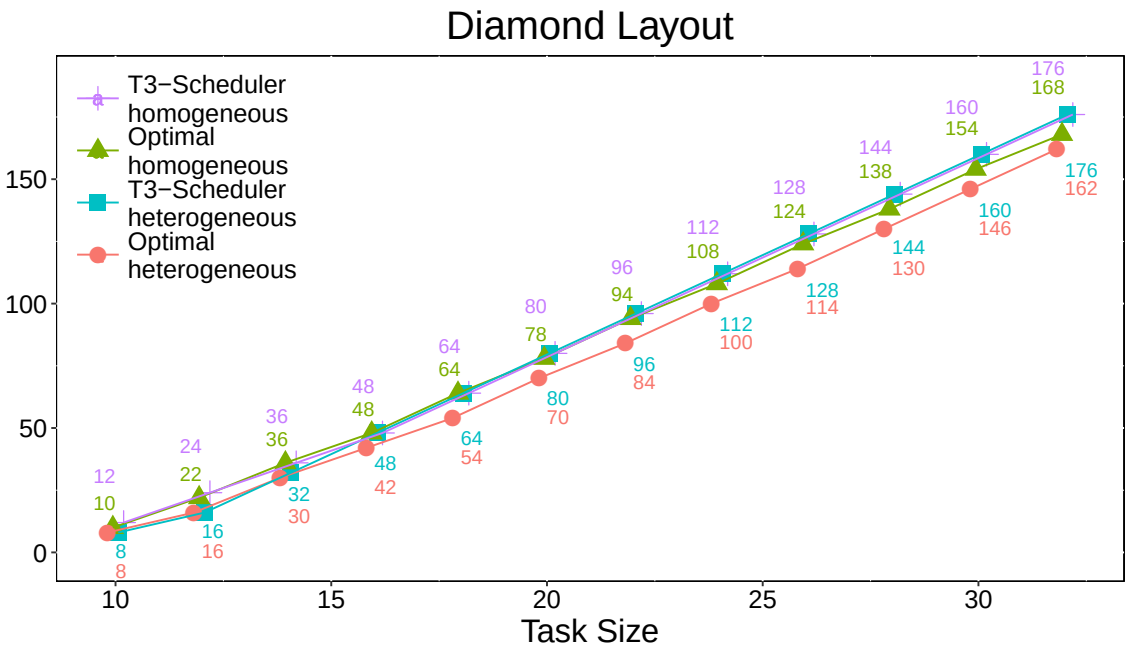


Figure 8.7: T3-Scheduler vs. optimal scheduler for diamond layout

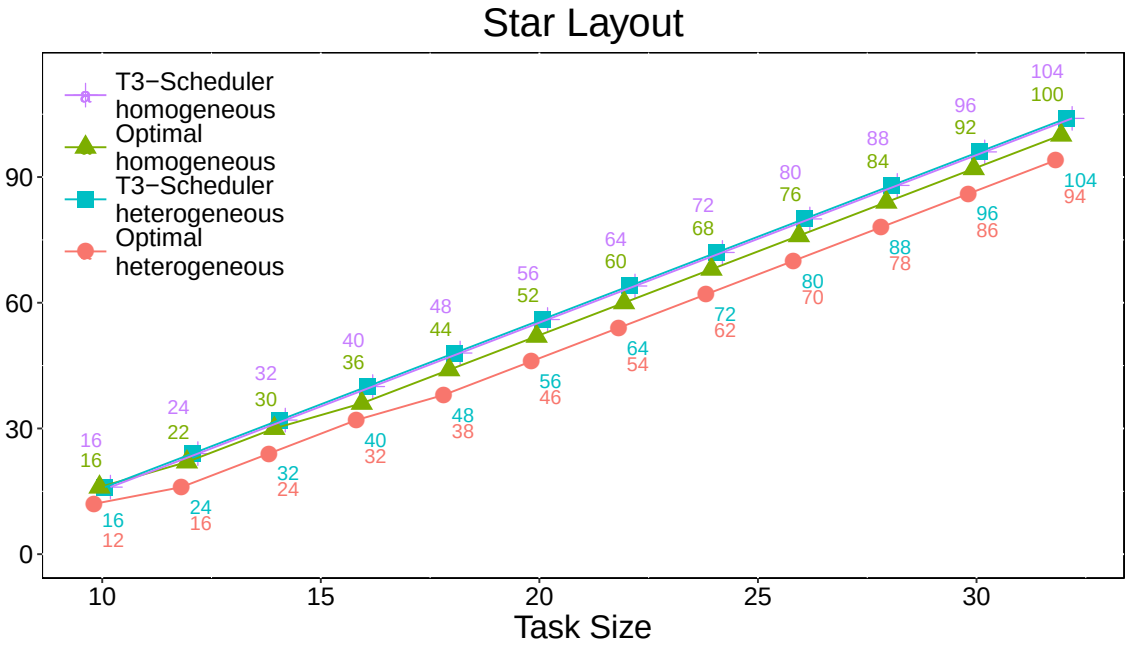


Figure 8.8: T3-Scheduler vs. optimal scheduler for star layout

from the figures, there is a difference between the communication costs found by T3-Scheduler and optimal scheduler for some problem sizes. This can be attributed to

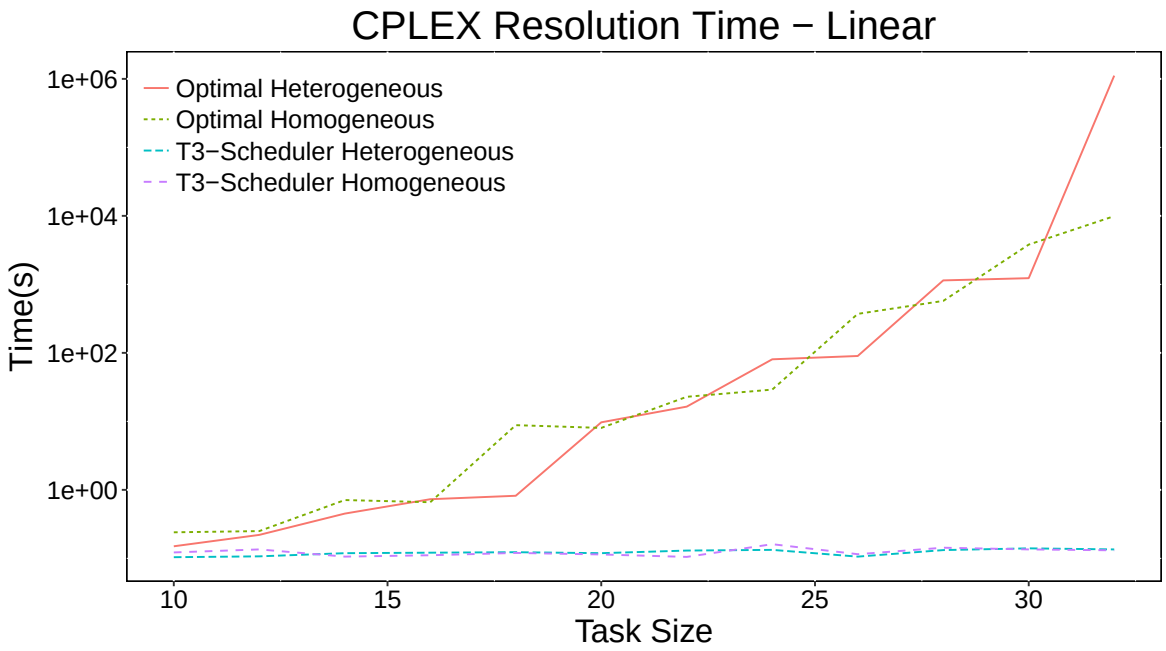


Figure 8.9: Resolution time for optimal scheduler and T3-Scheduler for linear layout

the different approaches, where T3-Scheduler tries to place all of the tasks within an operator along with its neighbouring operators which results in a sub-optimal solution. In comparison, the optimal scheduler selects tasks from multiple operators to achieve the optimal result.

It can also be seen from Figures 8.6, 8.7 and 8.8 that the communication costs of both schedulers for diamond and star layouts are larger than for the linear layout. The reason is that in the linear layout, more communicating tasks can be grouped and placed in the same node because of the shape of the layout when compared to the other two layouts. In the diamond layout, the source and sink operators only have 4 tasks each, which means that the number of communicating tasks that are able to be placed in the same node is limited. As communicating tasks from the source or sink are selected and assigned, along with the tasks from a few of the middle operators, this leaves a number of the middle operators unassigned. In turn, when these operators are to be assigned, there are no neighbouring operators left for them to be grouped with, as the source and sink were previously assigned. The same happens in the star layout where the middle operator only has 4 tasks and can be placed with a few tasks from the sink or source operators in the same node.

While this evaluation has shown that it is possible to use optimisation software to determine an optimal schedule for some small problem sizes, it is worth noting that it quickly becomes impractical as the problem size grows. To illustrate this, Fig-

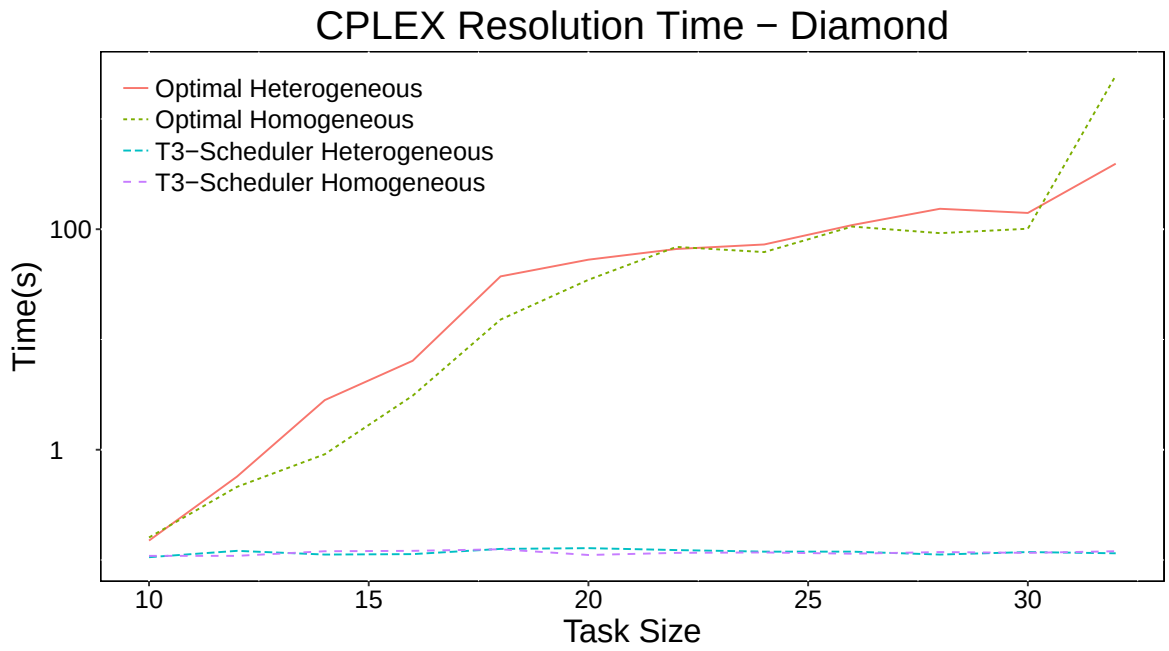


Figure 8.10: Resolution time for optimal scheduler and T3-Scheduler for diamond layout

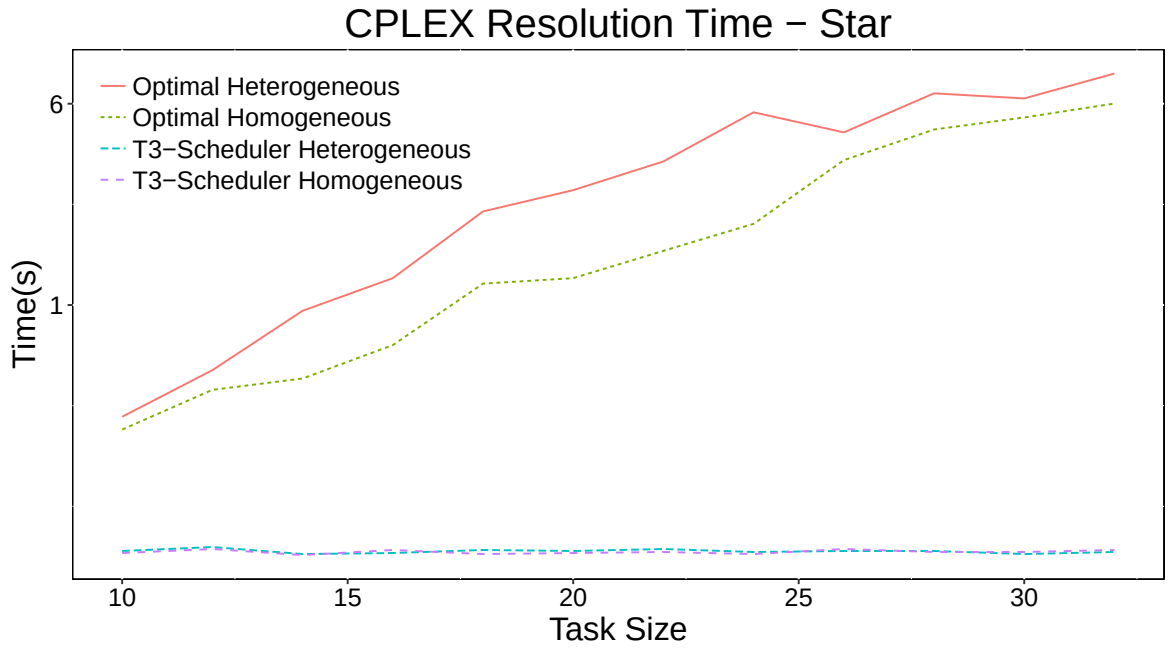


Figure 8.11: Resolution time for optimal scheduler and T3-Scheduler for star layout

ures 8.9, 8.10 and 8.11 shows the resolution times for the micro-benchmark evaluations. From the figures, it can be seen that the resolution time begins to increase as the task size goes above about 20. For example, for the linear micro-benchmark on the heterogeneous cluster, shown in Figure 8.9, the resolution time for a task size of 32 is 1,114,257 seconds, and that 50GiB of RAM is used. This required the optimisation software to be run on a 64-core server with 512GiB of RAM, meaning that it would take substantially longer on a more modestly powered system. While the resolution times for both diamond and star are not quite as high as for linear, they still quickly reach times that are impractical for scheduling, even for these small problem sizes. For 32 tasks, diamond and star have resolution times of 2,471 and 6 seconds for the homogeneous cluster and 393 and 7 for the heterogeneous cluster, respectively.

Overall, the T3-Scheduler is able to efficiently allocate tasks to the nodes within the cluster with near optimal results for the diamond and star layouts, and equalling the optimal scheduler for the linear layout. This is an important result for practical application deployments, which commonly implement the linear layout. The heuristic approach adopted by T3-Scheduler is able to quickly find a near optimal solution, making it a good solution for scheduling in practice. We evaluate the practical performance of our implementation in the next section.

8.5 Experimental evaluation

In this section, we evaluate T3-Scheduler using three micro-benchmarks and two real-world applications, described in Section 6.2 and Section 6.3, respectively.

8.5.1 Micro-benchmarks

I/O-intensive: We run each of the I/O-intensive micro-benchmarks on a heterogeneous cluster consisting of one high capacity node and two low capacity nodes. The small cluster size ensures each scheduler assigns tasks to both high and low capacity nodes, allowing us to evaluate how well highly communicating tasks are co-located together. This configuration also removes the impact of node consolidation, performed by T3-Scheduler and R-Storm. The results for the micro-benchmark execution for T3-Scheduler, R-Storm and OLS are presented in Figure 8.12a. As can be seen in the figure, T3-Scheduler is able to outperform OLS by 7–41% for all of the micro-benchmarks, while achieving a similar average throughput to R-Storm for the linear and star micro-benchmarks. For the dia-

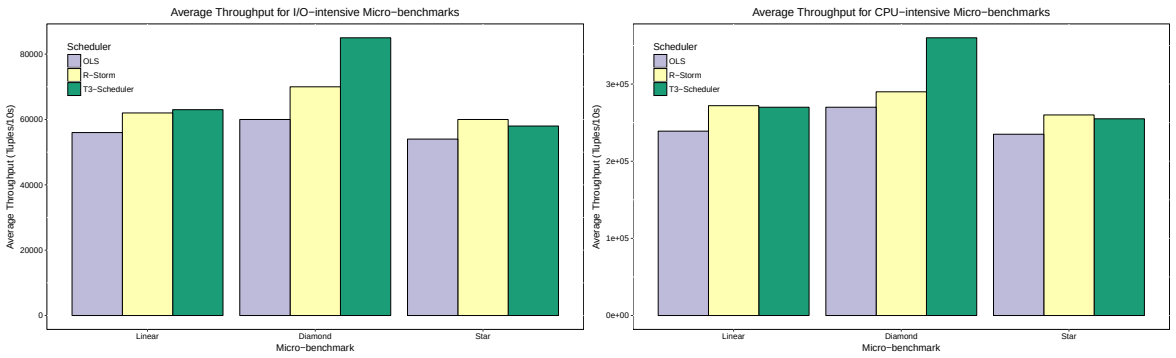
mond micro-benchmark, T3-Scheduler outperforms R-Storm by 20%, which is the result of a more efficient task placement of all of the bolts. R-Storm was unable to assign any of the tasks in one of the middle bolts to the same node as either the source or sink, increasing inter-node communication. It is also worth noting that these results for R-Storm were only achieved after tuning user-configured parameters, whereas T3-Scheduler does not have such a requirement.

CPU-intensive: Each micro-benchmark is run on a heterogeneous cluster consisting of two high capacity nodes and four low capacity nodes. The results for the CPU-intensive micro-benchmarks are shown in Figure 8.12b. As can be seen from the figure, T3-Scheduler achieves similar results as the previous I/O-intensive configuration, outperforming OLS, with similar average throughput to R-Storm for linear and star, and higher for diamond. The throughput for each of the micro-benchmarks is much higher than was previously seen for the I/O-intensive configuration, placing a greater load on the CPUs, which is a result of a higher rate for the spouts. By placing more communicating tasks closer together, T3-Scheduler has an average throughput 24–33% higher than R-Storm and OLS for the diamond micro-benchmark, showing the benefit of our group based assignment approach. T3-Scheduler can also outperform OLS by 8–12% for linear and star micro-benchmarks.

Overall, these results demonstrate the ability of T3-Scheduler to efficiently place more communicating tasks closer together by monitoring the execution, improving overall throughput. In comparison, R-Storm is an offline scheduler which is unable to respond to dynamic changes in the data transfer rate between tasks. While this is not a problem for the micro-benchmarks, due to their consistent communication pattern, this is a limitation for real-world applications. Further, R-Storm needs the user to configure the application requirements before execution which might not be practical or convenient. The best fit approach of OLS, assigns each task pair to the least loaded node, which is likely to spread the communicating tasks over the compute nodes.

8.5.2 Load prediction application for smart homes

We first evaluate T3-Scheduler by running the load prediction topology, described in Section 6.3.1, on a homogeneous cluster with 8 high capacity nodes using T3-Scheduler, R-Storm and OLS. Figure 8.13 shows the experimental results.



(a) I/O-intensive micro-benchmarks (b) CPU-intensive micro-benchmarks

Figure 8.12: Throughput results of I/O-intensive and CPU-intensive linear, diamond and star micro-benchmarks, using T3-Scheduler, R-Storm and OLS

As can be seen from the figure, OLS and R-Storm have an average throughput of 3,700 and 5,200 respectively, while T3-Scheduler has an average throughput of 6,600. This represents an improvement of 26% and 78% over R-Storm and OLS respectively. The lower throughput of OLS is because of the best fit approach that is used, which results in tasks being spread across all the nodes as it assigns each task pair to the least loaded node. While R-Storm co-locates tasks from multiple components reasonably well, the Plug Load component is the exception, where its tasks are assigned separately from all other tasks, resulting in a lower throughput. It also fails to consider the data transfer rate between tasks due to offline scheduling.

On average, R-Storm and T3-Scheduler use 4 out of 8 nodes while OLS uses all 8 nodes because of its best fit approach. Therefore, for a fairer comparison we rerun OLS on 4 nodes, removing the impact of node consolidation. As can be seen in Figure 8.13, the throughput of OLS increases when run on fewer nodes, however it is still lower than the other two schedulers.

We also run the topology on a heterogeneous cluster, consisting of 2 high capacity nodes and 4 low capacity nodes, using all three schedulers. This is intended to demonstrate which of the schedulers is better able to place the highly communicating tasks within the limited high capacity nodes, reducing inter-node communication. The results are shown in Figure 8.14. As can be seen, T3-Scheduler can outperform R-Storm and OLS because of its task selection. It is worth noting that the throughput is lower for all schedulers than in the homogeneous setting, as there is more inter-node communication.

Figures 8.15 and 8.16 show the execution latency for the topology run by the three

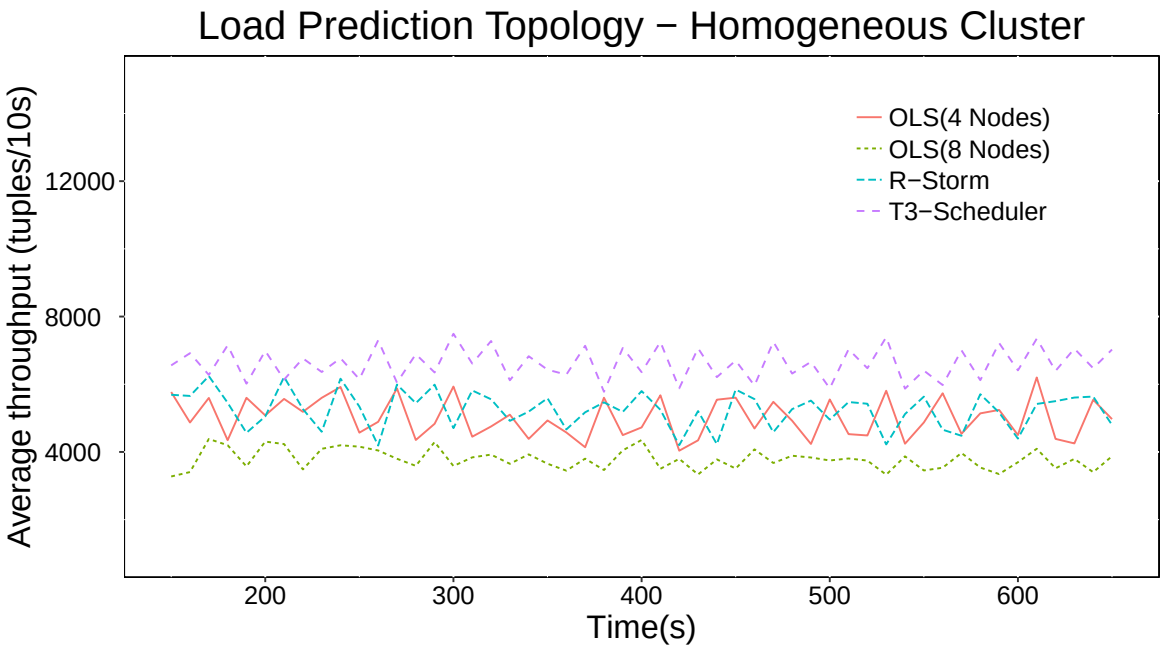


Figure 8.13: Throughput results of smart home load prediction topology in a homogeneous cluster, using T3-Scheduler, R-Storm and OLS

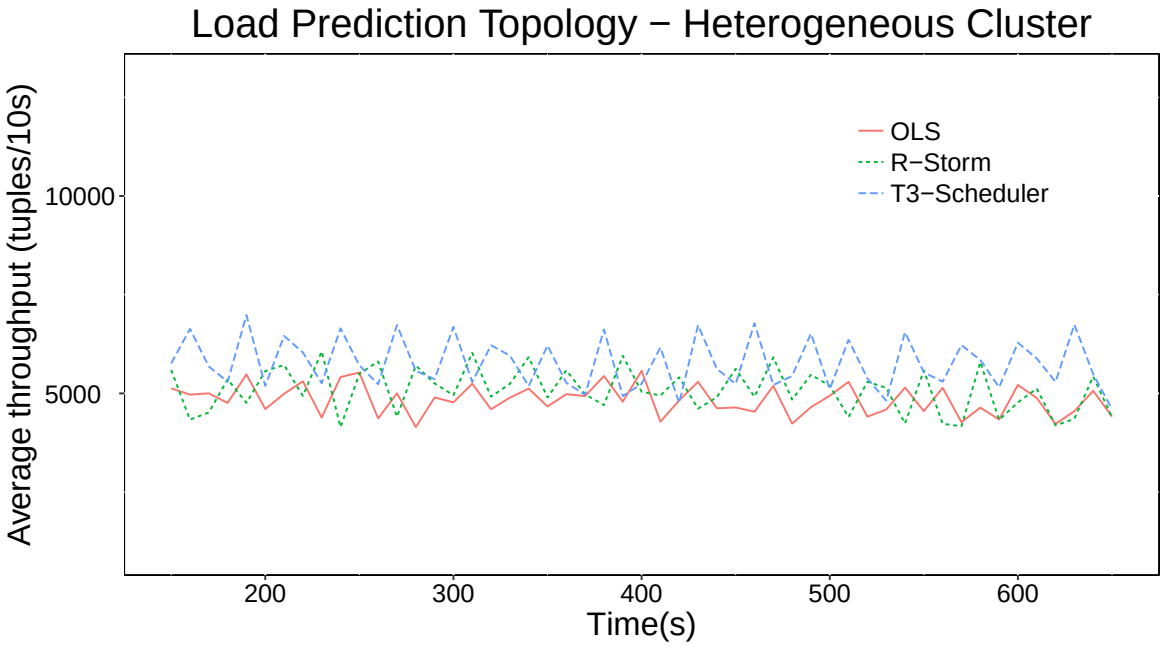


Figure 8.14: Throughput results of smart home load prediction topology in a heterogeneous cluster, using T3-Scheduler, R-Storm and OLS

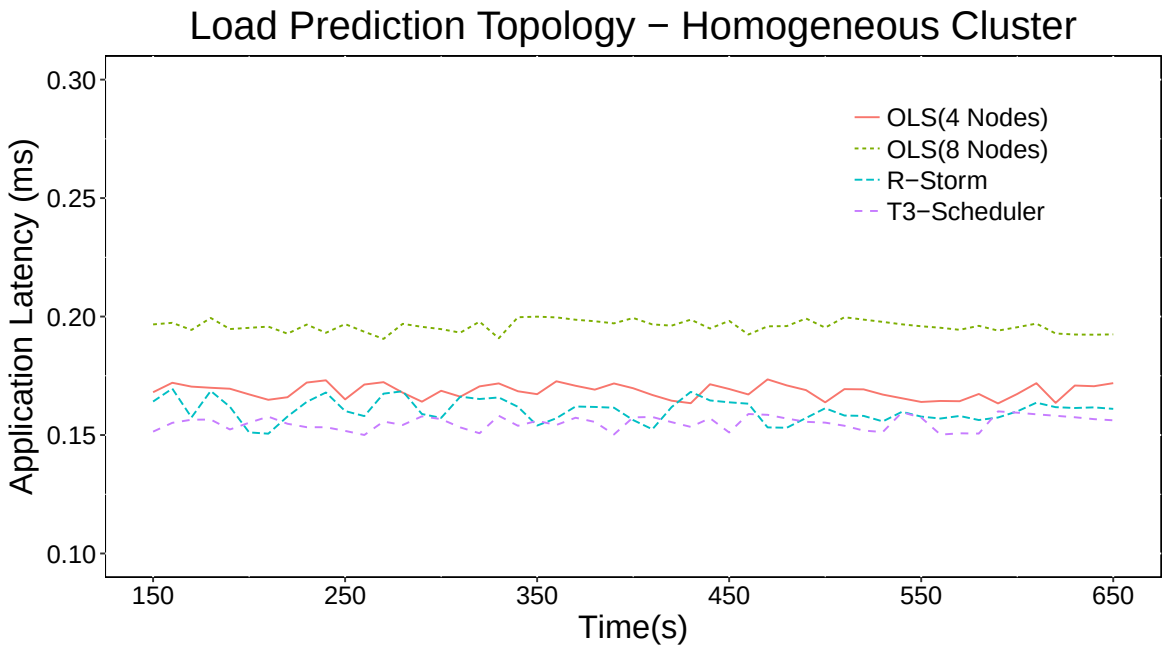


Figure 8.15: Latency results of smart home load prediction topology in a homogeneous cluster, using T3-Scheduler, R-Storm and OLS

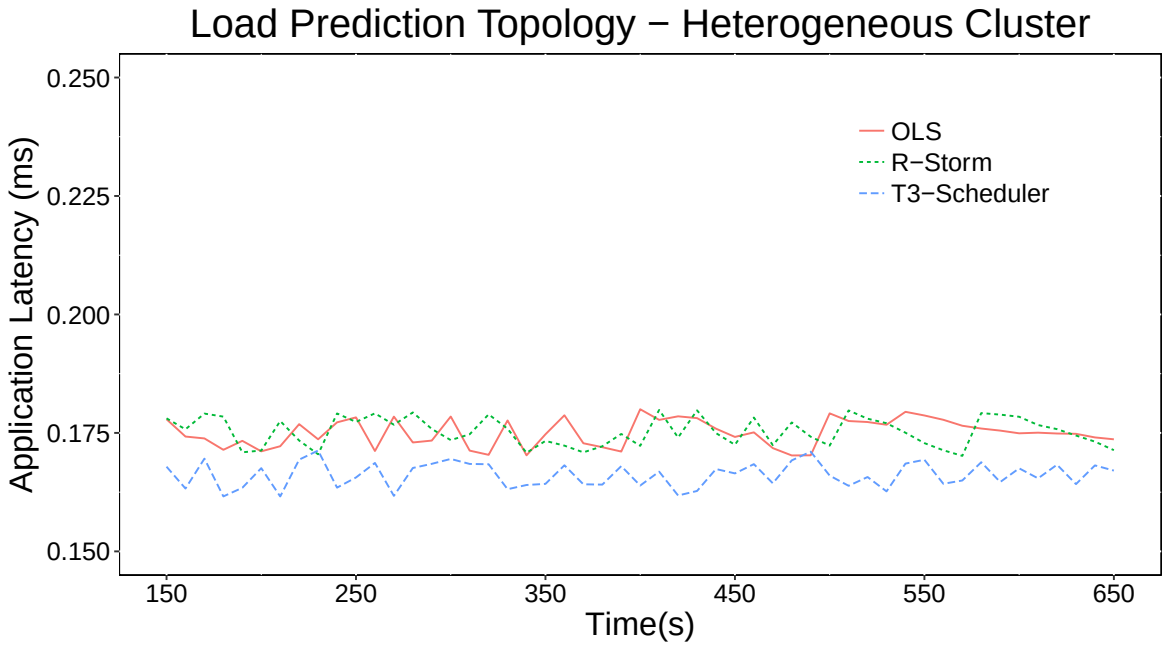


Figure 8.16: Latency results of smart home load prediction topology in a heterogeneous cluster, using T3-Scheduler, R-Storm and OLS

schedulers. From the figures, it can be seen that as the execution latency is reduced, we can achieve higher throughput. Overall, the experimental results show that T3-Scheduler can achieve a better performance as a result of a more efficient task placement.

8.5.3 Top frequent routes in NYC taxi data

We now compare the throughput of the three schedulers running the top frequent routes topology, previously described in Section 6.3.2, on an 8 node homogeneous cluster. As can be seen in Figure 8.17, the average throughput for T3-Scheduler, R-Storm and OLS is 15,700, 15,500 and 9,800 respectively. Again, OLS suffers from the problem of spreading the tasks across all 8 nodes, while both T3-Scheduler and R-Storm only use 3 nodes. When running the application with OLS on a 3-node cluster, the average throughput is 13,900, which is still lower than the other two schedulers.

We further evaluate each of the schedulers on a heterogeneous cluster with 2 high capacity nodes and 2 low capacity nodes. From the results, shown in Figure 8.18, it can be seen that T3-Scheduler has an average throughput of 14,900 while R-Storm and OLS have an average of 14,800 and 13,000 respectively. Finally, Figures 8.19 and 8.20 show the latency for each scheduler running on the homogeneous and heterogeneous cluster configurations.

Although, T3-Scheduler does not provide a significant improvement over R-Storm for top frequent routes topology, it is worth noting that R-Storm requires a considerable amount of tuning by the user. In comparison, T3-Scheduler is able to achieve the same results without requiring such extensive tuning. Both R-Storm and T3-Scheduler outperform OLS as it takes a greedy best fit approach which fails to see the whole communication pattern and therefore is not able to co-locate the highly communicating tasks efficiently. As seen in the load prediction topology, which has a diamond shape, R-Storm is unable to reliably find the communicating tasks because of its task selection method. While R-Storm operates offline and does not have any rescheduling overhead, it is unable to dynamically react to changes in the communication pattern. In comparison, T3-Scheduler incurs an overhead for rescheduling, but is better able to find highly communicating tasks by monitoring task execution and can react accordingly.

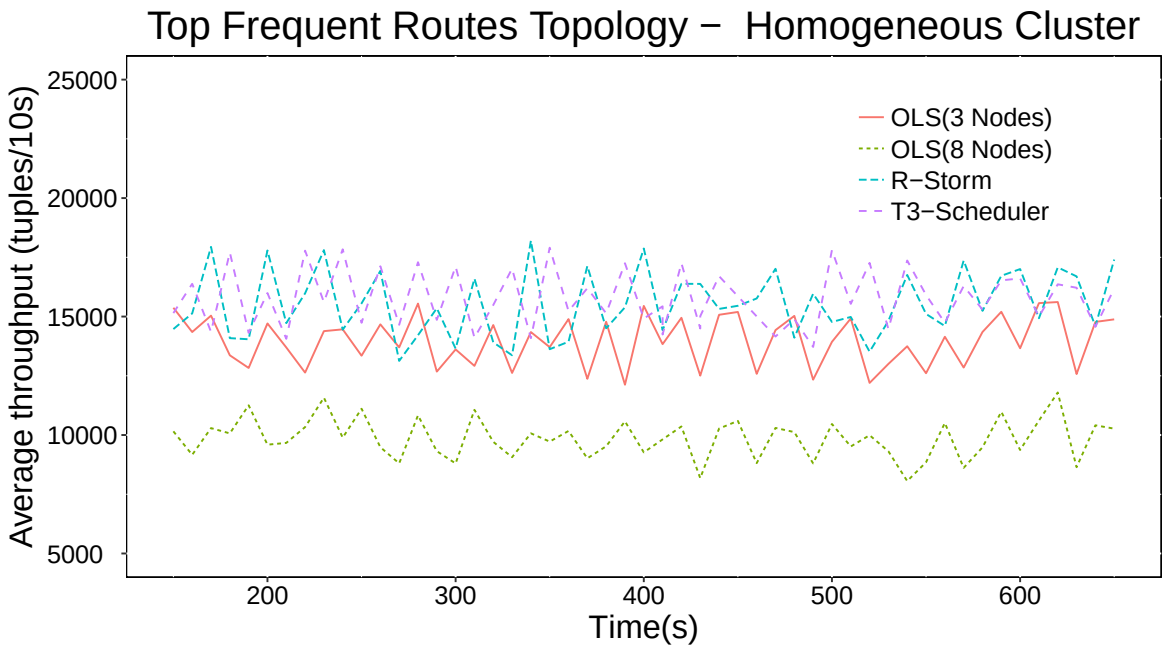


Figure 8.17: Throughput results of top frequent routes topology in a homogeneous cluster, using T3-Scheduler, R-Storm and OLS

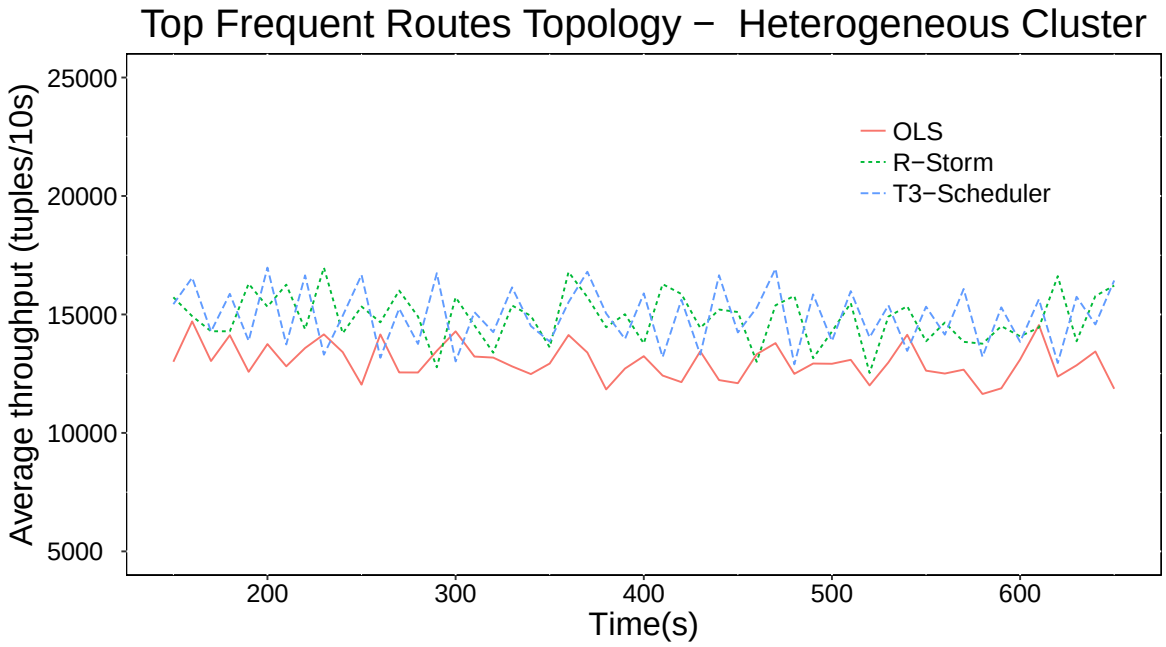


Figure 8.18: Throughput results of top frequent routes topology in a heterogeneous cluster, using T3-Scheduler, R-Storm and OLS

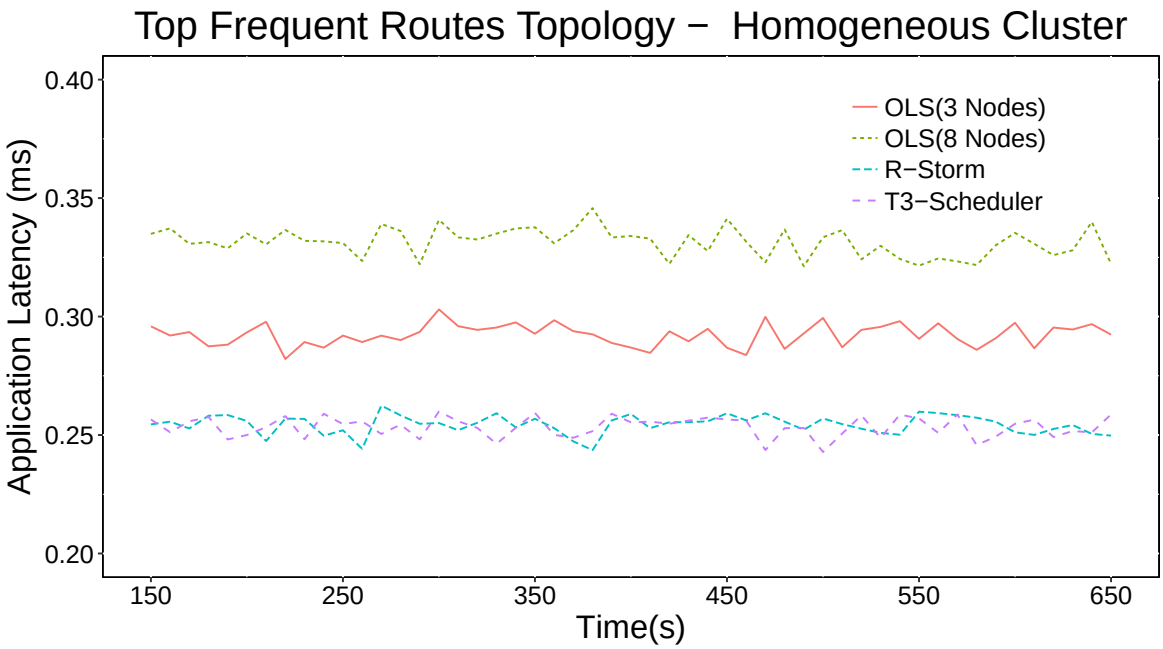


Figure 8.19: Latency results of top frequent routes topology in a homogeneous cluster, using T3-Scheduler, R-Storm and OLS

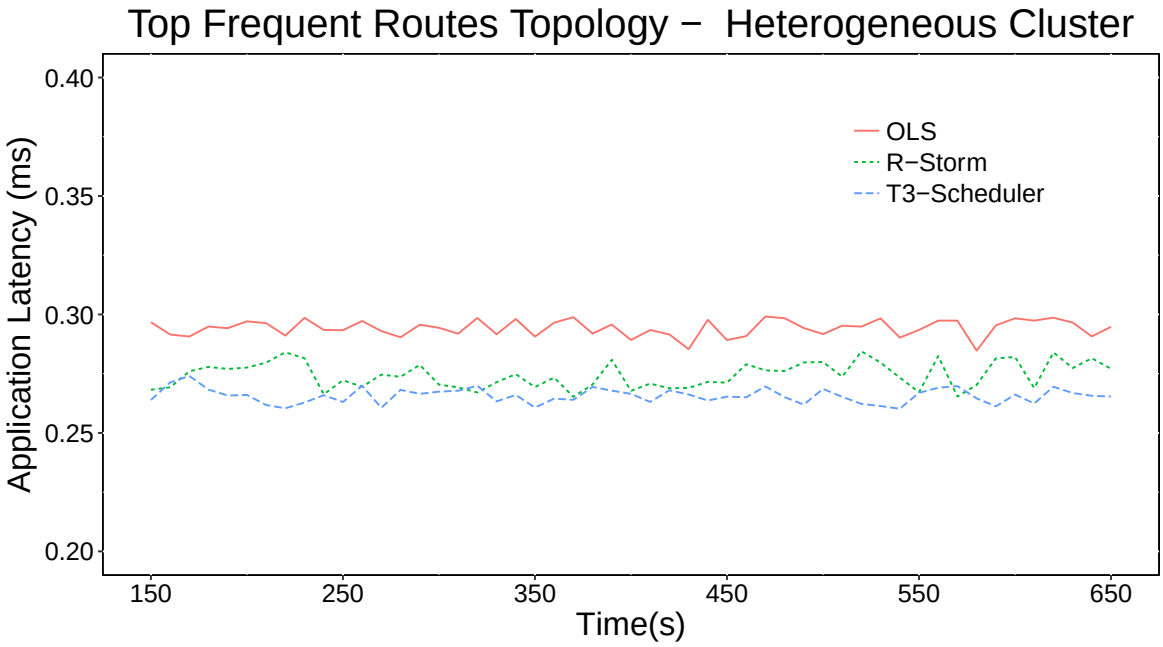


Figure 8.20: Latency results of top frequent routes topology in a heterogeneous cluster, using T3-Scheduler, R-Storm and OLS

8.6 Discussion

Our results for the homogeneous and heterogeneous cluster configurations show that T3-Scheduler can efficiently find the highly communicating tasks and co-locate them on the same node. In comparison, R-Storm has a lower average throughput for the load prediction topology as a result of its task selection method which is unable to co-locate all of the communicating tasks for the diamond shape topology. Also, OLS has lower throughput for both the top frequent routes and load prediction topologies due to its best fit greedy approach. These results have a similar trend to those previously seen for P-Scheduler, discussed in Section 7.7.

While T3-Scheduler can find different sized partitions of highly communicating tasks, these partitions are mostly based on the components pairs, which are not always as efficient as the partitions found by K-way partitioning. This can be seen when comparing the communication costs of P-Scheduler and T3-Scheduler for the homogeneous configuration, where P-Scheduler is better able to match the optimal scheduler, while T3-Scheduler has a slightly higher communication cost than the optimal scheduler for some problem sizes. Also, P-Scheduler has a slightly higher throughput for the real-world topologies for the homogeneous configuration. However, P-Scheduler is only able to be run on a homogeneous cluster, whereas T3-Scheduler has a good overall throughput for both homogeneous and heterogeneous configurations. The better performance of P-Scheduler is because it may sometimes better able to select tasks from multiple components when forming partitions to be assigned to a node. Instead, T3-Scheduler prioritises finding component pairs to ensure each node contains communicating tasks, which can be sub-optimal.

Therefore, we address this limitation of T3-Scheduler by proposing an iterative K-way partitioning algorithm, where each iteration finds partitions of a different size in a heterogeneous configuration, described in detail in the next chapter.

8.7 Conclusion

In this chapter, we presented T3-Scheduler, which efficiently places tasks within the compute nodes, resulting in lower amounts of inter-node and intra-node communication. We evaluated T3-Scheduler using three micro-benchmarks and two real-world streaming applications. The experimental results showed that T3-Scheduler outperformed OLS, improving throughput by 12–78% for two real-world applications. Fur-

ther, T3-Scheduler outperformed R-Storm in load prediction topology by 26% and achieved a similar average throughput for top frequent routes without the need for tuning.

Chapter 9

I-Scheduler

In this chapter, we present I-Scheduler, a partition-based scheduler for DAG-based data stream processing systems. This chapter is organised as follows. Section 9.1 provides an overview of I-Scheduler. The I-Scheduler algorithm is described in Section 9.2, followed by an example in Section 9.3. Section 9.4 presents a comparison of I-Scheduler with an optimal scheduler using three micro-benchmarks, previously described in Section 6.2. Section 9.5 evaluates I-Scheduler with two real-world applications, previously presented in Section 6.3, with a detailed discussion in Section 9.6. Finally, Section 9.7 concludes the chapter.

9.1 Introduction

I-Scheduler is an online scheduling algorithm for DAG-based DSPSs, run on homogeneous and heterogeneous clusters, which reduces the size of the task graph by fusing highly communicating tasks. This reduced task graph size allows a mathematical optimisation software package to find a more efficient task assignment, improving the overall performance. In cases where the optimisation software cannot be used, (e.g., too expensive in terms of execution time) a fallback heuristic method fuses all of the tasks based upon cluster node capacity, directly assigning the tasks to nodes with a relative capacity. The contributions covered within this chapter are summarised as follows:

- We present I-Scheduler, a scheduling algorithm that iteratively uses graph partitioning to reliably find groups of communicating tasks, which are then fused into single larger tasks, based upon the available capacity of computing nodes within

the cluster. This novel approach of fusing large groups of communicating tasks reduces the task graph size, allowing optimisation software to be used.

- The communication cost of I-Scheduler is evaluated by comparing it to a theoretically optimal scheduler, implemented in CPLEX, when run on three micro-benchmarks, each representing a different communication pattern, previously described in Section 6.2. The evaluation shows that I-Scheduler can achieve results that are close to optimal in a number of different cluster configurations.

- We implement I-Scheduler in Apache Storm 1.1.1 and through experimental results, we show that I-Scheduler can outperform state-of-the-art R-Storm (Peng *et al.*, 2015) and OLS (Aniello *et al.*, 2013), previously described in Chapter 3. The results show that I-Scheduler outperforms OLS, increasing throughput by 15–86% and R-Storm by up to 32% for the real-world applications, previously described in Section 6.3.

9.2 I-Scheduler algorithm

As presented in Chapter 4, an optimal solution to the scheduling problem can be formulated. However, it is often not practical to use optimisation software to find such a solution for large problem sizes, due to the large search space and computational complexity. One approach is to reduce the size of the task graph by fusing each pair of tasks into a single task (Chu *et al.*, 1980). However, this has a number of drawbacks: Firstly, fusing task pairs takes a localised view of the task communications and may be unreliable at placing communicating tasks within the same node. Secondly, this approach may not be able to scale to larger problem sizes as it is only able to halve the task graph size.

To address these problems, I-Scheduler, an iterative graph partitioning-based heuristic algorithm, reduces the number of tasks for a given problem such that it can be solved in a specified scheduling time by the optimisation software.

I-Scheduler consists of four main steps as follows.

1. **Monitoring:** To operate online, I-Scheduler monitors the communication between tasks, measuring the tasks' load and data transfer rates.

2. **Constructing a weighted graph:** The online profile from the previous step is then used to build a weighted graph, providing I-Scheduler with a global view

Algorithm 4 Pseudo-code for I-Scheduler algorithm

```
1: θ ← time allowed for scheduling
2: if θ > 0 then
3:     use_optimisation_software ← True
4:     φ ← FIND_THRESHOLD(θ)  ▷ Find the threshold task size based on θ from previous experiments
5: else
6:     use_optimisation_software ← False
7:     φ ← 0
8: end if
9: SCHEDULE(nodes, g, φ, T)
10: function SCHEDULE(nodes, g, φ, T)
11:     task_map ← FIRST_LEVEL(nodes, g, φ)
12:     for each t_m in task_map do
13:         sbg ← t_m.tasks
14:         n ← t_m.node
15:         SECOND_LEVEL(n, sbg, T)
16:     end for
17: end function
18: function FIRST_LEVEL(nodes, g, φ)
19:     i ← 1                                              ▷ Iteration number
20:     t ← FIND_TASK_SIZE(g)                              ▷ Number of tasks in g
21:     f ← 0                                              ▷ Number of fused tasks
22:     while t + f > φ do
23:         k_i ← ⌈ t / N[i].c ⌉
24:         if k_i = 1 then
25:             FUSE_TASKS(g, 1)          ▷ Fuse t remaining tasks in g into a single task
26:             Break to line 43
27:         end if
28:         if k_i > φ and use_optimisation_software then
29:             use_optimisation_software ← False
30:             φ ← 0
31:         end if
32:         GRAPH_PARTITION(g, k_i)          ▷ Partition g into k_i parts on unfused vertices
```

Algorithm 5 Pseudo-code for I-Scheduler algorithm - Part 2

```
33:         if use_optimisation_software then
34:             α ← min(⌈(t−φ)/(N[i].c−1)⌉, N[i].q)
35:         else
36:             α ← min(k_i, N[i].q)
37:         end if
38:         FUSE_TASKS(g, α)                    ▷ Fuse the tasks in α partitions
39:         t ← t − (α × N[i].c)                ▷ Update remaining task size
40:         f ← f + α                           ▷ Update the number of fused tasks
41:         UPDATE_GRAPH(g)                     ▷ Update g with new vertices and edges
42:         i ← i + 1
43:     end while
44:     if use_optimisation_software then
45:         task_map ← SOLVE_MODEL(nodes, g)  ▷ Use an optimisation software to
    solve the optimisation problem
46:     else
47:         task_map ← DIRECT_MAPPING(nodes, g)  ▷ Map each fused task in g to
    a node in nodes with relative capacity
48:     end if
49:     return task_map
50: end function
51: function SECOND_LEVEL(n, sbg, T)
52:     t ← FIND_TASK_SIZE(sbg)
53:     w = ⌈t/T⌉
54:     parts ← Partition sbg into w parts using METIS
55:     Assign each part in parts to one worker on n
56: end function
```

of the task loads and communication, which is updated with new weights for vertices and edges when rescheduling.

3. **First level of scheduling:** At this level, I-Scheduler determines which groups of tasks should be co-located within each node by finding a sub-graph of highly communicating tasks from the weighted graph.

4. **Second level of scheduling:** To reduce inter-worker communication within

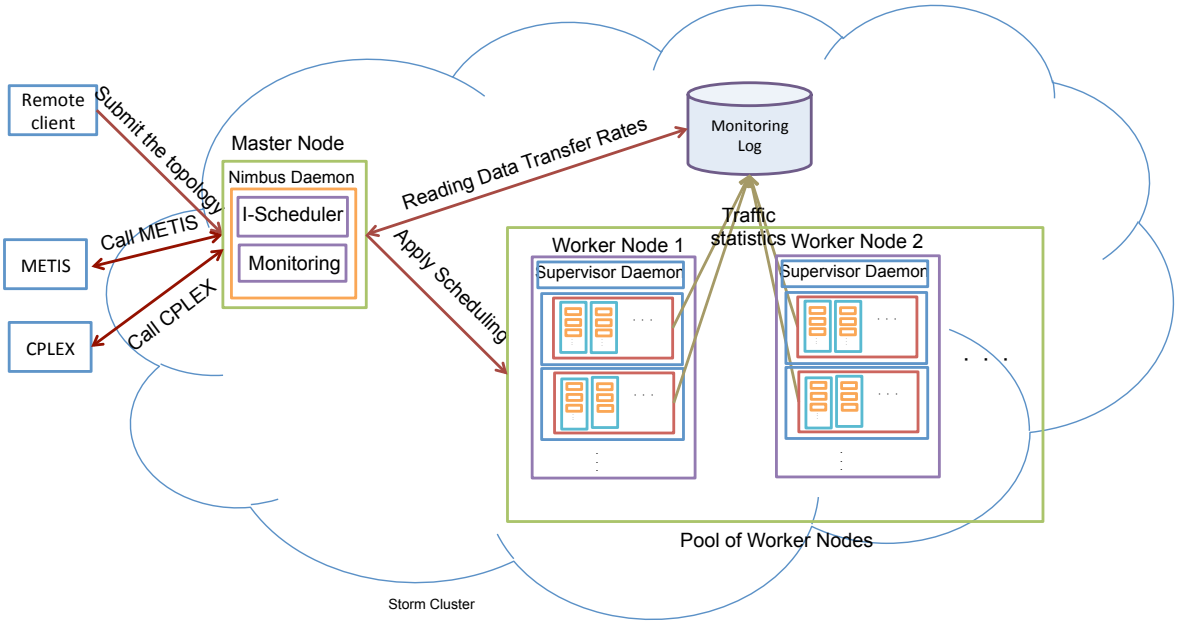


Figure 9.1: The interaction between I-Scheduler, METIS, CPLEX, monitoring log and worker nodes in a Storm cluster

each node, I-Scheduler partitions the sub-graph assigned to each node, into the required number of workers using K-way partitioning. This ensures highly communicating tasks are placed in the same worker.

Algorithm 4 presents the pseudo-code for I-Scheduler. Table 9.1 shows the notation used in this chapter for the I-Scheduler algorithm. Figure 9.1 shows the interaction between I-Scheduler, METIS, CPLEX, monitoring log and worker nodes in a Storm cluster. In the following sections, each step is discussed in detail.

9.2.1 Monitoring

Similar to monitoring step in P-Scheduler and T3-Scheduler, I-Scheduler monitors the execution of the streaming application to construct a profile of all task communications and task loads. These values are used regularly when rescheduling, to provide an up-to-date view of the application graph to perform online scheduling.

9.2.2 Constructing a weighted graph

The online profile, collected during monitoring, is used to construct a weighted graph where the vertices represent tasks and edges show the communication between tasks. Vertices are weighted by the task load and the edges are weighted by the total data

Table 9.1: Notation for I-Scheduler algorithm

Symbol	Description
g	Application graph
t	Number of unfused tasks within g
θ	A feasible amount of time to solve the optimisation problem
ϕ	Threshold task size, solvable in θ
nodes	Sorted array of available nodes (capacity, quantity)

transfer rate between the corresponding tasks. Having this view of the application graph helps I-Scheduler find and co-locate highly communicating tasks on the same node when scheduling. The weights in the graph are updated when rescheduling occurs, keeping the graph up-to-date.

9.2.3 First level of scheduling

By exploiting graph partitioning to find partitions of highly communicating tasks, sized according to node capacities, we can fuse each partition into a single task. This process is performed iteratively for each cluster node capacity until we reach a task graph size that is solvable by the optimisation software. However, in the event that the task graph is too large and complex to be sufficiently reduced in size, we can continue using the iterative task fusions until all tasks are fused and then assign each of the fused tasks to a node with a relative capacity, providing a heuristic fallback.

Since the time permitted for scheduling may vary between use-cases, a user parameter, θ, is defined for this time interval. For use-cases that tolerate longer scheduling times, a larger θ can be chosen, which will result in an improved scheduling solution as the optimisation software does more of the work. This parameter provides the ability to tune the scheduling efficiency according to the time allowed for scheduling by the use-case. Based on the value set for θ, we can then determine a task threshold ϕ based on previous experiments, such that a solution for ϕ tasks can be found in a time interval of θ. t is defined as the total number of tasks within an application graph, g. Therefore, by reducing t to ϕ, we can then use the optimisation software to find a near optimal solution. While this approach can help improve the efficiency of the schedule, it is not necessarily optimal as the steps used to reduce the problem size are heuristic. Despite this lack of a guarantee, our experimental results, presented later in this chapter, show that this approach helps to improve the performance of task scheduling.

To determine highly communicating tasks, we use K-way partitioning, previously discussed in Section 7.2. Such K-way partitioning algorithms are able to reliably find partitions of highly communicating tasks, which allows us to fuse some of these partitions into single larger tasks. This partitioning is performed iteratively, where each iteration, and corresponding graph partition size, is based upon the largest capacity node in the heterogeneous cluster. Each iteration reduces the size of t, which continues until t reaches the threshold ϕ. Note that this scheduling approach would only require one iteration on a homogeneous cluster, as all the node capacities are the same.

The graph partitioning is performed on a weighted graph such that each vertex weight is the task load and each edge weight is the data transfer rate between two communicating tasks. To simplify our formulation, we assume that a node's capacity is defined as the number of tasks it can host, however it can be defined as the sum of the CPU speed for all cores within a node. We assume that all available nodes are stored in an array, $N = \{(c_1, q_1), (c_2, q_2), \ldots, (c_j, q_j)\}$, such that each element of the array has two properties: capacity, c, and quantity, q. We use '.' notation to denote accessing each property of an element of the array, such that $N[i].c$ is the capacity of node N_i, and $N[i].q$ is the number of nodes with the given capacity in the heterogeneous cluster.

We begin by sorting the available nodes in descending order of capacity. By starting with the highest capacity nodes, we are able to assign the largest groups of tasks to these nodes, reducing the overall communication cost. In comparison, using smaller nodes results in smaller task groups which are spread across more nodes, increasing the inter-node communication cost, resulting in a less efficient schedule.

In order to partition the task graph g of size t into k_i parts of roughly size $N[i].c$, we need to calculate, k_i, for iteration i as follows:

$$k_i = \left\lceil \frac{t}{N[i].c} \right\rceil \tag{9.1}$$

After finding k_i at the start of each iteration, we compare it with ϕ as follows:

- $k_i \leq \phi$: In this case, we are able to select k_i partitions for fusion in order to reach ϕ. However, to maximise the work performed by the optimisation software, we may not require all k partitions to be fused to reach ϕ. Instead we find the minimum number of fusions required, β, to reach ϕ. By selecting β partitions, the task size t is reduced by $\beta \times N[i].c$ tasks, and increased by β as we place β fused tasks back in the graph. This relationship between β, t and ϕ is shown as:

$$\phi = t - \beta \times N[i].c + \beta \tag{9.2}$$

Therefore, the minimum number of fusions required, β out of k_i, is calculated as follows:

$$\beta = \left\lceil \frac{t - \phi}{N[i].c - 1} \right\rceil \tag{9.3}$$

While β partitions will reach ϕ, as desired, there may not be enough nodes, $N[i].q$, of the current capacity, $N[i].c$, to host the fused partitions. Therefore, we compare β with $N[i].q$ as follows:

- $\beta \leq N[i].q$: we fuse the tasks in β partitions out of k_i and we have successfully reduced t to ϕ.
- $\beta > N[i].q$: This means that the number of available nodes for the current capacity is not enough to host β fused tasks. Therefore, only $N[i].q$ partitions are fused and a further iteration of partitioning is performed on the remaining unfused tasks, where the size of each partition is the next available largest capacity node.

This decision is formulated as:

$$\alpha = \min(\beta, N[i].q) \tag{9.4}$$

The value of t is updated to the new remaining task size, calculated as: $t \leftarrow t - \alpha \times N[i].c$. A new application graph is built at each step based on the connectivity and weights from the original application graph. The task size of the new graph (which includes remaining and fused tasks) is compared to ϕ and the process of K-way graph partitioning continues until we reach ϕ.

- $k_i > \phi$: This means that even if we fuse all of the k_i partitions into k_i tasks, we will not be able to reduce the task size t to ϕ. Therefore, the optimisation software cannot solve the problem within the time interval θ. This can occur at any stage during iterative partitioning, which is resolved by setting the threshold ϕ to zero, and falling back to the heuristic iterative partitioning algorithm. We then continue with the I-Scheduler algorithm. This will result in every task in g being part of a fused group, which can fit within a given node, as there is no longer a stopping threshold. Therefore, for each iteration i, α task groups are fused and assigned to the $N[i].q$ nodes with a corresponding capacity $N[i].c$.

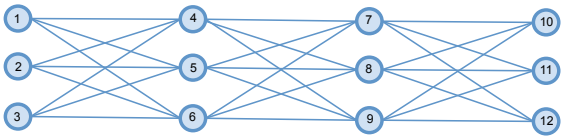
In summary, I-Scheduler uses iterative graph partitioning to reduce the size of the task graph by fusing tasks of some partitions to reach a threshold. The smaller graph size enables optimisation software to be used for task assignment. The configurable threshold determines the number of task fusions that are required, setting the amount of work to be performed by the optimisation software. The fallback I-heuristic method is used when the task graph is too large to be reduced and can be viewed as iterative graph partitioning. In this case, I-Scheduler essentially treats each step of the problem as a homogeneous cluster, using K-way partitioning to determine the groups of tasks for each node capacity. It then assigns different sized partitions of the task graph to their relative capacity nodes.

9.2.4 Second level of scheduling

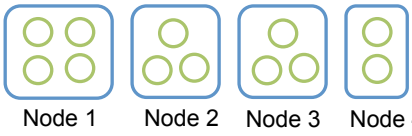
Once the first level is complete, each heterogeneous node in the cluster has been assigned a sub-graph by I-Scheduler. In the second level of scheduling, I-Scheduler decides how many workers need to be used to run the tasks within each sub-graph on a compute node. Similar to P-Scheduler and T3-Scheduler, I-Scheduler calculates the number of workers based on Equation 7.3 (see Section 7.3.4), before using K-way partitioning to divide the sub-graph assigned to each node into the number of workers, minimising inter-worker communication.

9.3 An example of task assignment by I-Scheduler

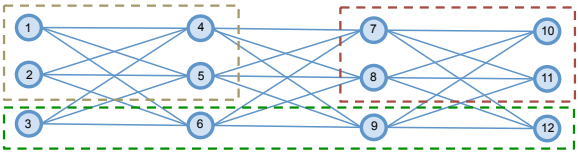
Consider the scheduling problem, with a task graph of size 12, shown in Figure 9.2a and a heterogeneous cluster, made up of 3 types of nodes, 1 node with capacity of 4, 2 nodes with capacity of 3, and 1 node with capacity of 2, shown in Figure 9.2b. The threshold ϕ is set to 8. The first iteration of I-Scheduler partitions the 12 task graph into 3 partitions of size 4, calculated by Equation 9.1 as shown in Figure 9.2c. In order to reach the threshold of 8, we need to fuse 2 out of the 3 partitions calculated by Equation 9.3. However, only 1 of these partitions can be fused, based on Equation 9.4, as we only have 1 node of capacity 4. So we fuse 1 partition and place it back into the graph as the new fused task, shown in Figure 9.2d. This reduces the unfused task size to 8 tasks, shown in the figure. The number of tasks in the new graph is 9 (including 8 unfused tasks plus 1 fused task) which is greater than the threshold of 8. Therefore, a further iteration of graph partitioning is performed on the remaining tasks, where the unfused tasks in the graph are partitioned into 3 parts of roughly



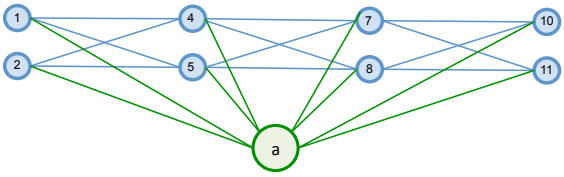
(a) The initial task graph g has 12 tasks.



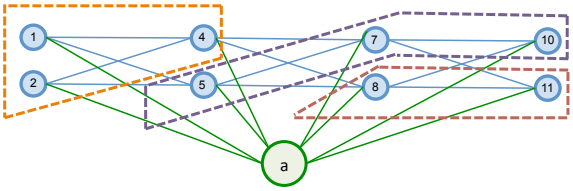
(b) The cluster consists of 1 node of capacity 4, 2 nodes of capacity 3 and 1 node of capacity 2.



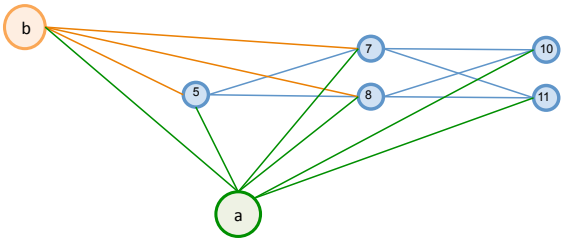
(c) Partitioning g into 3 partitions of size 4, calculated by Equation 9.1.



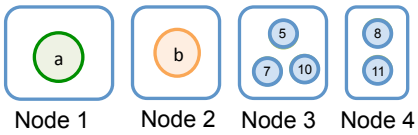
(d) Fusing 1 partition out of 3, calculated by Equation 9.4, shown in green



(e) Partitioning the remaining tasks into 3 partitions of size 3, calculated by Equation 9.1.



(f) Fusing 1 partition out of 3, calculated by Equation 9.4, shown in orange



(g) Final assignment

Figure 9.2: An example of task assignment by I-Scheduler

size 3, shown in Figure 9.2e. We fuse 1 of the 3 partitions, shown in orange in the figure, as this is all that is required to reach the threshold of 8, shown in Figure 9.2f. Finally, the optimisation software finds a solution and I-Scheduler performs the final task assignment, shown in Figure 9.2g.

9.4 I-Scheduler vs. optimal scheduler

In this section, we evaluate the communication cost of I-Scheduler, by comparing it with a theoretically optimal scheduler. This allows us to see how close to optimal

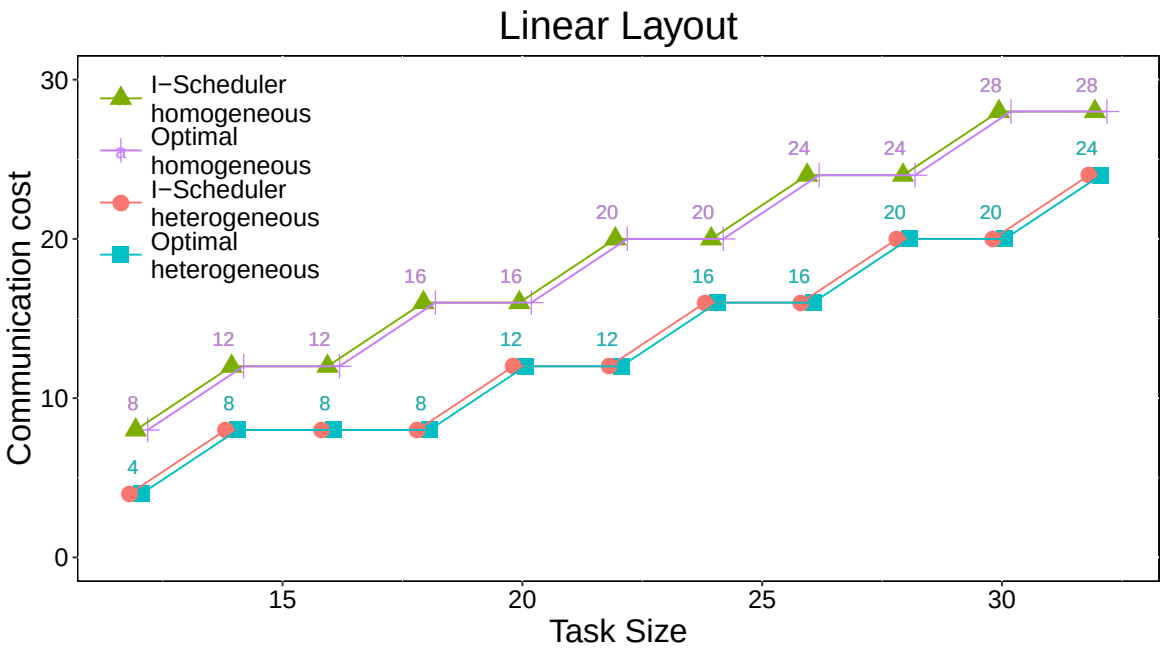


Figure 9.3: I-Scheduler vs. optimal scheduler for linear layout

the results of I-Scheduler are for the three micro-benchmarks, where each represents a different congestion pattern. Details of each micro-benchmark, such as the number of tasks and configuration were previously presented in Section 6.2. We also compare the resolution times of I-Scheduler and the optimal scheduler for each configuration. These results demonstrate how the resolution time for finding the optimal solution significantly increases as the problem size becomes larger, due to the increased computational complexity. This further demonstrates how I-Scheduler can effectively reduce the problem size, such that optimisation software can be used for task assignment, in a feasible amount of time.

Figures 9.3, 9.4 and 9.5 show the results for the linear, diamond and star micro-benchmark respectively. For the linear layout, shown in Figure 9.3, we can see that I-Scheduler is able to match the results of the optimal scheduler for both the homogeneous and heterogeneous configurations. As was noted for T3-Scheduler, the gap between the homogeneous and heterogeneous results is due to the differences in cluster capacities, where the larger nodes in the heterogeneous cluster allow more tasks to be placed in a single node, reducing the inter-node communication.

Figure 9.4 shows that I-Scheduler can achieve the same results as optimal for the diamond micro-benchmark, except for 30 tasks run on the heterogeneous cluster, where I-Scheduler has a communication cost of 148, which is two units higher than the optimal scheduler's 146. This is the result of putting highly communicating tasks from multiple

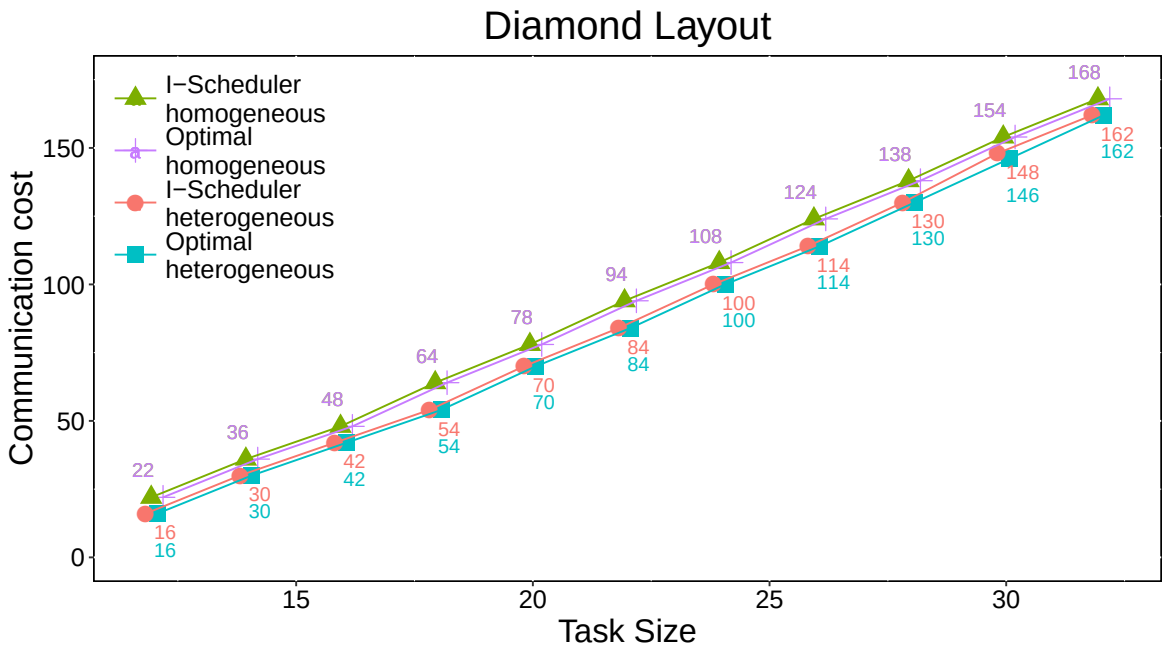


Figure 9.4: I-Scheduler vs. optimal scheduler for diamond layout

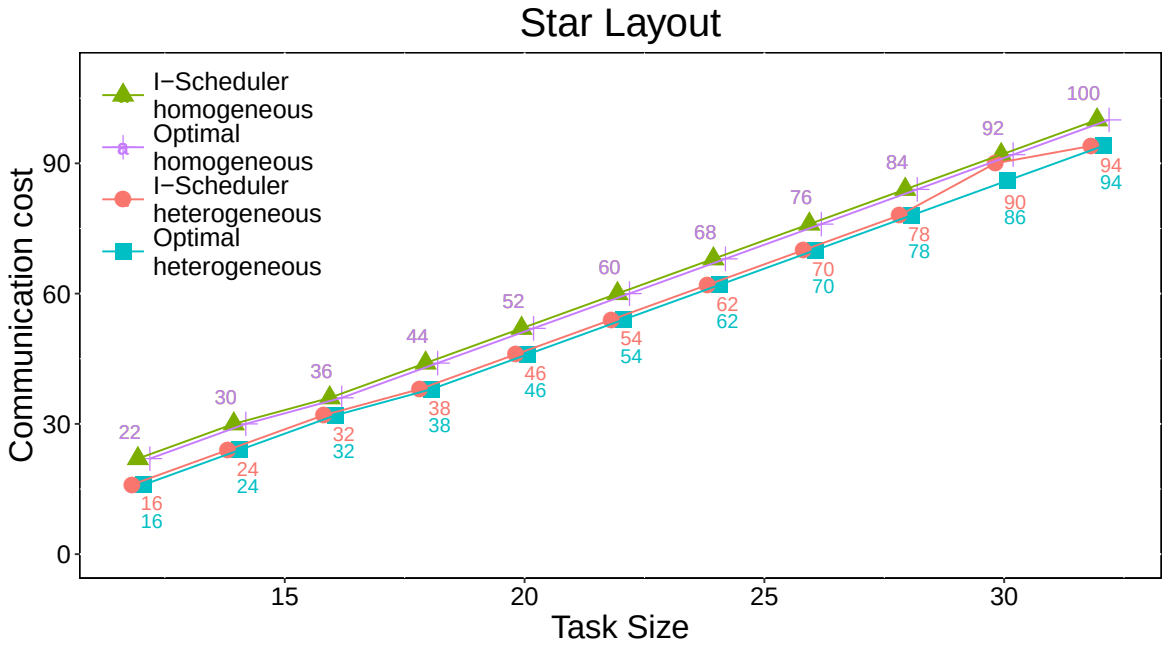


Figure 9.5: I-Scheduler vs. optimal scheduler for star layout

operators together. The optimal scheduler also selects tasks from multiple operators to achieve the optimal result.

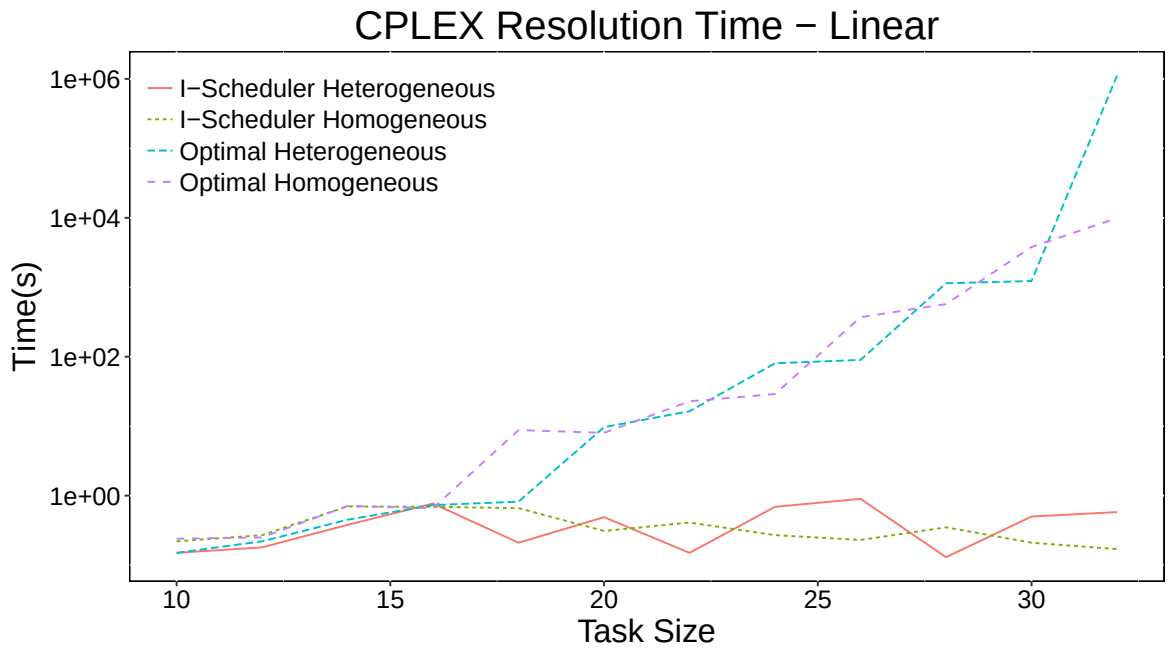


Figure 9.6: Resolution time for optimal scheduler and I-Scheduler for linear layout

Figure 9.5 shows the results for star layout for I-Scheduler and optimal scheduler. As can be seen from the figure, I-Scheduler can find the optimal solution in all cases but for the heterogeneous configuration with 30 tasks where the optimal solution is 86 and I-Scheduler finds 90.

The communication cost for diamond and start layouts, shown in Figures 9.4 and 9.5, is much higher than that of the linear layout, Figure 9.3. This is a result of the shape of each layout, where the linear shape allows more communicating tasks to be placed together within each node, while diamond and star provide fewer opportunities to place communicating tasks together. A more detailed description was previously given in Section 7.5.

Figures 9.6, 9.7 and 9.8 show the resolution times for I-Scheduler and the optimal scheduler for each of the micro-benchmarks. From the figures, it can be seen that the resolution time of the optimal scheduler increases significantly with the problem size. This is a clear demonstration showing that while the optimal scheduler can be used for small problem sizes, it quickly becomes impractical as the problem size increases. In comparison, I-Scheduler has a resolution time of less than θ for all problem sizes. While there is some minor fluctuation in the resolution times, this is a result of the problem size actually handled by the optimisation software. That is, T sets a target task size, but as entire groups are fused during each iteration, the resulting task size may be slightly smaller than T, resulting in a smaller resolution time than those with

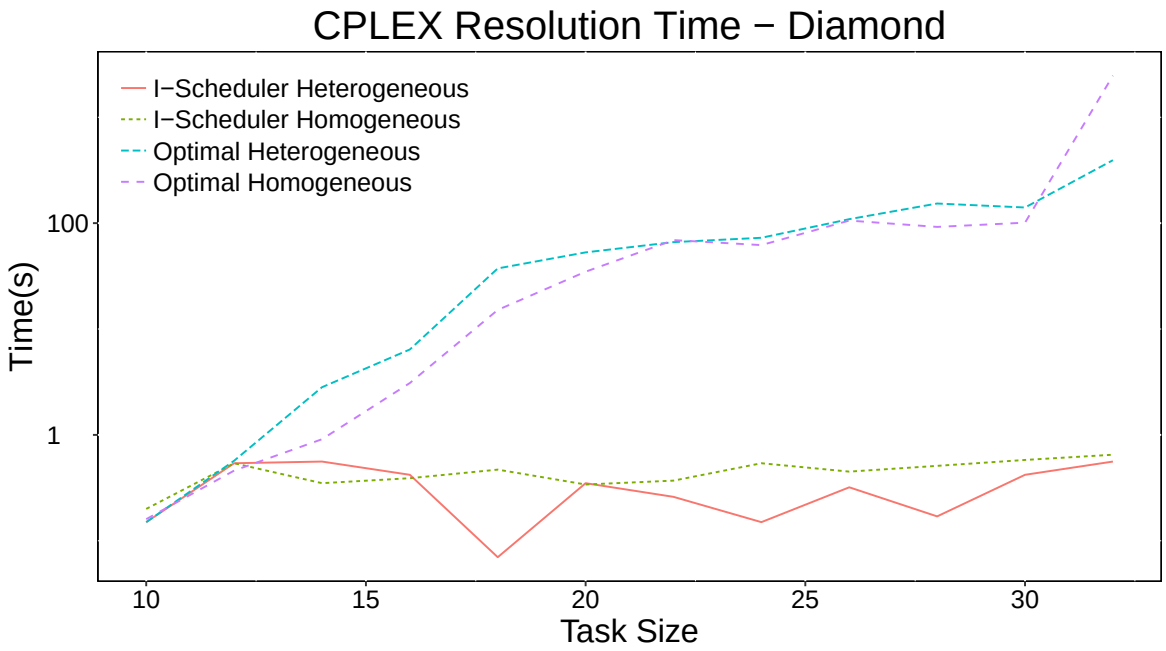


Figure 9.7: Resolution time for optimal scheduler and I-Scheduler for diamond layout

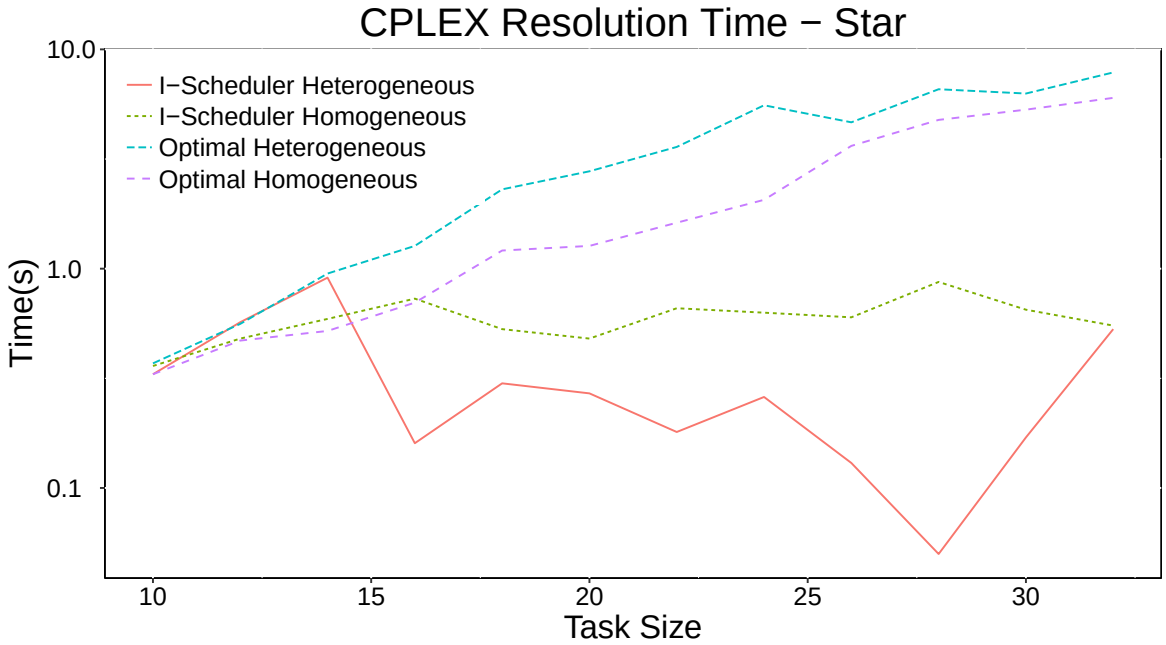


Figure 9.8: Resolution time for optimal scheduler and I-Scheduler for star layout

task sizes closer to T.

Overall, I-Scheduler is able to efficiently assign tasks to the homogeneous and het-

erogeneous cluster nodes. It was able to quickly find near optimal results for each of the three micro-benchmarks, making it a good solution for practical deployments. This approach can be used to reduce even larger scheduling problems, making them optimisation problems that can be solved in a practical time. We evaluate the practical performance of I-Scheduler in the next section.

9.5 Experimental evaluation

In this section, we evaluate I-Scheduler using three micro-benchmarks and two real-world applications, described in Section 6.2 and Section 6.3, respectively.

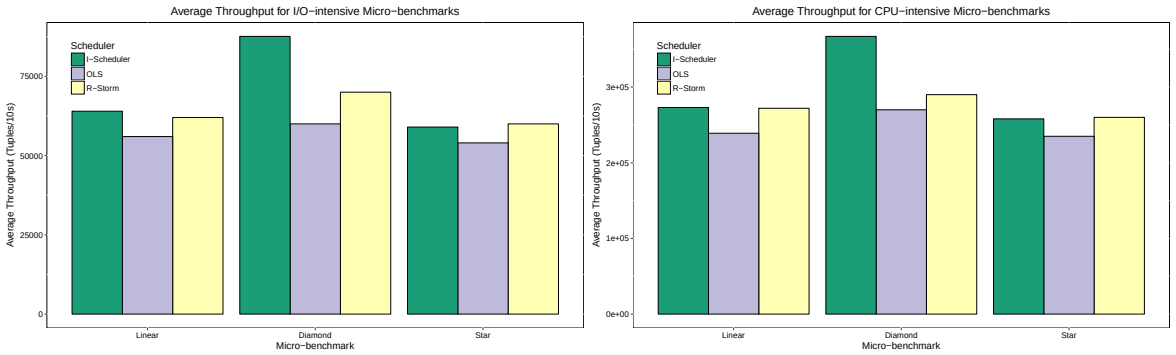
9.5.1 Micro-benchmarks

I/O-intensive: Similarly to T3-Scheduler, we evaluate the throughput of I-Scheduler for the I/O-intensive micro-benchmarks, which are run on a small heterogeneous cluster, consisting of one high capacity node and two low capacity nodes. This configuration ensures each of the three schedulers uses both high and low capacity nodes, and is not impacted by node consolidation. The results for the micro-benchmarks, run by I-Scheduler, R-Storm and OLS are shown in Figure 9.9a. As can be seen in the figure, I-Scheduler outperforms R-Storm by 25% for the diamond micro-benchmark and achieves a similar throughput for linear and star. In comparison, I-Scheduler outperforms OLS for linear, diamond and star micro-benchmarks by 8–46%, as OLS has a lower average throughput.

CPU-intensive: We then run each of the CPU-intensive micro-benchmarks on a heterogeneous cluster with two high capacity nodes and four low capacity nodes. From the results shown in Figure 9.9b, it can be seen that I-Scheduler outperforms OLS by 10–35% for all micro-benchmarks, with similar results to R-Storm for linear and star micro-benchmarks. Again, I-Scheduler has an average throughput 26% higher than R-Storm for diamond, as a result of a better task placement for I-Scheduler. The average throughput for each CPU-intensive micro-benchmark is higher than that of the I/O-intensive micro-benchmarks, due to the higher load placed on the CPU.

Overall, these results show that I-Scheduler is better able to place highly communicating tasks on nodes, reducing inter-node communication. In comparison, R-Storm is

unable to find an efficient schedule for diamond shaped layouts, while the greedy best fit approach of OLS has a lower average throughput for each micro-benchmark.



(a) I/O-intensive micro-benchmarks (b) CPU-intensive micro-benchmarks

Figure 9.9: Throughput results of I/O-intensive and CPU-intensive linear, diamond and star micro-benchmarks, using I-Scheduler, R-Storm and OLS

9.5.2 Load prediction application for smart homes

We evaluate I-Scheduler by running the load prediction topology on a homogeneous and heterogeneous cluster, which is compared with R-Storm and OLS. The homogeneous cluster is configured with 8 high capacity nodes, while the heterogeneous cluster has 2 high capacity and 4 low capacity nodes.

Figure 9.10 shows the experimental results for each of the three schedulers run on the homogeneous cluster. As can be seen from the figure, OLS and R-Storm have an average throughput of 3,700 and 5,200 respectively, while I-Scheduler with theta set to 1 has an average throughput of 6,900, representing an improvement of 86% and 30%, respectively. The lower throughput of OLS is because of the greedy approach used, which assigns tasks to the least loaded node.

The task selection algorithm of R-Storm is unable to place any of the Plug Load component's tasks with other communicating tasks, separating this component from others, lowering the throughput. Further, with R-Storm being offline, it cannot see the data transfer rates between tasks and cannot prioritise highly communicating tasks.

As a result of OLS assigning tasks to the least loaded nodes, it uses all 8 nodes while R-Storm and I-Scheduler use 4 out of 8 nodes on average. As a result, R-Storm and I-Scheduler consolidate the cluster, reducing inter-node communication. For a fairer comparison which removes the impact of node consolidation, we rerun OLS on 4 nodes, similar to the number of nodes used by R-Storm and I-Scheduler. As can be seen in

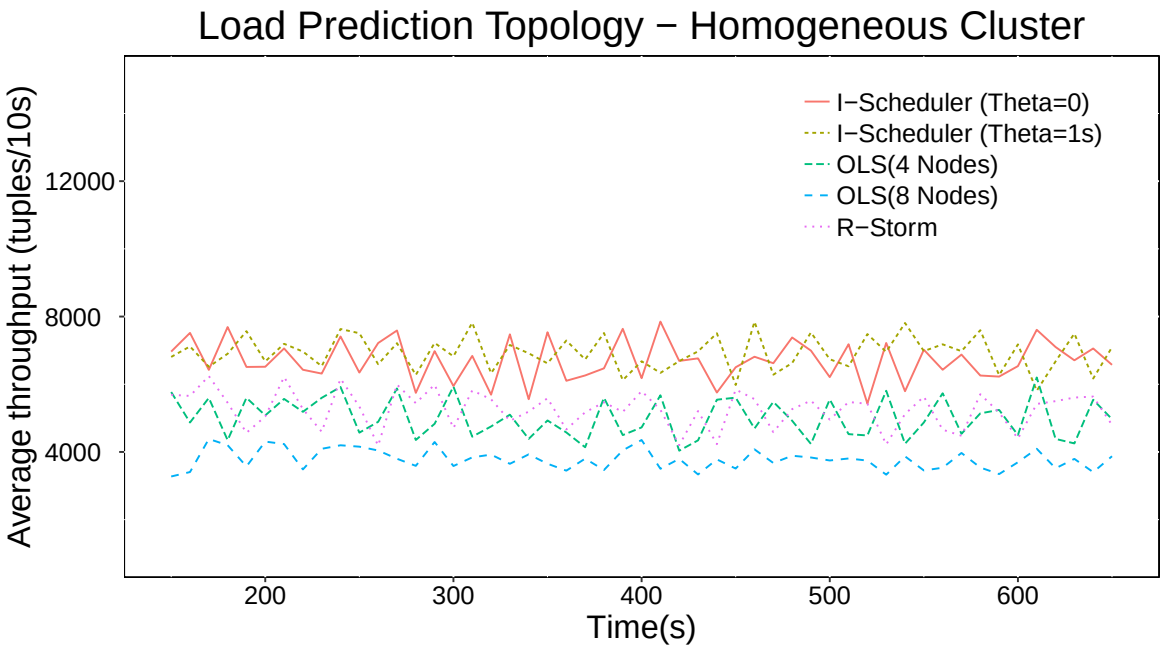


Figure 9.10: Throughput results of smart home load prediction topology in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

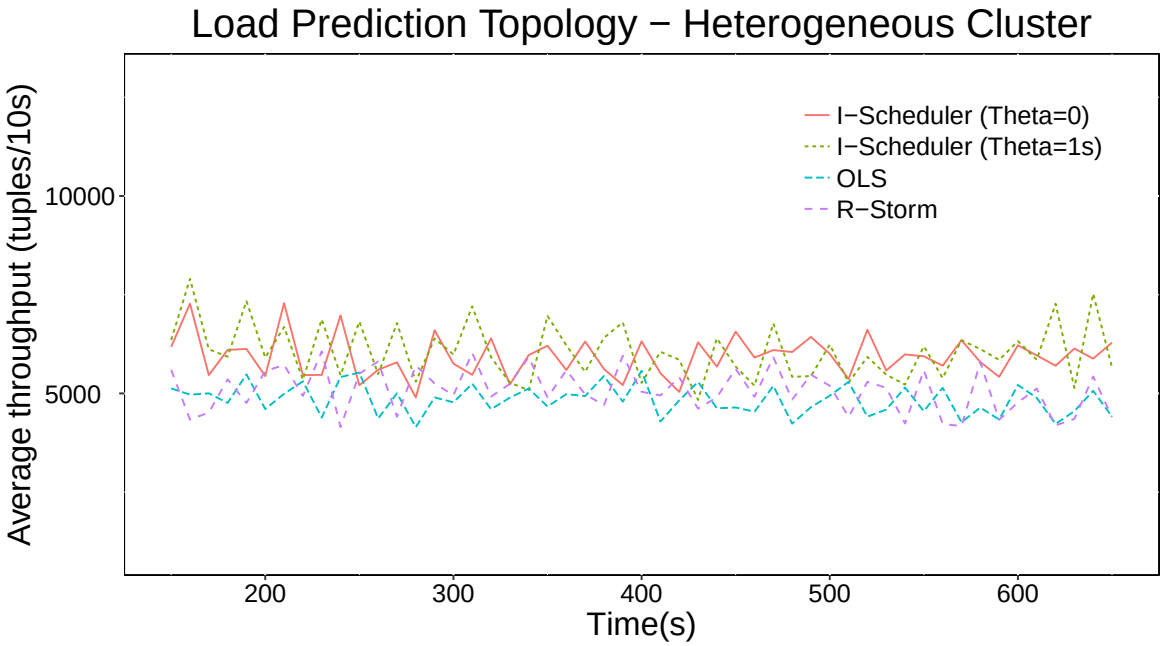


Figure 9.11: Throughput results of smart home load prediction topology in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

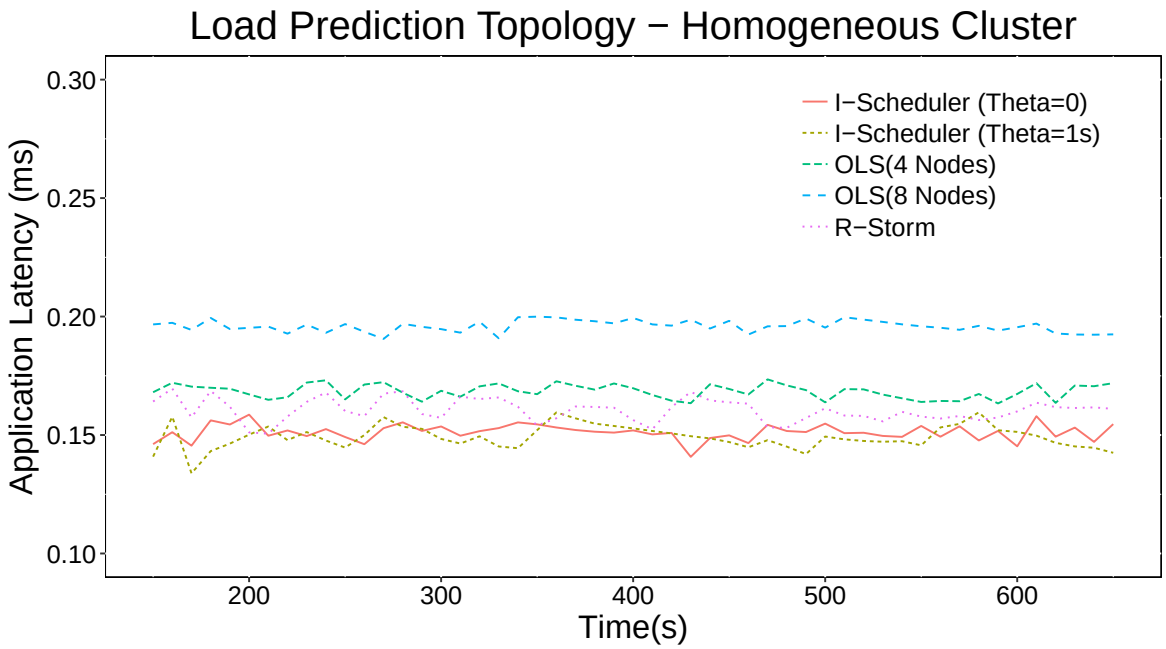


Figure 9.12: Latency results of smart home load prediction topology in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

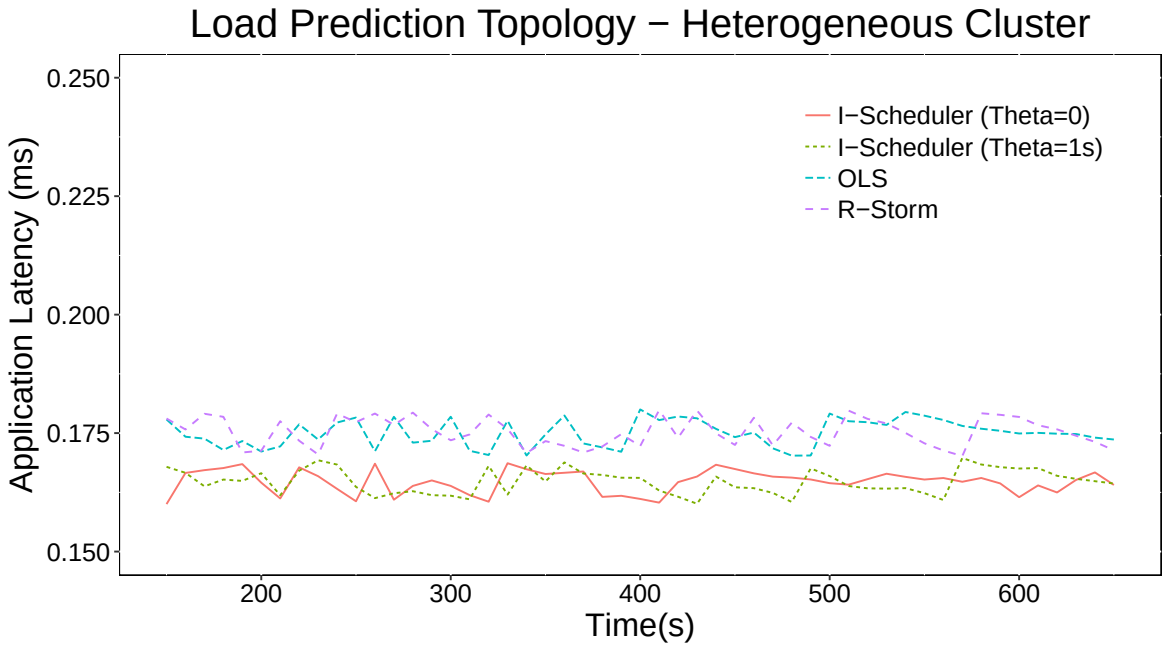


Figure 9.13: Latency results of smart home load prediction topology in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

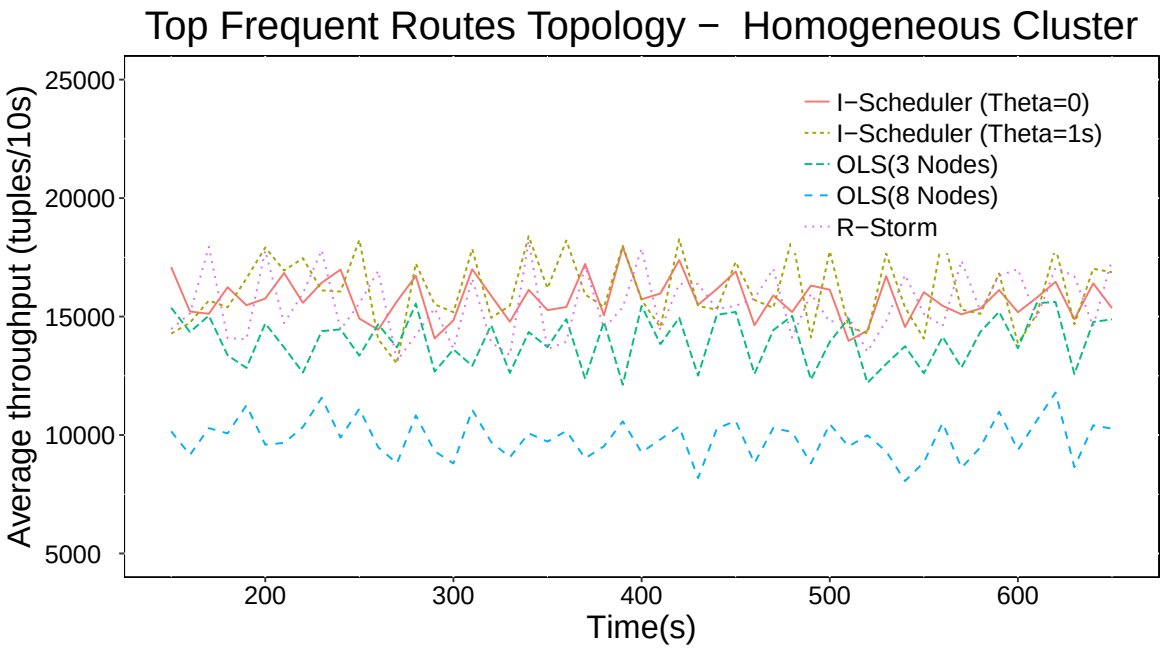


Figure 9.14: Throughput results of top frequent routes topology in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

Figure 9.10, the throughput of OLS increases when run on fewer nodes, however it is still lower than the other two schedulers.

From the figure, it can be seen that I-Scheduler with θ set to zero performs similarly to I-Scheduler with a θ of 1. This demonstrates the efficiency of the graph partitioning algorithms in finding groups of highly communicating tasks.

We then run all three schedulers on a heterogeneous cluster, consisting of 2 high capacity nodes and 4 low capacity nodes. This configuration evaluates which scheduler is better able to reduce inter-node communication by more efficiently assigning highly communicating tasks to the limited number of high capacity nodes. As can be seen in Figure 9.11, both settings of I-Scheduler can outperform R-Storm and OLS because of its task selection approach. It is worth noting that the throughput is lower for all schedulers than the homogeneous configuration, as there is more inter-node communication.

Figures 9.12 and 9.13 show the execute latency for the topology run by the three schedulers. From the figures, it can be seen that as the execute latency is reduced, we can achieve higher throughput, however the overall improvement in latency is not significant. In summary, the experimental results show that I-Scheduler can more efficiently place highly communicating tasks within the same node, resulting in a more efficient task assignment.

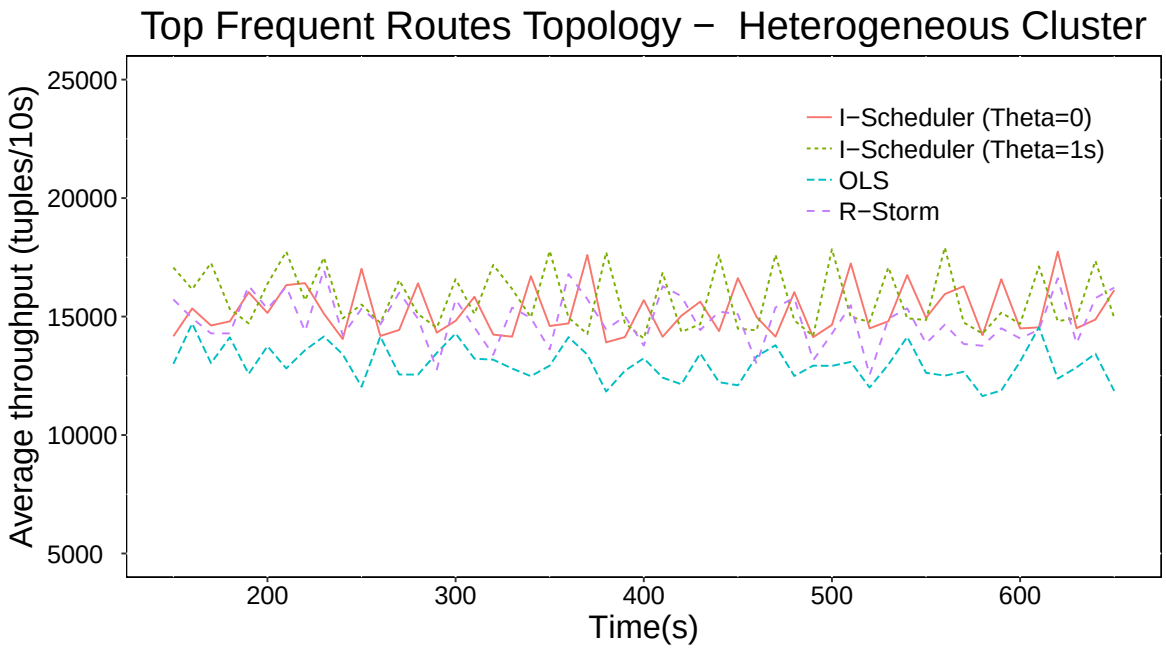


Figure 9.15: Throughput results of top frequent routes topology in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

9.5.3 Top frequent routes in NYC taxi data

Figure 9.14 shows the throughput for each of the three schedulers, running the top frequent routes topology on an 8 node homogeneous cluster. As can be seen from the figure, the average throughput for I-Scheduler, R-Storm and OLS is 16,000, 15,500 and 9,800 respectively. The results show that both I-Scheduler and R-Storm are able to efficiently place highly communicating tasks together on the same node, reducing inter-node communication. It is worth nothing that R-Storm operates offline and required substantial fine tuning to be able to achieve this results, while I-Scheduler operates online without requiring such tuning. The lower average throughput of OLS is due to tasks being spread across nodes. By rerunning OLS on a 3-node cluster, a higher throughput of 13,900 was achieved. We further evaluate each of the schedulers on a heterogeneous cluster with 2 high capacity nodes and 2 low capacity nodes. From the results, shown in Figure 9.15, it can be seen that I-Scheduler has an average throughput of 15,700 while R-Storm and OLS have an average of 14,800 and 13,000 respectively. Finally, Figures 9.16 and 9.17 show the latency for each scheduler running on the homogeneous and heterogeneous cluster configurations.

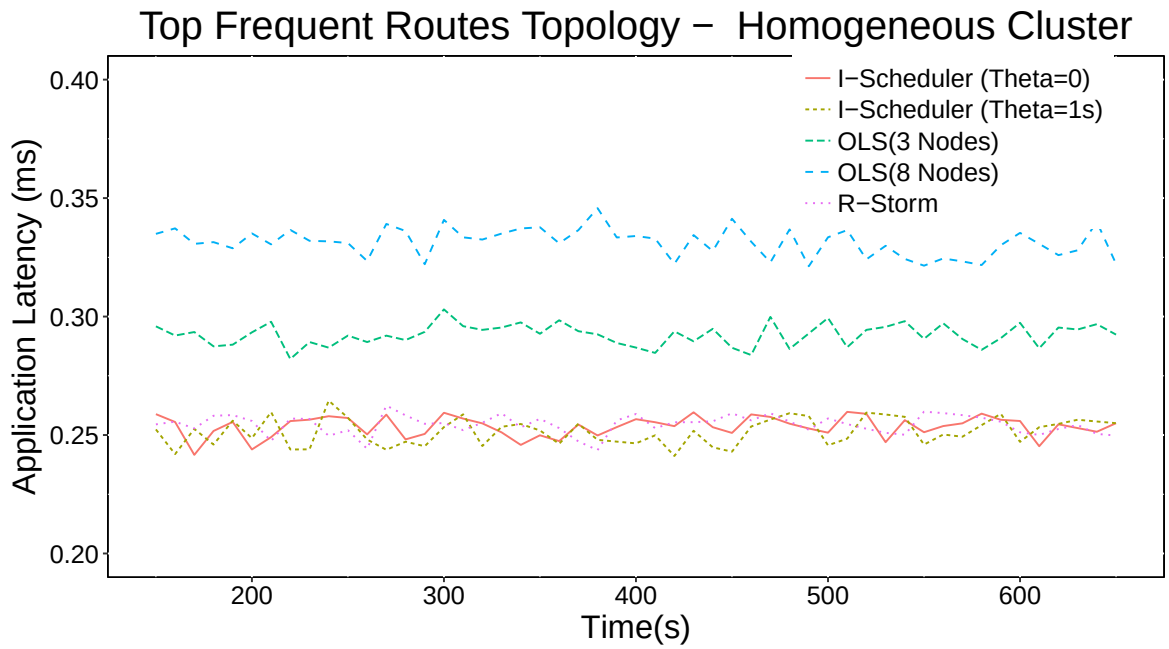


Figure 9.16: Latency results of top frequent routes topology in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

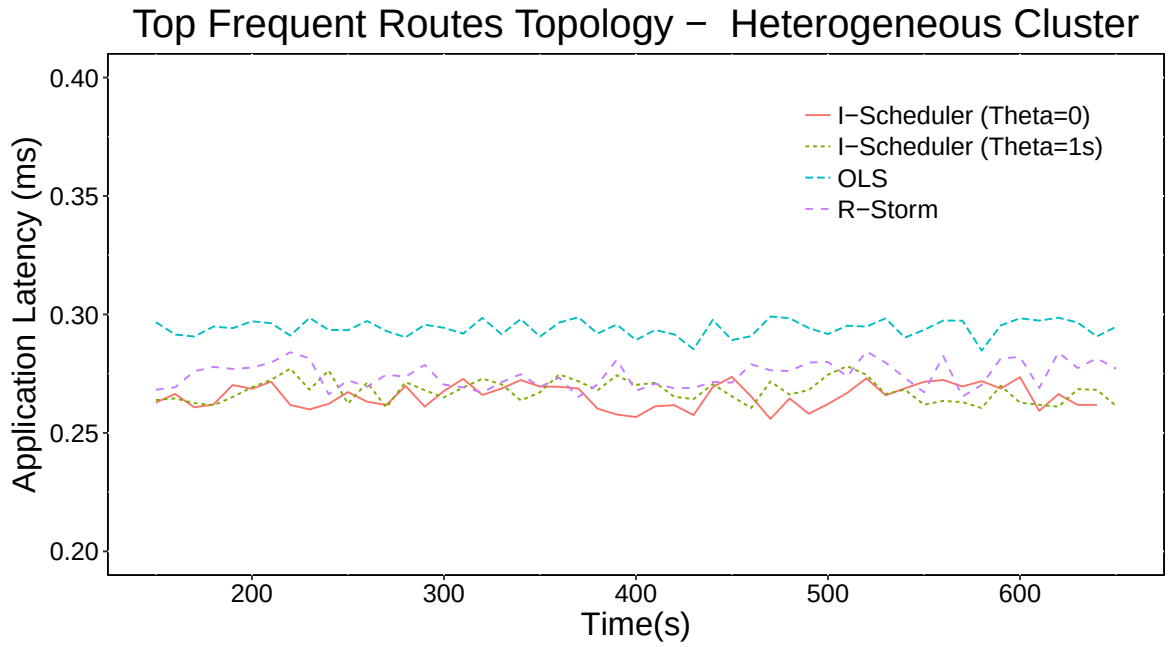


Figure 9.17: Latency results of top frequent routes topology in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

9.6 Discussion

Our evaluation of the communication cost of I-Scheduler, compared to a theoretically optimal scheduler, has shown that it is able to match, or get near optimal results for both homogeneous and heterogeneous cluster configurations. When compared to our previous schedulers, P-Scheduler and T3-Scheduler, I-Scheduler can match the results of P-Scheduler for the homogeneous configuration, and obtain a lower communication cost than T3-Scheduler for the heterogeneous and homogeneous configuration. This represents an overall improvement in performance, as I-Scheduler has achieved either optimal or near optimal results for all configurations.

This is further supported by the experimental results for the two real-world applications, where I-Scheduler shows an improved performance, compared to P-Scheduler and T3-Scheduler. This is a result of employing efficient K-way partitioning in an iterative manner to reliably find partitions of different sizes with highly communicating tasks. Further, by using optimisation software to perform task assignment after fusion, the inter-node communication is reduced.

I-Scheduler has overcome the main limitations of P-Scheduler and T3-Scheduler. While P-Scheduler had good performance for homogeneous clusters, it was unable to operate on heterogeneous clusters due to its use of a single phase of K-way partitioning. The iterative approach adopted by I-Scheduler overcomes this, resulting in equivalent performance for homogeneous clusters, while being able to efficiently operate on heterogeneous clusters. Further, the prioritisation of operator task pairs of T3-Scheduler is overcome by the use of K-way partitioning, which reliably finds partitions of tasks across multiple operators.

Similar to P-Scheduler and T3-Scheduler, I-Scheduler is an online scheduler which has an overhead when rescheduling. In our implementation of the proposed schedulers in Apache Storm, this means using native Storm task migration. Currently, Storm uses a simple method of task migration, where the execution is stopped, allowing the tasks to be moved, before restarting execution with the new configuration. In our future work, we will investigate methods for smooth task migration, which does not stop the entire execution and this will reduce the overhead of rescheduling.

9.7 Conclusion

In this chapter, we presented I-Scheduler, a heuristic scheduling algorithm that utilises graph partitioning to efficiently and reliably fuse groups of communicating tasks, reducing the graph size such that it can be solved by optimisation software. We evaluated I-Scheduler using three micro-benchmarks and two real-world streaming applications. The experimental results showed that I-Scheduler outperformed R-Storm by up to 32% and OLS by 15–86%, when run on two real-world applications.

Chapter 10

Conclusion and future work

10.1 Conclusion

It is essential to have an efficient data stream processing system for real-time streaming applications such as real-time analysis, network monitoring, security surveillance and real-time search. An important aspect to having such a near real-time system is an effective scheduler. Since it is NP-hard to determine the optimal schedule, many heuristic scheduling algorithms have been proposed, which are used in existing systems to find efficient schedules. However, existing schedulers still have some scope for improvement, such as a more reliable placement of communicating tasks on nodes or improved QoS.

In this thesis, we have focused on the scheduling problem. We began by providing background on three different generations of data stream processing systems. Then, we presented work related to task scheduling in distributed stream processing systems. With this, we highlighted the challenges of scheduling and the limitations of existing approaches adopted in the state-of-the-art. To address these challenges, we have proposed three scheduling policies which partition a streaming application graph into groups of highly communicating tasks. This has the objective of minimising the inter-node communication when each partition is assigned to a node, which can increase system throughput and reduce latency. The three schedulers, P-Scheduler, T3-Scheduler and I-Scheduler are described as follows:

- **P-Scheduler** employs graph partitioning to efficiently and reliably partition the application graph into the number of nodes required to run the application. The partitions in P-Scheduler are of roughly equal sized groups of highly communicating tasks, which are assigned to homogeneous nodes, reducing the inter-node

communication. P-Scheduler further divides each partition/sub-graph assigned to a node into the number of required workers, minimising inter-worker communication.

- **T3-Scheduler** finds partitions of different sizes in the application graph such that highly communicating tasks are grouped together. By repeatedly finding a starting point within the application graph, and expanding the search outwards from this point to include the most highly communicating neighbours, T3-Scheduler can partition the application graph where each part is sized relative to the capacity of a node. Once the partitions are assigned to the heterogeneous nodes, they are further partitioned into the number of workers required for the node, similarly to P-Scheduler.

- **I-Scheduler** iteratively uses graph partitioning to efficiently fuse groups of highly communicating tasks into a single task, in order to reduce the task size of the application graph. Once the task graph is sufficiently small, optimisation software is able to determine the task assignment within a user defined time tolerance. In cases where the task graph is too large to be solved by the optimisation software or the user cannot tolerate any delays in finding a schedule, a fallback heuristic method is used, where each fused task is assigned to a node with a relative capacity. I-Scheduler performs a second level of graph partitioning on each node, to minimise the inter-worker communication, similarly to P-Scheduler and T3-Scheduler.

The key advantages of our algorithms are that they operate online, are able to consolidate the number of nodes used, and by adopting our novel two-level scheduling method, they are able to reduce intra-node communication as well as the inter-node communication, reducing latency and increasing throughput. However, each algorithm requires rescheduling, and will incur some additional overhead when rescheduling occurs. Further, there are some limitations to each scheduler. P-Scheduler relies on the efficiency of the K-way graph partitioning algorithm. T3-Scheduler has a group pair view which means it works better with applications where aggregation rates are reduced as data flows through the system. The efficiency of the algorithm is also dependent upon the selection of the starting point for the task grouping, where an inefficient selection may give a sub-optimal result. I-Scheduler relies on the efficiency of the K-way graph partitioning algorithm and optimisation software such as CPLEX. It is also dependent on choosing the right partitions for fusion during each iteration.

Table 10.1: A comparison of our proposed schedulers: P-Scheduler, T3-Scheduler and I-Scheduler

Scheduling Algorithm	**Solution Approach**	**Optimisation Goal**	**Dynamic**	**Heterogeneity**
P-Scheduler	K-way partitioning	Reducing inter-node communication	✓	×
T3-Scheduler	Task grouping	Reducing inter-node communication	✓	✓
I-Scheduler	- Iterative K-way partitioning - Use opt. software	Reducing inter-node communication	✓	✓

A further limitation of each algorithm is that rescheduling is based upon recent traffic patterns, without the use of prediction. While this approach is common, it can result in inefficient schedules when the application has bursty traffic patterns which continually change. To handle such bursty traffic, rescheduling would have to be repeatedly performed. This is a worst case scenario, which is common to schedulers that do not use prediction. In this thesis, the experimental applications used to evaluate all of the schedulers had a stable traffic pattern, and did not test this worst case scenario.

A summary of each scheduler is shown in Table 10.1, which can be compared to the related work previously provided in Table 3.2.

From these limitations, each scheduler has advantages and disadvantages compared to each other. For instance, P-Scheduler is only able to operate on a homogeneous cluster, but may be more reliable at finding partitions on such a cluster than T3-Scheduler. However, T3-Scheduler is able to find task groups on both homogeneous and heterogeneous clusters, achieving better overall performance and usability than P-Scheduler. I-Scheduler is able to improve on both P-Scheduler and T3-Scheduler by using optimisation software when determining the task placement, but the scalability of the optimisation software is limited. However, with the fallback heuristic method, I-Scheduler achieves better overall performance.

While the run-time of each algorithm depends on the properties of the application graph being scheduled, we can calculate the worst case run-time. P-Scheduler builds $G = (V, E)$ from all of the communicating tasks, obtained from the monitoring step in $O(|E|)$, and finds the number of required partitions, k, in $O(1)$. The graph G is then partitioned into k partitions using a K-way partitioning algorithm, taking $O(|E|logk)$. This procedure is then repeated for the second level of partitioning, where the maximum number of edges per node is assumed to be $|E'|$ and the maximum number of partitions per node is considered to be k', taking $O(|E'|logk')$. The worst case for T3-Scheduler

is when there is only one task per operator, meaning there is no task grouping. The first step is to prioritise the N nodes, based on their capacity, taking $O(|N|logN)$, before prioritising the task pairs based on their communication rates, $O(|E|logE)$. Each node is then filled by finding a starting point in the graph from the ordered list of task pairs, where this sub-graph is extended until the node is full. The second level of scheduling in T3-Scheduler uses the same procedure as was previously described for P-Scheduler, and has a run-time of $O(|E'|logk')$. In cases where I-Scheduler uses optimisation software, it has a run-time of $O(\theta)$, which is the user-defined parameter for the scheduling time threshold, previously described in Section 9.2. In other cases, when the fallback method is used, I-Schedulers performs iterative graph partitioning based on the node capacities, meaning the worst case occurs when every node has a different capacity, requiring more iterations to be performed. For iteration i, I-Scheduler finds the required number of partitions, k_i, in $O(1)$. The graph is then partitioned into k_i parts in $O(|E_i|logk_i)$, with a worst case run-time of $O(|N||E|logk)$.

We evaluated the communication cost of each of the proposed schedulers by comparing them with a theoretically optimal scheduler. Our evaluation shows that our proposed schedulers can achieve the results close to optimal. We further performed experiments comparing our proposed schedulers with two popular and open source schedulers, R-Storm and OLS, using two real-world applications. The results show that our schedulers can outperform OLS by 12–86% and R-Storm up to 32%.

10.2 Future work

In this thesis we have presented three heuristic scheduling algorithms that improve on the task assignment of streaming applications in DSPSs, run on homogeneous and heterogeneous clusters. However, a number of challenges remain to be addressed in our future work, as follows.

- **Extended evaluation:** We plan to implement new micro-benchmarks and real-world applications which have different communication and congestion patterns to those already implemented. This will allow us to further evaluate our proposed schedulers: P-Scheduler, T3-Scheduler and I-Scheduler.

- **Load prediction:** We plan to further investigate the observed run-time characteristics of each application, such as the burstiness of some communication, to gain further insights. These insights could be used to develop a predictive

scheduling algorithm that can preemptively adapt to expected changes in the workload characteristics, such as a bursty workload phase. Predicting the workload can help overcome an existing limitation of our work by better handling the worst case bursty traffic.

- **Rescheduling frequency:** A further area of interest is determining when rescheduling should optimally occur for a given workload. This work would build on the analysis and prediction of the load, by using these insights to guide rescheduling, as the scheduler should respond to the dominant workload patterns, ensuring scheduling is not based on infrequent bursts in traffic. Additionally, a balance needs to be struck between the overhead of rescheduling, which causes a temporary pause in execution, and the potential performance gains of a new schedule.

- **State migration:** We will investigate improved methods for state migration of tasks, as surveyed in (To *et al.*, 2018), which would allow for more efficient rescheduling. Currently, Apache Storm's state migration stops all tasks from executing to allow them to be moved, before being restarted. This is inefficient as it interrupts execution, and causes significant overhead. There is great interest in finding a solution to this problem, which is an area of ongoing research. By implementing an improved method of state migration, we would be able to have a more efficient scheduler, with lower overhead, enabling more frequent rescheduling to be performed in response to bursty traffic.

- **Parallelism degree:** We plan to work on finding the optimal number of tasks per operator for a streaming application and determine how those tasks are distributed during execution. As the number of tasks is assigned at the start of execution, and remains fixed for the entire execution, load prediction is used to determine the number of tasks required by each operator in order to handle the load. Further, during execution, the parallelism degree is determined by the ratio of tasks per executor, where increasing the number of executors, which decreases the tasks per executor, results in increased parallelism with more tasks executing in parallel. While increasing the parallelism degree can improve performance, it comes at the cost of rescheduling overhead. Some of this cost can be overcome with the development of effective load prediction, rescheduling frequency analysis and minimal cost state migration, which will help ensure the performance gain out weight any overhead.

- **Multi-application scheduling:** Currently, our schedulers assign applications to clusters based on the assumption that the single application has exclusive access to the compute resources. However, this will not always be the case, where multiple-applications will share the compute resources of large clusters. Extending our schedulers to assign multiple-applications will further require considering fairness, QoS and the relative needs of each application. This may require an additional management layer during scheduling, or a new, adapted scheduling approach.

- **Theoretical analysis:** We will analyse the trend in communication cost for each algorithm and study the differences between the proposed schedulers and the optimal scheduler. This will involve determining the scalability of each algorithm for much larger problems with more complicated traffic pattern, and the relative impact on communication cost.

References

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., *et al.* (2005). The design of the Borealis stream processing engine. In *CIDR*, Volume 5, 277–289.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. *the VLDB Journal*, *12*(2), 120–139.

Abrams, Z. and Liu, J. (2006). Greedy is good: On service tree placement for in-network stream processing. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 72–72. IEEE.

Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C. (2006). SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, 27–37. ACM.

Anderson, Q. (2013). *Storm real-time processing cookbook*. Packt Publishing Ltd.

Andrade, H., Gedik, B., Wu, K.-L., and Yu, P. S. (2011). Processing high data rate streams in System S. *Journal of Parallel and Distributed Computing*, *71*(2), 145–156.

Aniello, L., Baldoni, R., and Querzoni, L. (2013). Adaptive online scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, 207–218.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). STREAM: The stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, 665–665. ACM.

Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, *15*(2), 121–142.

Aslett, M. (2011). How will the database incumbents respond to NoSQL and NewSQL.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1–16. ACM.

Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Load management and high availability in the Medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 929–930. ACM.

Balkesen, C., Tatbul, N., and Özsu, M. T. (2013). Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, 15–26. ACM.

Bellavista, P., Corradi, A., Reale, A., and Ticca, N. (2014). Priority-based resource scheduling in distributed stream processing systems for big data applications. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 363–370. IEEE.

Buddhika, T., Stern, R., Lindburg, K., Ericson, K., and Pallickara, S. (2017). Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, *28*(12), 3553–3569.

Caneill, M., El Rheddane, A., Leroy, V., and De Palma, N. (2016). Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference*, 4. ACM.

Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2015). Distributed QoS-aware scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, 344–347. ACM.

Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2016). Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, 69–80. ACM.

Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2017). Optimal operator replication and placement for distributed stream processing systems. In *ACM SIGMETRICS Performance Evaluation Review*, Volume 44, 11–22. ACM.

Carney, D., Cetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very large Data Bases*, Volume 29, 838–849. VLDB Endowment.

Cetintemel, U. (2003). The Aurora and Medusa projects. *Data Engineering*, *51*(3).

Cetintemel, U., Abadi, D., Ahmad, Y., Balakrishnan, H., Balazinska, M., Cherniack, M., Hwang, J.-H., Madden, S., Maskey, A., Rasin, A., *et al.* (2016). The Aurora and Borealis stream processing engines. *Data Stream Management*, 337–359.

Chakraborty, R. and Majumdar, S. (2017). Priority based resource scheduling techniques for a resource constrained stream processing system. In *Proceedings of the 4th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 21–31. ACM.

Chakravarthy, S. and Jiang, Q. (2009). *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*, Volume 36. Springer Science & Business Media.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, 668–668. ACM.

Chatzistergiou, A. and Viglas, S. D. (2014). Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management*, 1579–1588. ACM.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. B. (2003). Scalable distributed stream processing. In *CIDR*, Volume 3, 257–268.

Chu, W. W., Holloway, L. J., Lan, M.-T., and Efe, K. (1980). Task allocation in distributed data processing. *IEEE Computer*, *13*(11), 57–69.

Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, *44*(3), 15.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Dinsmore, T. W. (2016). *Disruptive analytics: Charting your strategy for next-generation business analytics*. Springer.

Eidenbenz, R. and Locher, T. (2016). Task allocation for distributed stream processing. In *Proceedings of 35th IEEE International Conference on Computer Communications (INFOCOM)*, 1–9. IEEE.

Eskandari, L., Huang, Z., and Eyers, D. (2016). P-Scheduler: Adaptive hierarchical scheduling in Apache Storm. In *Proceedings of the Australasian Computer Science Week Multiconference*, 26. ACM.

Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2017). A Topology and Traffic Aware Two-Level Scheduler for stream processing systems in a heterogeneous cluster. In *Proceedings of the 23rd European Conference on Parallel Processing*, 68–79. Springer.

Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2018a). Iterative scheduling for distributed stream processing systems. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, 234–237. ACM.

Eskandari, L., Mair, J., Huang, Z., and Eyers, D. (2018b). T3-Scheduler: A Topology and Traffic aware Two-level Scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems*, *89*, 617–632.

Etzion, O., Niblett, P., and Luckham, D. C. (2011). *Event processing in action*. Manning Greenwich.

Fischer, L. and Bernstein, A. (2015). Workload scheduling in distributed stream processors using graph partitioning. In *Proceedings of the IEEE International Conference on Big Data (Big Data)*, 124–133. IEEE.

Fu, T. Z., Ding, J., Ma, R. T., Winslett, M., Yang, Y., and Zhang, Z. (2015). DRS: Dynamic Resource Scheduling for real-time analytics over fast streams. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)*, 411–420. IEEE.

Gary, M. R. and Johnson, D. S. (1983). Computers and intractability: A guide to the theory of NP-completeness. *Journal of Symbolic Logic*, *48*(2), 498–500.

Gautam, J. V., Prajapati, H. B., Dabhi, V. K., and Chaudhary, S. (2015). A survey on job scheduling algorithms in big data processing. In *International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 1–11. IEEE.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). SPADE: The System S declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 1123–1134. ACM.

Ghaderi, J., Shakkottai, S., and Srikant, R. (2016). Scheduling storms and streams in the cloud. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, *1*(4), 14.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). *The Google file system*, Volume 37. ACM.

Grolinger, K., Higashino, W. A., Tiwari, A., and Capretz, M. A. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, *2*(1), 22.

Guo, C., Wu, H., Tan, K., Shiy, L., Zhang, Y., and Luz, S. (2004). MapReduce: Simplified data processing on large clusters. In *OSDI*, Volume 51, 107–113.

He, C., Lu, Y., and Swanson, D. (2011). Matchmaking: A new mapreduce scheduling technique. In *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science (CloudCom)*, 40–47. IEEE.

Heinze, T., Aniello, L., Querzoni, L., and Jerzak, Z. (2014). Tutorial: Cloud-based Data Stream Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, 238–245. ACM.

Hidalgo, N., Wladdimiro, D., and Rosas, E. (2017). Self-adaptive processing graph with operator fission for elastic stream processing. *Journal of Systems and Software*, *127*, 205–216.

Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, *4*(11), 1123–1134.

Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. (2003). A comparison of stream-oriented high-availability algorithms. Technical report, TR-03-17, Computer Science Department, Brown University.

InfoSphere (2018). An introduction to InfoSphere Streams. http://www.ibm.com/developerworks/library/bd-streamsintro/.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Volume 41, 59–72. ACM.

Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. (2009). Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, 261–276. ACM.

Jain, A. (2017). *Mastering Apache Storm: Real-time big data streaming using Kafka, Hbase and Redis*. Packt Publishing.

Jain, A. and Nalya, A. (2014). *Learning Storm*. Packt Publ.

Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. (2006). Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of data*, 431–442. ACM.

Jankowski, M., Pathirana, P., and T. Allen, S. (2015). *Storm applied: Strategies for real-time event processing*. Manning Publications.

Jiang, Y., Huang, Z., and Tsang, D. H. (2016). Towards max-min fair resource allocation for stream big data analytics in shared clouds. *IEEE Transactions on Big Data*.

Kalyvianaki, E., Wiesemann, W., Vu, Q. H., Kuhn, D., and Pietzuch, P. (2011). SQPR: Stream query planning with reuse. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, 840–851. IEEE.

Karypis, G. and Kumar, V. (1998). Multilevel K-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, *48*(1), 96–129.

Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B. (2009). COLA: Optimizing stream processing applications via graph partitioning. In *Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 308–327. Springer.

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 239–250. ACM.

Lakshmanan, G. T., Li, Y., and Strom, R. (2008). Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, *12*(6), 50–60.

Leibiusky, J., Eisbruch, G., and Simonassi, D. (2012). *Getting started with Storm*. O'Reilly Media, Inc.

Li, C., Zhang, J., and Luo, Y. (2017). Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of Storm. *Journal of Network and Computer Applications*, *87*, 100–115.

Li, T., Tang, J., and Xu, J. (2015). A predictive scheduling framework for fast and distributed stream data processing. In *Proceedings of 2015 IEEE International Conference on Big Data (Big Data)*, 333–338. IEEE.

Li, T., Tang, J., and Xu, J. (2016). Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, *2*(4), 353–364.

Liu, X. and Buyya, R. (2017). D-Storm: Dynamic resource-efficient scheduling of stream processing applications. In *Proceedings of the 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 485–492. IEEE.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012). Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, *5*(8), 716–727.

Luthra, M., Koldehofe, B., Weisenburger, P., Salvaneschi, G., and Arif, R. (2018). TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of the 12th ACM International Conference on Distributed Event-Based Systems*, 136–147.

Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 49–60. ACM.

Marz, N. and Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

Mayer, R., Koldehofe, B., and Rothermel, K. (2015). Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, *2*(4), 274–286.

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Proceedings of the 2010 International Conference on Data Mining Workshops (ICDMW)*, 170–177. IEEE.

Panigati, E., Schreiber, F. A., and Zaniolo, C. (2015). Data streams and data stream management systems and languages. In *Data Management in Pervasive Systems*, 93–111. Springer.

Peng, B., Hosseini, M., Hong, Z., Farivar, R., and Campbell, R. (2015). R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference*, 149–161. ACM.

Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. (2006). Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 49–49. IEEE.

Querzoni, L. and Rivetti, N. (2017). Data streaming and its application to stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 15–18. ACM.

Rao, B. T. and Reddy, L. (2012). Survey on improved scheduling in Hadoop MapReduce in cloud environments. *arXiv preprint arXiv:1207.0780*.

Rizou, S., Dürr, F., and Rothermel, K. (2010). Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of the 19th International Conference on Computer Communication Networks (ICCCN)*, 1–6. IEEE.

Rychlỳ, M., Škoda, P., and Smrž, P. (2015). Heterogeneity–aware scheduler for stream processing frameworks. *International Journal of Big Data Intelligence*, *2*(2), 70–80.

Saxena, S. and Gupta, S. (2017). *Practical real-time data processing and analytics: Distributed somputing and event processing using Apache Spark, Flink, Storm, and Kafka*. Packt Publishing.

Shukla, A. and Simmhan, Y. (2018). Model-driven scheduling for distributed stream processing systems. *Journal of Parallel and Distributed Computing*, *117*, 98–114.

Smirnov, P., Melnik, M., and Nasonov, D. (2017). Performance-aware scheduling of streaming applications using genetic algorithm. *Procedia Computer Science*, *108*, 2240–2249.

Srivastava, U., Munagala, K., and Widom, J. (2005). Operator placement for in-network stream query processing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 250–258. ACM.

Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM Sigmod Record*, *34*(4), 42–47.

Sullivan, M. and Heybey, A. (1998). A system for managing large databases of network traffic. In *Proceedings of USENIX*.

Sun, D. and Huang, R. (2016). A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access*, 8593–8607.

Sun, D., Yan, H., Gao, S., Liu, X., and Buyya, R. (2018). Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. *The Journal of Supercomputing*, *74*(2), 615–636.

Sun, D., Zhang, G., Yang, S., Zheng, W., Khan, S. U., and Li, K. (2015). Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences*, *319*, 92–112.

Tanaka, M. and Tatebe, O. (2012). Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 65–72. IEEE Computer Society.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, *7*(3), 193–204.

To, Q.-C., Soto, J., and Markl, V. (2018). A survey of state management in big data processing systems. *The International Journal on Very Large Data Bases*, *27*(6), 847–872.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., *et al.* (2014). Storm@ Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, 147–156. ACM.

Wang, L., Xiao, Y., Shao, B., and Wang, H. (2014). How to partition a billion-node graph. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)*, 568–579. IEEE.

Wang, Y., Tari, Z., HoseinyFarahabady, M. R., and Zomaya, A. Y. (2017). QoS-aware resource allocation for stream processing engines using priority channels. In *Proceedings of the 16th International Symposium on Network Computing and Applications (NCA)*, 1–9. IEEE.

White, T. (2012). Hadoop-The definitive guide: Storage and analysis at internet scale.

Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., and Fleischer, L. (2008). SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 306–325. Springer.

Wu, K.-L., Hildrum, K. W., Fan, W., Yu, P. S., Aggarwal, C. C., George, D. A., Gedik, B., Bouillet, E., Gu, X., Luo, G., *et al.* (2007). Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *Proceedings of the 33rd International Conference on Very large data bases*, 1185–1196. VLDB Endowment.

Xhafa, F. and Abraham, A. (2008). Meta-heuristics for grid scheduling problems. In *Metaheuristics for Scheduling in Distributed Computing Environments*, 1–37. Springer.

Xing, Y., Zdonik, S., and Hwang, J.-H. (2005). Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 791–802. IEEE.

Xu, J., Chen, Z., Tang, J., and Su, S. (2014). T-Storm: Traffic-aware online scheduling in Storm. In *Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS)*, 535–544. IEEE.

Xu, L., Peng, B., and Gupta, I. (2016). Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proceedings of 2016 IEEE International Conference on Cloud Engineering (IC2E)*, 22–31. IEEE.

Yoo, D. and Sim, K. M. (2011). A comparative review of job scheduling for MapReduce. In *Proceeding of IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 353–358. IEEE.

Yu, J., Buyya, R., and Ramamohanarao, K. (2008). Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, 173–214. Springer.

Zaharia, M. (2009a). Hadoop fair scheduler design document.

Zaharia, M. (2009b). Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, *9*.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer systems*, 265–278. ACM.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2–2. USENIX Association.

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, 10–10. USENIX Association.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. (2008). Improving MapReduce performance in heterogeneous environments. In *OSDI*, Volume 8, 7.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., *et al.* (2016). Apache Spark: a unified engine for big data processing. *Communications of the ACM*, *59*(11), 56–65.

Zikopoulos, P. and Eaton, C. (2011). *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media.