Stepan Klymonchuk

Optimizing Worker Communication in Serverless Environments with Worker Co-Location

Bachelor's Thesis

Supervised by

Dr. Pedro Antonio García López

Co-supervised by

Aitor Arjona Pérez

Bachelor's Degree in Computer Engineering

Department of Computer Engineering and Mathematics

UNIVERSITAT
ROVIRA i VIRGILI

Tarragona 2024

Dedico esta tese às muitas horas de depuração, às muitas linhas de código, às muitas horas de escrita, às muitas horas de edição, às muitas horas de revisão, às muitas horas de re-revisão, às muitas horas de re-re-revisão, às muitas horas de re-re-re-revisão, às muitas horas de re-re-re-re-revisão, às muitas horas de re-re-re-re-re-revisão, às muitas horas de re-re-re-re-re-re-revisão, às muitas horas de re-re-re-re-re-re-re-revisão, às muitas horas de re-re-re-re-re-re-re-re-revisão, e leitura sobre computação sem servidor na fronteira.

Bicho papão,
sai de cima do telhado,
deixe esse menino
dormir sossegado.

Acknowledgements

Thanks to my supervisor, Dr. Pedro Antonio García López, for his research ideas and guidance throughout the development of this thesis.

I am also grateful to his CloudLab research group at Universitat Rovira i Virgili (URV), for providing support and collaborative enviroment that enriched my research experience. Special thanks to Aitor Arjona, who provided valuable feedback and insights during the development of the *Burst Communication Middleware* (BCM) and his constructive feedback on this work, and to Enrique Molina and Germán Eizaguirre, who were part of my daily life at the lab.

Finally, I would like to thank my family and friends for their continuous support and encouragement during my studies.

Abstract

The rapid evolution of cloud computing has introduced Function-as-a-Service (FaaS), which excels in handling stateless, independent tasks through on-demand, scalable computing resources with a pay-as-you-go model. However, FaaS struggles with burst-parallel jobs requiring simultaneous, coordinated function invocations due to its lack of group awareness, leading to inefficiencies in resource allocation and worker communication. Burst computing is presented as a novel serverless solution designed for burst-parallel jobs, enabling group invocation and collective communication semantics, guaranteed parallelism and worker co-location on top of an existing FaaS platform. Central to this thesis is the design and implementation of the Burst Communication Middleware (BCM), an essential component of the burst computing platform. BCM enhances indirect communication efficiency among workers by leveraging group awareness and exploiting locality for local communications, thereby addressing the limitations of FaaS in managing burst-parallel workloads. It also provides mechanisms for message passing and group collectives, seamlessly benefiting from locality with zero-copy data transfers. This thesis details the design, architecture, and implementation of BCM and evaluates its performance against standard FaaS solutions. Results demonstrate that BCM in the burst computing context significantly reduces communication overheads making it a promising solution for burst-parallel workloads in cloud computing.

Keywords: Middleware, communication, collective, locality, serverless, cloud, burst.

Resumen

La rápida evolución de la computación en la nube ha introducido Function-as-a-Service (FaaS), que destaca en la gestión de tareas independientes y sin estado a través de recursos computacionales escalables bajo demanda con un modelo de pago por uso. Sin embargo, FaaS tiene dificultades con las ráfagas de trabajos paralelos que requieren invocaciones de funciones simultáneas y coordinadas debido a su falta de noción de grupo, lo que conduce a ineficiencias en la asignación de recursos y la comunicación entre workers. Burst computing se presenta como una nueva solución serverless diseñada para ráfagas de trabajos paralelos, que permite la invocación en grupo y la semántica de comunicación colectiva, el paralelismo garantizado y la co-localización de workers sobre una plataforma FaaS existente. Un aspecto central de este trabajo de fin de grado es el diseño y la implementación del Burst Communication Middleware (BCM), un componente esencial de la plataforma de Burst Computing. BCM mejora la eficiencia de la comunicación indirecta entre los workers aprovechando la noción de grupo y explotando la localidad para las comunicaciones locales, abordando así las limitaciones de FaaS en la gestión de cargas de ráfagas de trabajos paralelos. También proporciona mecanismos para el paso de mensajes y operaciones colectivas, beneficiándose de la localidad con transferencias de datos zero-copy. Este trabajo detalla el diseño, la arquitectura y la implementación de BCM y evalúa su rendimiento frente a las soluciones FaaS estándar. Los resultados demuestran que BCM en el contexto de Burst Computing mejora significantemente el rendimiento de la comunicación, por lo que es una solución prometedora para cargas de ráfagas de trabajos paralelos en el ámbito de la computación en la nube.

Palabras clave: Middleware, communication, collective, locality, serverless, cloud, burst.

Resum

La ràpida evolució de la computació al núvol ha introduït Function-as-a-Service (FaaS), que destaca en la gestió de tasques independents i sense estat a través de recursos computacionals escalables sota demanda amb un model de pagament per ús. Tot i això, FaaS té dificultats amb les ràfegues de treballs paral·lels que requereixen invocacions de funcions simultànies i coordinades a causa de la seva falta de noció de grup, cosa que condueix a ineficiències en l'assignació de recursos i la comunicació entre workers. Burst computing es presenta com una nova solució serverless dissenyada per a ràfegues de treballs paral·lels, que permet la invocació en grup i la semàntica de comunicació col·lectiva, el paral·lelisme garantit i la col·localització de workers sobre una plataforma FaaS existent. Un aspecte central d'aquest treball de fi de grau és el disseny i la implementació de Burst Communication Middleware (BCM), un component essencial de la plataforma de Burst Computing. BCM millora l'eficiència de la comunicació indirecta entre els workers aprofitant la noció de grup i explotant la localitat per a les comunicacions locals, i aborda així les limitacions de FaaS en la gestió de càrregues de ràfegues de treballs paral·lels. També proporciona mecanismes per al pas de missatges i operacions col·lectives, beneficiant-se de la localitat amb transferències de dades zero-copy. Aquest treball detalla el disseny, l'arquitectura i la implementació de BCM i n'avalua el rendiment enfront de les solucions FaaS estàndard. Els resultats demostren que BCM en el context de Burst Computing millora significantment el rendiment de la comunicació, per la qual cosa és una solució prometedora per a càrregues de ràfegues de treballs paral·lels en l'àmbit de la computació al núvol.

Paraules clau: Middleware, communication, collective, locality, serverless, cloud, burst.

Research is what I'm doing when I don't know what I'm doing.
Wernher von Braun

Table of contents

List of figures

List of tables

List of Listings

Nomenclature

Acronyms / Abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
BCM	Burst Communication Middleware
COW	Copy-on-Write
DB	Database
EKS	Elastic Kubernetes Service
EMR	Elastic MapReduce
FaaS	Function as a Service
GCP	Google Cloud Platform
HPC	High-Performance Computing
HTTP	HyperText Transfer Protocol
MAD	Median Average Deviation
ML	Machine Learning
MPI	Message Passing Interface
MPSC	Multiple Producer, Single Consumer
OS	Operating System
S3	Simple Storage Service
SGD	Stochastic Gradient Descent
SPMD	Single Program Multiple Data

TCP Transmission Control Protocol

VM Virtual Machine

Chapter 1

Introduction

In recent years, the landscape of cloud computing has seen a significant transformation with the introduction of various paradigms designed to optimize resource utilization, scalability, and cost efficiency. The advent of Function-as-a-Service (FaaS) has been particularly noteworthy, promising rapid, on-demand scaling with a pay-as-you-go billing model that operates at fine granularity. Due to its flexibility and resource efficiency, the adoption of FaaS has been propelled across various domains, from data analytics to video encoding and even code compilation. However, despite FaaS's excellent performance in handling stateless, independent tasks, it falls short when dealing with burst-parallel jobs, which require multiple, simultaneous function invocations.

This thesis is part of a larger research effort, encapsulated in the to-be-published paper *"Burst Computing: Quick, Sudden, Massively Parallel Processing on Serverless Resources"* [7], written by various authors from the CloudLab research group at Universitat Rovira i Virgili (URV). The central theme of this work revolves around a novel serverless solution referred to as *burst computing*.

1.1 *Burst computing*: a novel approach

Burst computing emerges as a new serverless paradigm tailored to address the limitations of FaaS for burst-parallel jobs. This cloud computing model is designed to handle quick, sudden, massively parallel workloads — referred to as *bursts*. Unlike FaaS, *burst computing* elevates the isolation boundary from individual function invocations to entire jobs using a group invocation primitive and enabling the simultaneous launch of large groups of workers. This approach optimizes resource allocation through worker packing, co-locating workers in fewer environments to accelerate initialization and exploit locality.

The simultaneity of worker launches allows for synchronous communication, using message passing and group collectives. Worker packing favours exploiting locality, using shared memory to communicate workers co-located in the same pack, while maintaining remote communication to reach remote workers. The seamless combination of local and

remote communication greatly reduces the overhead of collective operations. Doing so efficiently is the focus of this thesis.

1.2 Main contribution: *Burst Communication Middleware* (BCM)

The primary contribution of this thesis is the design and implementation of the *Burst Communication Middleware* (BCM), an essential component of the *burst computing* platform. BCM enables seamless and efficient communication among workers by leveraging **group-awareness** and **locality** established through *burst computing*. It provides mechanisms for message passing and group collectives, significantly reducing the overheads associated with remote communication in FaaS.

The objective of this thesis is to implement a communication middleware in Rust for the *burst computing* platform. Rust is chosen for its performance, safety, and concurrency capabilities, making it ideal for developing high-performance applications. The goals for the BCM include:

- Develop a message-oriented MPI-like API that abstracts both local and remote communication making it transparent to the user.
- Ensure ultra-fast local communication using zero-copy techniques.
- Create an extensible architecture that supports various communication backends such as Redis, RabbitMQ, and others.

However, before delving into the details of BCM, it is essential to understand the *burst computing* platform and the challenges it addresses.

1.3 Structure of the thesis

To provide a comprehensive understanding of *burst computing* and the role of BCM, the thesis is structured as follows:

- **Chapter 2: Background** - This chapter delves into the concept of *burst computing*, explaining its motivations and the challenges it addresses. It provides the necessary context to understand the importance of BCM within the *burst computing* framework.
- **Chapter 3: Burst Communication Middleware design and implementation** - This chapter details the design principles, architecture, and implementation of BCM. It discusses how BCM integrates with the *burst computing* platform to facilitate efficient worker communication.

- **Chapter 4: Evaluation** - This chapter presents an evaluation of the *burst computing* platform and BCM. It includes performance metrics and comparative analyses with standard FaaS solutions, demonstrating the improvements in job invocation latency and communication efficiency.
- **Chapter 5: Related work** - This chapter compares the contributions of this thesis with existing research related to communication middleware. It highlights the unique features of BCM and its differentiation from other solutions in the cloud computing domain.
- **Chapter 6: Conclusions and future work** - The final chapter summarizes the findings of the thesis, highlighting the contributions of BCM and *burst computing* to the field of cloud computing. It also suggests potential future research directions to further enhance the capabilities of burst computing.

Chapter 2

Background

In this chapter, the concept of *burst computing* is introduced as a novel cloud computing model designed to address the limitations of Function-as-a-Service (FaaS) for burst-parallel jobs. First, the motivation for *burst computing* is discussed, followed by some background on FaaS and its limitations. Then, the chapter details the differences between *burst computing* and FaaS. Finally, the chapter details the architecture of the *burst computing* platform and highlights the role of the *burst communication middleware* (BCM) within this framework.

This background information is essential for understanding the context and motivation behind the development of the BCM. The definitions and concepts presented in this chapter are based on the not yet published paper on *burst computing* [7]. It is important to note that the term *burst computing* is used throughout this thesis to refer to the proposed cloud computing model.

2.1 Motivation for *burst computing*

2.1.1 What is a *burst*?

For certain applications, the ability to immediately tap into virtually unlimited resources is a game-changer. Consider, for example, a scientist engaged in an interactive workflow, conducting various analyses on massive datasets and frequently altering parameters, which significantly changes the computational requirements. Similarly, real-time applications that process data streams or video feeds must handle fluctuating volumes and varying analytic complexities. These scenarios are often referred to as ***Extreme Data*** and are characterized by their unpredictable nature and the need for rapid, on-demand scaling.

We denote these types of applications as ***bursts*** due to their sudden, intense computational demands. *Bursts* are **massively parallel processing** (MPP) jobs that emerge unexpectedly, requiring the processing of large, variable amounts of data within very short timeframes, typically one to two minutes or less.

2.1.2 Clusters are not *bursty*

Running a data processing platform is a challenging task: it must a) handle varying amounts of data, which arrive in unpredictable patterns, while simultaneously b) providing real-time or interactive responses and c) ensuring cost-effectiveness. Traditional cluster technologies like Spark or Dask, which are widely used for big data processing, struggle in these scenarios because they rely on a ***cluster-centric*** design. This design focuses on maximizing the utilization of the cluster resources, but it leads to significant issues with resource misprovisioning [35].

One of the primary limitations of the before-mentioned technologies is their inability to **quickly adjust to sudden changes in demand**: this concept is known as *elasticity*. The existing elasticity mechanisms, whose purpose is to scale resources up or down based on workload, in these technologies are too slow.

- On the one hand, the lack of responsiveness means that clusters often remain **under-provisioned** during peak demand, failing to meet the real-time processing requirements of applications.
- On the other hand, to avoid this problem, clusters might be **over-provisioned** for peak demand, allocating more resources than necessary for most of the time. As a result, the cluster becomes very expensive to operate, due to additional resources remaining mostly idle during periods of normal or low demand [46, 29, 31].

Nevertheless, existing data processing solutions like Spark, Dask, Flink and Ray struggle to support *bursts* effectively as their response times to workload fluctuations are too slow. This inefficiency stems from their **cluster-centric** design, which manages capacity at coarse granularity level, such as VMs. This results in high startup times and complex resource management.

Table 2.1 Start-up time of different cluster technologies compared to a FaaS service. AWS EMR Spark and GCP Dataproc use m5 and E2-standard VM families, respectively. Dask and Ray are deployed on managed m6i family EC2 VMs.

Technology	Total vCPUs	Nodes	Start-up time
EMR Spark	96	6	296 s
		24	431 s
Dataproc	96	6	95 s
		24	113 s
Dask	128	8	184 s
		64	253 s
Ray	128	8	187 s
		64	229 s
AWS λ 10 GiB	6000	1000	6 s

For example, table 2.1 shows that the startup time for these technologies is prohibitively long for critical real-time applications. Even cloud-managed versions of these platforms, such as "serverless Spark", require several minutes to initiate just a few nodes [42], making them unsuitable for *burst* workloads.

2.1.3 FaaS: panacea or just a buzzword?

FaaS has emerged as a promising solution to the resource allocation issues faced by traditional cluster technologies. FaaS is a cloud computing model that allows users to run code in response to events without managing the underlying infrastructure. In the FaaS model, developers write functions, which are small stateless units of code designed to perform a specific task [25, 10]. It offers some unique features that make it well-suited for tackling the resource provisioning problem. These include:

- **Rapid, on-demand scalability:** Functions automatically scale up or down based on the number of incoming requests. Additionally, FaaS platforms can start new functions in milliseconds, making them ideal for applications with sudden, unpredictable workloads [5].
- **No-ops:** FaaS infrastructure is managed by the cloud provider, freeing developers from the burden of managing servers. *No-ops* means no need for manual intervention to scale the infrastructure, as it is done automatically by the platform [25].
- **Fine-grained billing:** FaaS platforms charge users based on the actual resources consumed by their functions. This pay-as-you-go billing model allows users to pay only for the resources they use, making FaaS cost-effective for applications with varying workloads. Also, the billing granularity is very fine, often measured in megabytes per millisecond (see AWS Lambda pricing [4, 3]).

In table 2.1, the startup time of cluster technologies is shown. In stark contrast, FaaS services offer a much faster startup time. As illustrated in fig. 2.1, AWS Lambda can deploy 100 functions in under 4 s or 1000 functions in 6 s during a cold start. This rapid deployment capability makes FaaS services far more suitable for handling *bursts*. Thus, serverless computing shifts the paradigm from a **cluster-centric**, which relies on available resources, to a **job-centric** approach that precisely utilizes the necessary resources from a theoretically infinite cloud pool.

FaaS *burstability*

The capability of FaaS to rapidly scale is referred to as *burstability*, and it is a key feature that distinguishes FaaS from traditional cluster technologies (see table 2.1). The concept of ***burstability*** means that FaaS can handle sudden increases in workload by quickly provisioning additional resources.

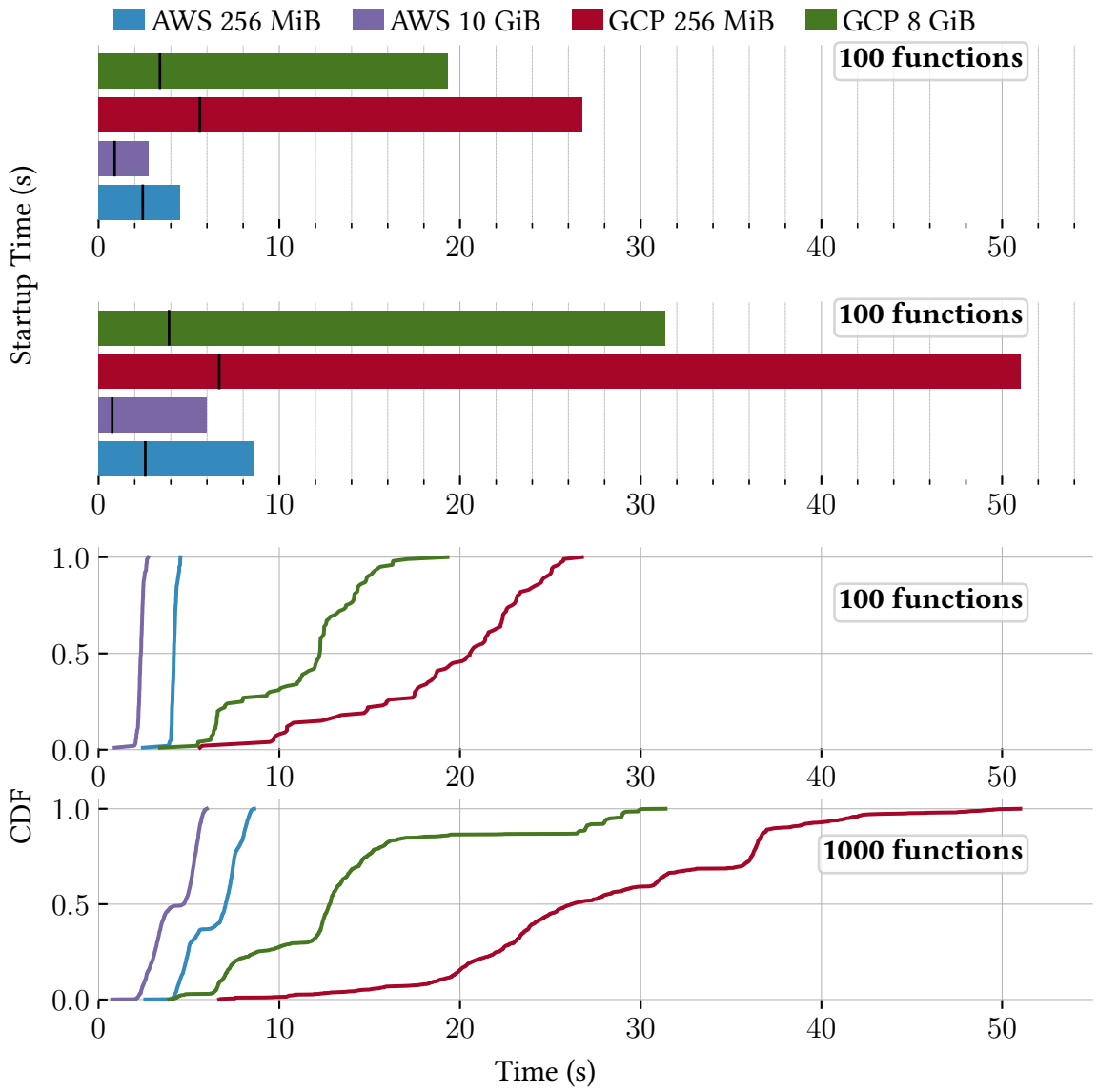


Fig. 2.1 CDF of FaaS function start-up time (cold start) in AWS Lambda and GCP Cloud Functions for 100 and 1000 function invocations on two memory configurations. The vertical split denotes the latency of the first invocation.

Multiple research projects have demonstrated the burstability of FaaS by running thousands of short-lived functions in parallel for tasks that require intensive data processing or computational power. For instance, studies have shown the effectiveness of FaaS for workloads such as a) **data analytics**, b) **video encoding**, c) **code compilation**, d) **sorting**, and other massive data processing tasks [28, 17, 41, 18, 32, 1, 49, 9].

Concepts similar to *burstability* have been explored by other researchers in the past such as Fouladi et al. [17] with **ExCamera** who described a "burstable supercomputer-on-demand" and a "burst-parallel swarm" of cloud functions working simultaneously on the same task of encoding a video. Another example is **PyWren** [28], a framework that allows users to run parallel workloads on FaaS platforms. PyWren has been used to run large-scale data analytics tasks, demonstrating the potential of FaaS for running embarrassingly parallel jobs. The

idea has evolved with the introduction of the term ***flash burst*** by Li, Park, and Ousterhout [32]. Flash bursts are applications that leverage a large number of servers for extremely short periods of time, down to a millisecond. They are suitable for applications that require high throughput and low latency, such as sorting, querying, and other data-intensive tasks. Furthermore, the concept of ***serverless clusters*** [34] has been proposed as a way to extend the *burstability* of FaaS to support more complex workloads. They aim to provide a more flexible and scalable alternative to traditional clusters by leveraging the *burstability* of FaaS platforms.

2.1.4 FaaS limitations for *bursty* workloads

Despite the numerous benefits the FaaS model provides, there is a consensus among researchers that the current FaaS model is too limited for massively parallel data processing programs (MPP). The primary appeal of serverless computing lies in its ability to provide **resource *burstiness*** [34].

For example, Jonas et al. [27] highlight the following limitations of FaaS: a) **statelessness** of functions, b) lack of **coordination** between functions, c) no support for **communication primitives**, and d) lack of **predictability in performance**. FaaS environments struggle with tasks that require **inter-function communication**, **shared state**, or **coordination** between functions as pointed out by Hellerstein et al. [23] and Barcelona-Pons and García-López [5].

Müller et al. [34] argue in their article that these limitations are **by design**, as FaaS was modelled for a fundamentally different use case: serving an unlimited stream of independent requests, whose volume may change rapidly and unpredictably and thus needs fast autoscaling of resources. In contrast, a set of closely-coupled workers running at the same time is a better fit for MPP programs.

Serverless functions lack several must-have features for running parallel workloads, namely a) **direct communication** between functions, b) **batch invocation** API, c) the ability to know **which functions are currently running**, and d) the ability to know how many **concurrent invocations** are active at any given time. In this thesis, the aforementioned limitations are summarized under the terms a) ***group awareness*** and b) ***locality exploitation***.

In a typical FaaS environment, each function invocation is treated as an **independent request**, and functions are not aware of each other. Users must make multiple independent service calls in order to spawn a multi-function job. Moreover, functions are executed in strongly isolated environments: they do not share memory or state with other functions. Such fine-grained isolation becomes detrimental for tasks that require collaboration between functions.

As a result, running parallel workloads on FaaS platforms can be challenging due to the following **friction points**:

F1 **Worker isolation or lack of *group invocation***

F2 **Job fragmentation with complex coordination**

F3 **Huge data movement**

To illustrate these challenges, consider executing a parallel job with six workers on a FaaS platform, as depicted in fig. 2.2. These jobs could either be **embarrassingly parallel** (**stateless**) like hyperparameter tuning or require **synchronization** and **state sharing** among workers (**stateful**) such as TeraSort or PageRank. The details of these algorithms are described in section 2.3.2 and section 4.6.

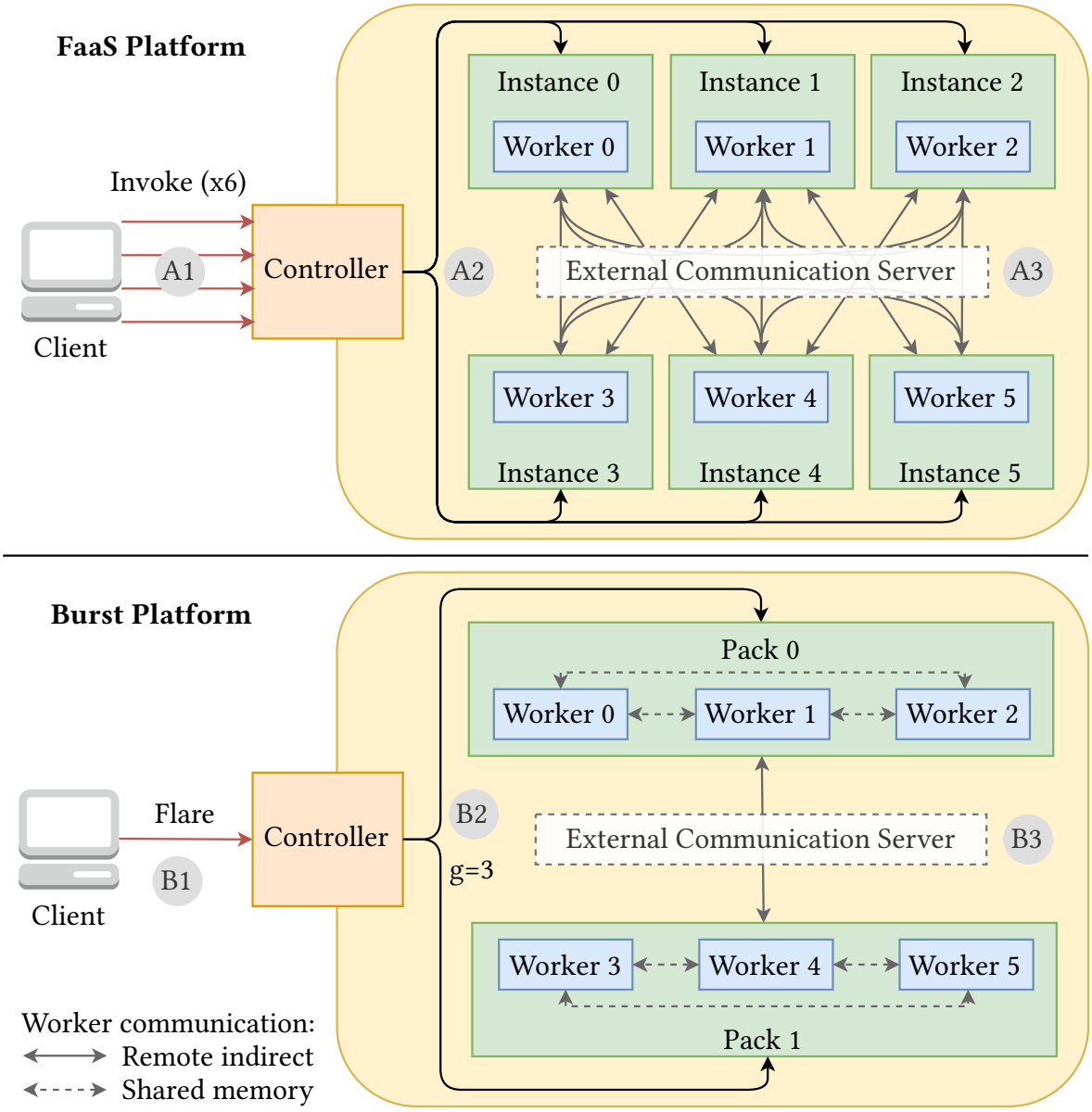


Fig. 2.2 Running a data processing job of 6 workers in FaaS and *burst computing* with granularity 3.

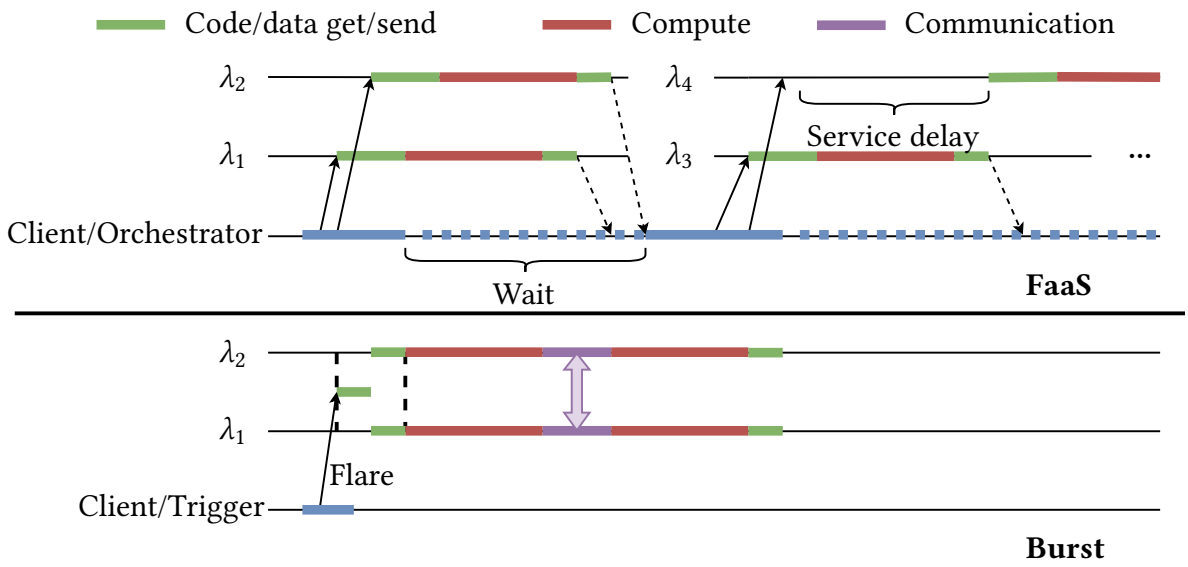


Fig. 2.3 Timeline of a parallel job in FaaS and *burst computing* approaches. FaaS thwarts parallel applications with job fragmentation and large data movements. *Burst computing* improves execution with *group awareness* and *locality* mechanisms.

F1 The first friction point stems from the **multi-tenant isolation** at the level of **individual function** invocations. In FaaS environments functions are spawned **independently**, one by one, necessitating multiple HTTP requests to initiate six workers A1, which adds **latency** and hinders **parallelism**. For instance, as shown in fig. 2.1, the last of 1000 function invocation may start up to 6 s after the first one. Moreover, the platform is unaware that these workers are part of a **coordinated task**, thereby failing to guarantee their **parallelism**. This results in skewed execution times, redundant loading of environments, due to the lack of ***group awareness***, and **memory duplication** [36, 45].

F2 The second friction point arises when workers need to **coordinate** or **share state**. In scenarios like TeraSort, which includes a data shuffle stage, or PageRank, which iteratively aggregates vectors, workers must **communicate** and **synchronize**. The lack of **guaranteed simultaneity** means workers may not exist concurrently, necessitating **job fragmentation** into multiple stages, **external storage** for intermediate data, and an **active orchestration process**, as depicted in fig. 2.3. Iterative algorithms, such as PageRank or k-means clustering, which require **repeated data aggregation**, become unfeasible under the FaaS model [8].

F3 As regards the third friction point, isolated workers require **numerous remote connections** for communication patterns like data shuffling or aggregation A3, which results in significant data transfers. As functions cannot directly communicate, they must resort to indirect methods, further increasing **latency** and **bandwidth overhead**, as shown by Lu et al. [33] and Copik et al. [11].

2.2 Introducing *burst computing*

To address these limitations, the concept of *burst computing* is introduced: an innovative cloud computing model designed to handle quick, sudden, and massively parallel workloads, referred to as ***bursts***.

2.2.1 *Burst computing*

The core feature of *burst computing* is a *group invocation primitive* (`flare`), which manages the entire job as a single unit and the *burst communication middleware* (BCM), which enables efficient communication among workers. This primitive optimizes resource allocation, guarantees worker **parallelism**, and enables worker ***packing***, co-locating multiple workers within the same environment. This co-location not only speeds up worker start-up latency but also creates worker **locality**, enhancing code and data loading efficiency. Additionally, it allows for **worker-to-worker communication** patterns, such as broadcast and all-to-all, by leveraging shared memory channels with zero-copy mechanisms.

Burst computing is versatile and applicable to a wide range of tasks that are currently challenging or impractical to execute using FaaS. These tasks include data analytics and machine learning workloads, such as TeraSort, TPC-DS, and k-means clustering, as well as stream processing applications like video encoding. Other compute-intensive workloads, such as code compilation or simulations, also benefit from this model. *Bursts* can be either *stateless*, like grid search or Monte Carlo simulations, or *stateful*, involving table joins or aggregations.

2.2.2 Comparing FaaS and *burst computing*

Unlike traditional cluster computing, which requires minutes for elasticity adjustments, *burst computing* offers faster **elasticity**, reducing adjustment times to milliseconds. Moreover, it surpasses current FaaS solutions, which are not designed for parallel computing [5]. As stated before, FaaS systems are limited by their *statelessness* and lack of *direct communication* capabilities [17, 34], and they do not provide a *group invocation* API that can guarantee worker **parallelism** or optimize execution by exploiting **data locality** and **group coordination**.

The *burst computing* approach addresses the limitations of traditional FaaS (friction points explained in section 2.1.4) by incorporating two key principles that extend the capabilities of FaaS platforms:

1. ***Group awareness***
2. ***Locality exploitation***

In terms of implementation, these principles directly translate into the 1) **group invocation primitive** (`flare`), which manages the entire job as a single unit, and 2) **burst**

communication middleware (BCM), which enables efficient communication among workers during the execution of a *burst*.

Figure 2.2 illustrates these distinctions by comparing the execution of a parallel job with six workers in FaaS and *burst computing*, and fig. 2.3 provides a timeline of the job execution in both approaches.

The *burst computing* model is designed to address the limitations of traditional FaaS platforms. Three key friction points are resolved as follows:

F1 Given that a job typically belongs to a single tenant, it is logical to elevate the isolation level to the entire job and treat all associated workers as a **single entity**. To facilitate this, *burst computing* introduces the `flare` primitive **B1**: a **group invocation** primitive to launch massive process groups with guaranteed **parallelism** and **co-location** of workers. `Flares` imbue the platform with ***group awareness***, which is essential for the process of worker *packing* **B2**.

Consider the sample job in fig. 2.2 with six workers, referred to as a *burst* size of six. By setting a *pack granularity* of three ($g = 3$), the platform would need to spawn only two *packs*, each containing three workers. This *packing* approach establishes worker **locality**, which optimizes environment generation by reducing the number of container creations.

F2 **Guaranteed parallelism** and the provision of **job context** enable **synchronous communication**. This context can include information like *burst* size or worker IDs and is explained in more detail in section 2.3.2. Worker parallelism allows previously unfeasible **communication patterns**, such as worker-to-worker message passing and collective operations, simplifying job orchestration.

F3 **Burst communication middleware** (BCM), which enables efficient communication among workers, addresses the third friction point **F3**. The communication **B3** can transparently **exploit locality** by using shared memory for intra-pack communication, minimizing remote data transfers, latency and bandwidth overhead.

2.2.3 *Burst* architecture

In this section, the architecture of the *burst computing* platform is presented.

Components overview

It is essential to understand the platform's architecture to appreciate the need for the BCM and its role in enabling efficient communication among workers during the execution of a *burst*. Figure 2.4 provides a comprehensive overview of the key components of the platform and how they interact with each other.

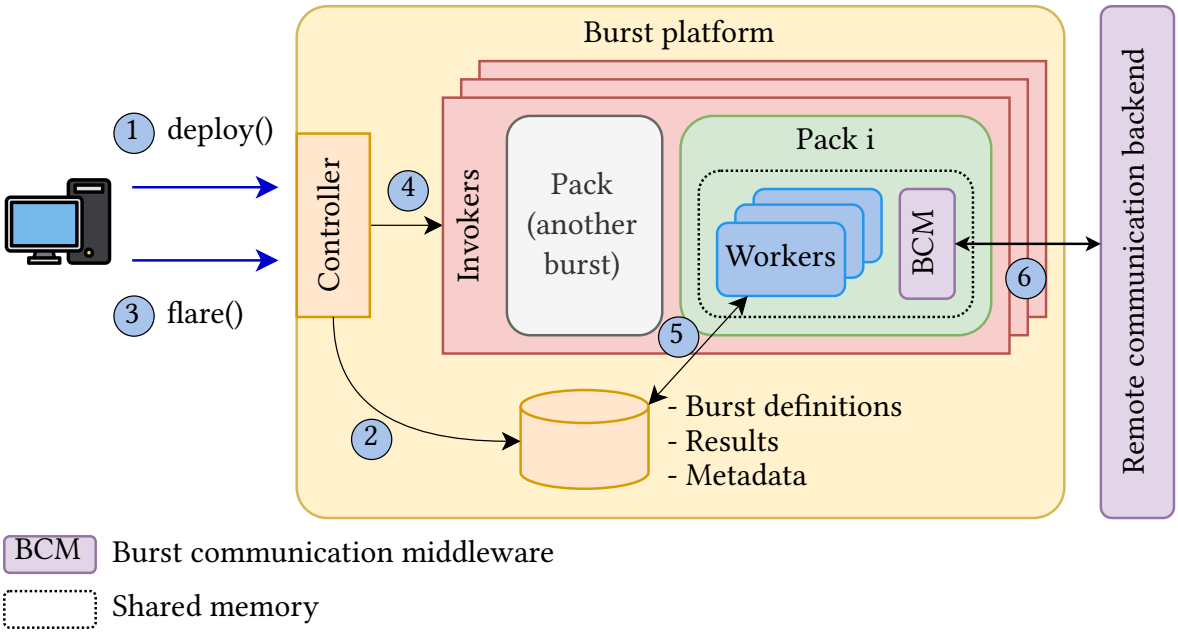


Fig. 2.4 *Burst computing* platform architecture overview.

The *burst computing* platform is built upon the architecture of a *Function-as-a-Service* (FaaS) platform, incorporating additional features to support **group invocation**, **worker packing**, and **burst communication**. The platform utilizes Apache OpenWhisk as its foundation, sharing several of its core components, which include the **controller**, **database**, **invokers**, and **packs**.

1. **Controller**: The controller is the central hub for user interaction with the platform. It handles incoming HTTP requests for deploying and invoking *bursts*, managing system resources, and overseeing the packing of workers into groups. This component ensures that the platform efficiently allocates resources based on the availability of invokers and the current system load, and coordinates the overall execution of *bursts*.

2. **Database**: The database component is responsible for storing all necessary data related to *burst* definitions, configurations, results, and execution metadata. It acts as a persistent storage layer, ensuring that the platform can retrieve and manage *burst*-related information as needed.

3. **Invokers**: The invokers are the compute resources of the platform. They are a collection of machines equipped to handle the execution of *burst* packs. Each invoker can host multiple packs from different *bursts* and users concurrently (i.e., multi-tenancy), providing the necessary isolation between them.

4. **Packs**: Packs are executed within containers that offer a custom runtime environment. This environment is specifically designed to manage and isolate a set of workers, ensuring efficient execution of *burst* tasks. The custom runtime, implemented in Rust, optimizes performance and resource utilization within the containers.

Life cycle of a *burst*

Figure 2.4 also illustrates the life cycle of interaction with the *burst computing* platform, detailing the process of deploying and invoking a *burst*. The life cycle can be broken down into server key stages:

1. **Deployment**: The user sends a `deploy` HTTP request to the controller, containing the *burst* definition and configuration.
2. **Register *burst* definition**: The controller registers the new *burst* definition in the platform's database.
3. **Invocation**: When the user wants to execute the *burst*, they send a `flare` HTTP request to the platform with specific parameters to be passed to the *burst* workers (payload), triggering the execution.
4. **Worker allocation and pack formation**: The controller evaluates the current state of the available invoker machines and decides on worker allocation. The invokers spawn the required runtime environments (*packs*) tailored to accommodate the necessary number of workers.
5. **Loading code and payload**: The invokers direct the packs to load the appropriate burst definition and execution parameters from the database. Each *pack* spawns its workers which execute the user-defined function (`work` in table 2.2) in parallel.
6. **Worker execution and communication**: To facilitate efficient communication and coordination, the workers utilize the BCM, which transparently manages communication channels, using shared memory for intra-pack communication and remote backends for inter-pack communication, as needed.

Workers may need to interact with **external storage systems**, such as object storage, for reading and writing data or storing some results back in the platform's database. Users can later retrieve these results by sending another HTTP request to the platform. The platform's database, however, is not meant to be a permanent storage solution.

Implementation

The prototype *burst computing* platform is implemented on top of Apache OpenWhisk (version 1.0.0) [19, 21], a widely used open-source Function-as-a-Service (FaaS) platform. Apache OpenWhisk was selected due to its robust, production-tested infrastructure. The modifications required for the burst computing platform encompass approximately 2K SLOC, which are publicly accessible on GitLab.[1] These changes span the main components of the platform, including the **controller**, the **invoker**, and the **runtime environment**:

[1] https://gitlab.com/burstcomputing/openwhisk

- **Controller** The controller has been enhanced to support two new HTTP endpoints: `deploy` and `flare`. These endpoints facilitate the deployment and the invocation of *bursts*, respectively, as detailed in section 2.2.3. The controller now implements the logic for handling those requests according to the *packing* strategy (see section 2.3.1). It is also configurable in terms of granularity and calculates the number and size of the packs based on the *burst* size and the available resources on the invokers.

- **Invokers** The invokers feature a new monitoring logic that reports their load to the controller based on CPU usage instead of RAM. Each worker is allocated 1 vCPU, given that *bursts* typically involve compute-intensive tasks, and parallelism within a worker is not considered (but can be adjusted) The overall parallelism of a *burst* is determined by the *burst* size (i.e., the number of workers). Invokers can create and manage packs: spawning Docker containers of the appropriate size and provide the runtime environment with the necessary details to execute (e.g., number of workers, worker IDs, and context)

- **Runtime environment** The runtime environment has been adapted from the official OpenWhisk Rust environment [20], though support for other runtimes is feasible. It spawns multiple workers within a single runtime instance: it creates one OS thread per worker, ensuring parallel execution. The runtime environment also integrates the BCM, facilitating efficient inter-worker communication.

2.3 BCM in the *burst* application

2.3.1 Worker locality

The *burst* platform allocates workers into *packs* to exploit **locality**. Each *pack* runs multiple workers in a single environments or container, with the number of workers per *pack* defined as the *burst*'s ***granularity***. Higher *granularity* results in fewer environments (*packs*), directly reducing allocation and initialization time by loading dependencies once per *pack*. This minimizes memory duplication [36] and speeds up data transfer through parallel downloads.

Three strategies for *packing* workers in *burst computing* include:

1. **Heterogeneous packing**: Maximizes **locality** but can lead to resource **fragmentation** by placing workers in the largest possible containers

2. **Homogeneous packing**: Uses **fixed-size** containers, simplifying scheduling but restricting worker locality.

3. **Mixed packing**: Combines fixed-size packs that can **merge** into a single container on the same machine, balancing locality and management ease.

Table 2.2 Burst computing abstractions and API.

Interface	Functions
Burst Service	**deploy**(*defName*, *package*, *config*) upload and deploy a burst definition **flare**(*defName*, [*inputParams*]) invokes a burst
Burst Function	abstract **work**(*inputParams*, *burstContext*) function to run on each worker
Communication Primitives	**send**(*data*, *dest*) → none **recv**(*source*) → data **broadcast**(*data*, *root*) → data **allToAll**(*[data]*) → [data] **reduce**(*data*, *f*(*data*, *data*) → *data*) → data

To leverage the benefits of *locality* and *group awareness* effectively, *burst* applications are designed to be **elastically distributed and collaborative**. Applications are aware of being distributed and can transparently handle any worker multiplicity. The **guaranteed parallelism** of workers within a *pack* allows for **synchronous coordination**, enabling common communication patterns that simplify job orchestration and reduce remote data transfers by exploiting worker **locality**.

To this extent, the **burst communication middleware** (BCM) is developed and presented in this thesis as the main contribution. The middleware provides a well-known abstractions such as **send**/**receive**, **broadcast**, and **all-to-all**, that strongly benefit from worker **locality**. The BCM is designed to be **transparent** to the user: messages between workers in the same pack use zero-copy memory sharing, while inter-pack messages are transferred remotely and optimized. It also supports various remote communication backends in an extensible manner.

2.3.2 How to use the BCM?

Users interact with a simple interface to define and schedule *bursts* using **resource-agnostic** code, similar to FaaS services. The abstractions for deploying and invoking *bursts*, the worker function definition and the BCM API are summarized in table 2.2.

Bursts are coded as a single `work` function, which is executed by each worker in the *burst*. This function is **elastic**, meaning it should function correctly regardless of the *burst* size and without requiring code modifications. It must also be designed to be agnostic of the underlying infrastructure (e.g., packing strategy or work distribution) to execute seamlessly. Each worker receives a `burstContext` object as an argument. It includes key attributes such as **worker ID** and ***burst* size** and gives access to the **BCM API** for communication.

Real-world example *burst* application

Listing 1 Simplified source code of the PageRang *work* function for *burst computing*, written in Rust. The access to *burst context* to obtain the worker ID or communicate is highlighted in red.

```rust
fn work(params: Input, burst: &BurstContext) -> Output {
    let num_nodes = params.num_nodes;
    let mut page_ranks = vec![1.0 / num_nodes; num_nodes];
    let mut sum = vec![0.0; num_nodes];
    let adjacency_matrix = get_adjacency_matrix(&params);
    while err < ERROR_THRESHOLD {
        page_ranks = burst.broadcast(page_ranks, ROOT_WORKER)};
        for (node, links) in graph {
            for link in links {
                sum[*link] += page_ranks[*node] / out_links(*node);
            }
        }
        let reduced_ranks = burst.reduce(sum, |vec1, vec2| {
            vec1.zip(vec2).map(|(a, b)| a + b).collect()
        });
        if burst.worker_id == ROOT_WORKER {
            err = calculate_error(&page_ranks, &reduced_ranks);
            page_ranks = reduced_ranks;
        }
        err = burst.broadcast(err, ROOT_WORKER);
        reset_sums(&mut sum);
    }
    Output { page_ranks }
}
```

Listing 1 presents a simplified Rust implementation of the PageRank algorithm and how it interacts with the BCM. PageRank is an iterative algorithm used to rank web pages based on their links. Workers process portions of the adjacency graph, compute global ranks, and use collective operations like *broadcast* and *reduce* via the `BurstContext` object. The algorithm runs iteratively until convergence or a set iteration limit, with workers performing different tasks based on their unique IDs, similar to the **MPI** model.

Chapter 3

Burst Communication Middleware design and implementation

In this chapter, the design and implementation details of the *burst communication middleware* (BCM) are presented.

3.1 Burst Communication Middleware

The *burst communication middleware* (BCM) is an essential part of the platform, facilitating efficient communication between workers. It has been designed with specific requirements in mind to ensure **efficient**, **reliable**, and **scalable** communication among workers executing a *burst*.

3.1.1 Requirements and design goals

The BCM has been designed to meet specific key requirements and design goals explained in detail in the following sections.

One of the key features of the BCM is its **locality-awareness**, which optimizes communication performance by leveraging shared memory for intra-pack communication. However, the BCM is designed to be transparent to the user: the user code must be agnostic to the underlying communication mechanisms (shared memory, remote backends, etc.) and the distribution of workers across packs. The seamless execution of user-defined functions across multiple workers is a core requirement of the platform, and the BCM plays a crucial role in enabling this functionality. This also means that users must not rely on specific workers being co-located in the same pack and, therefore, the communication between them being local.

Intra-pack communication

Intra-pack communication refers to communication between workers located in the same pack (i.e., the same machine). Provided that workers within the same pack are **co-located**

in the same container, they can communicate efficiently using **shared memory**. In order to further optimize the communication process, the BCM will support the following features:

- **Zero-copy data transfer**: The BCM must enable zero-copy data transfer between workers within the same pack. This ensures high efficiency by avoiding unnecessary data copying.
- **Shared memory utilization**: The BCM must leverage shared memory for intra-pack communication.
- **Thread-safe communication**: The BCM must ensure that communication and access to shared memory are thread-safe.

The before-mentioned requirements focus on exploiting the **locality** of workers to optimize communication performance.

Inter-pack communication

Inter-pack communication refers to communication between workers located in different packs (i.e., different machines). This type of communication is more challenging due to the distributed nature of workers across different invokers and requires dealing with network latency and bandwidth constraints. The BCM must address the following requirements to ensure efficient inter-pack communication:

Multiple backends Following the FaaS model, *burst computing* does not allow direct addressing for communicating workers. For example, a worker is not allowed to bind and listen to a TCP/IP socket. This decision is by design. Allowing direct communication would require (i) an overlay network with isolation between tenants and (ii) a *rendezvous* service for workers to find each other, which would complicate the system and affect its performance. Instead, indirect communication is needed: workers access an external managed service to synchronize and to send and receive messages. In this sense, the BCM must support multiple remote message passing or DB backends, such as Redis, RabbitMQ, and S3, to accommodate different communication requirements.

Connection pooling The BCM must maintain a shared connection pool for each pack to optimize bandwidth utilization, reduce latency, and manage resources efficiently. By maintaining a pool of reusable connections, the system avoids the overhead of repeatedly establishing and tearing down connections, thereby optimizing latency (e.g. TCP handshake). Connection pooling also facilitates numerous messages to be sent and received concurrently, maximizing the use of available bandwidth. Besides, the connection pool limits the number of connections that can be established, preventing workers from creating an excessive number of connections on their behalf. Another important aspect of the connection pooling mechanism is that it is shared between all the workers in the pack, allowing one worker to use multiple connections concurrently, further increasing throughput.

Chunked message transfer The BCM must support chunked message transfer for large data payloads to optimize network utilization and reduce latency. By dividing large messages into smaller chunks, the middleware can transmit these smaller portions concurrently. This way, the recipient does not have to wait for the entire message to be transmitted before processing it. Efficiently handling aspects related to chunking as memory and chunk reassembly is an essential requirement for the BCM.

Backend interface for remote communication

The BCM must provide a **unified interface** for interacting with remote backends, abstracting away the complexities of each backend's specific commands and data structures. These requirements are related to the previous ones, as they focus on remote communication optimization.

- **Backend extensibility**: The BCM must allow for easy integration of additional remote backends.
- **Message type differentiation**: The BCM must differentiate between **direct** (one-to-one) and **broadcast** (one-to-many) messages, optimizing retrieval processes accordingly.
- **Efficient message distribution**: For broadcast messages, the BCM must use the appropriate backend commands to ensure efficient message distribution.
- **Optimized data structures**: The BCM must utilize the most suitable data structures and commands for each backend to optimize message storage and retrieval.
- **Polling mechanisms**: The BCM must implement polling mechanisms for backends that are not message-oriented and lack blocking commands to wait for messages.

At-least-once delivery semantics

In distributed systems that use message passing for communication, such as the *burst computing* platform, **message delivery reliability** is crucial to ensure the correct execution of parallel tasks. These requirements focus on the inter-pack communication aspect of the BCM, as the intra-pack communication is inherently reliable due to shared memory usage.

There are three main types of message delivery semantics: **at-most-once**, **at-least-once**, and **exactly-once**.

- **At-most-once delivery**: In this delivery model, each message is delivered *at most once* or not at all. In other words, there is a possibility that some messages may be lost and no retransmission is attempted. In the context of the BCM, at-most-once delivery is not acceptable: losing messages could lead to incorrect results or program failures.

- **At-least-once delivery**: This model makes sure that every message is delivered *at least once*. This guarantees that no message is lost, as multiple attempts are made to deliver each message until at least one delivery is successful. However, this model may result in duplicate messages being delivered. Regardless, at-least-once delivery is the most suitable model for the BCM.

- **Exactly-once delivery**: This is the strongest delivery semantics, ensuring that each message is delivered *exactly once*. Messages cannot be lost or duplicated. Achieving exactly-once delivery is challenging and often requires more complex mechanisms.

The BCM must guarantee **at-least-once** delivery semantics to assure that no messages are lost during communication. This requirement is essential for reliable message passing and correct execution of jobs in the platform. As duplicates may occur in the at-least-once model, the BCM must implement mechanisms to detect and discard duplicate messages to prevent processing errors.

What is more, the BCM must ensure that messages are delivered in the **correct order** to the workers, as they may arrive out of order due to concurrency. Given that chunking is employed for large messages, it becomes even more challenging to tackle the issues related to message ordering and duplicates. The BCM must reconstruct the original message from the chunks in the correct order, even if they arrive out of order. This requirement is crucial for maintaining the integrity of the data and ensuring that the workers process the messages correctly.

In order to meet these requirements, the BCM must implement a **message tracking system** to manage message delivery and ordering effectively. This system must track the messages sent and received by each worker, ensuring that no messages are lost or processed out of order.

3.1.2 Architecture of the BCM

The BCM is designed as a standalone library that can be integrated with the custom runtime environment used by the platform. It consists of two main components: the core communication library and the remote backends.

1. **Core communication library**: The communication library provides the fundamental mechanisms for message passing and collective operations among workers. It is designed to optimize data transmission by using *zero-copy* techniques for local communications and efficiently managing remote data transfers. The library is transparent to the workers: the local and remote communication mechanisms are abstracted away.

2. **Remote backends**: The middleware is extensible with multiple backends, each utilizing different technologies to handle remote messaging. Current backends include Redis, DragonflyDB, RabbitMQ and S3. The backends are responsible for managing the communication between workers across different machines.

Figure 3.1 provides an overview of the BCM architecture, illustrating how the core communication library interacts with the remote backends to facilitate efficient communication among workers. The core library abstracts the communication mechanisms, allowing workers to interact with the BCM through a **unified interface**.

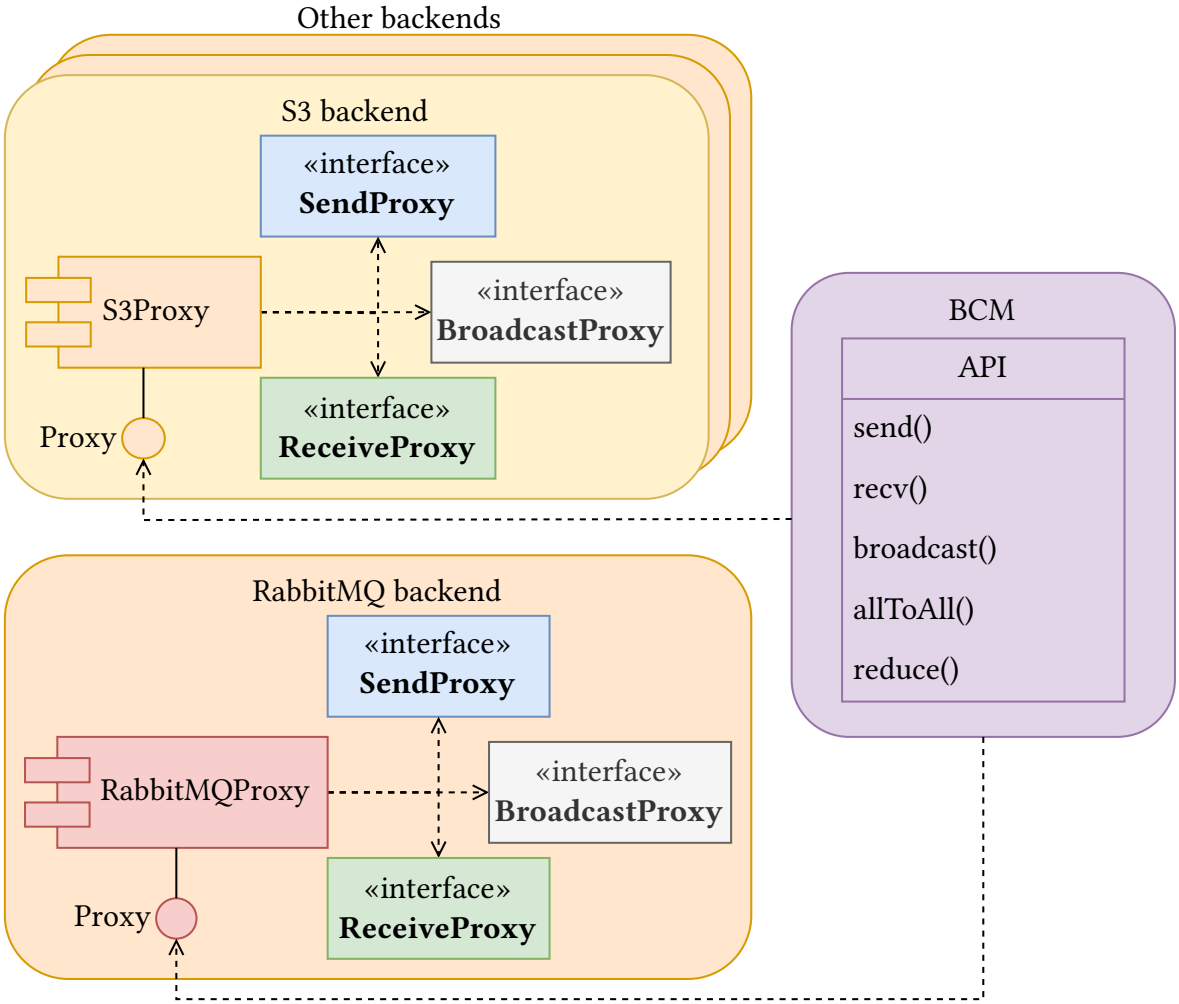


Fig. 3.1 *BCM* architecture overview.

3.2 Communication interface

The ***Burst Communication Middleware*** (BCM) is designed to provide straightforward and efficient communication between workers, drawing inspiration from the **Message Passing Interface** (MPI). The communication model is **elastic**, meaning it adapts to the *burst* size dynamically, ensuring that communication patterns scale with the number of workers. Workers utilize the BCM through the *burst context* object, which abstracts the underlying complexities of the communication infrastructure.

The BCM supports two fundamental communication primitives: **send** and **receive**. These primitives enable point-to-point communication, allowing one worker to send data

directly to another. The design ensures efficient data transfer and supports arbitrary data volumes, making it suitable for a wide range of parallel processing tasks.

In addition to point-to-point communication, the BCM supports several collective communication operations as seen in table 2.2. These operations are essential for coordinating and aggregating data among multiple workers. The supported collectives include:

- **Broadcast**: This operation allows a single worker to send data to all other workers in the *burst*.
- **All-to-all**: This collective operation enables each worker to send data to every other worker.
- **Reduce**: This operation aggregates data from all workers, applying a specified reduction function to combine the data into a single result.

This approach optimizes communication performance while keeping the programming model simple and **transparent** to the user: the programs are agnostic to the beforementioned locality optimizations. The communication primitives and collectives are designed to be **locality-aware**. Workers that are co-located within the same pack use shared memory for communication, which is faster and more efficient. For workers located in different packs, the BCM handles remote communication over the network.

3.3 BCM implementation

The Burst Communication Middleware (BCM) is implemented in **Rust**, encompassing approximately 5K SLOC. This middleware is designed to work seamlessly with the custom Rust runtime for OpenWhisk, and its architecture allows for integration with other programming languages through appropriate bindings. The prototype implementation of BCM is available for public access on GitLab.[1]

3.3.1 Language choice and asynchronous programming

Rust was chosen as the implementation language for the BCM due to its performance, safety, and support for **asynchronous programming** [47]. Asynchronous programming, or `async/await`, is a **concurrent programming model** that allows running multiple tasks on a small number of OS threads [2]. This model is essential for the BCM to handle concurrent communication tasks efficiently. However, asynchronous programming in Rust is challenging due to the language's strict memory safety guarantees.

The BCM facilitates efficient *intra-pack* and *inter-pack* communication. Intra-pack communication is achieved using *zero-copy* techniques, while inter-pack communication relies on remote backends. In order to accomplish this, each worker is provided with

[1] https://gitlab.com/burstcomputing/middleware

contextual information about the *flare* it is part of, which includes identifiers for the *flare*, pack, and worker, the total *burst* size, and the distribution of workers across packs. This contextual information is provided by the invoker, as described in section 2.3.2.

The BCM is initialized by the runtime environment and is accessible to workers through the `BurstContext` object in the `work` function, as shown in table 2.2.

3.3.2 Technical details

The BCM leverages the `tokio` library [16] for asynchronous I/O operations, enabling efficient, non-blocking communication between workers, both locally and remotely. The `tokio` library provides a robust foundation for building high-performance, network applications in Rust.

`Tokio` offers several key features that are essential for the BCM:

- **Task management**: `tokio` provides a task scheduler that manages the execution of asynchronous tasks, allowing the BCM to perform multiple operations concurrently. It also includes utilities for spawning and managing tasks, synchronization primitives (semaphores, mutexes, ...), communication channels, and timers.
- **Asynchronous I/O**: `tokio` enables workers to perform I/O operations concurrently, ensuring that communication tasks do not block the execution of other tasks. Both local and remote communication operations are implemented using `tokio`'s asynchronous I/O capabilities.
- **Runtime**: `tokio` provides a runtime that abstracts the underlying asynchronous I/O implementation. The runtime uses the operating system event queue (such as `epoll` on Linux) to efficiently manage I/O events.

When the custom Rust runtime environment initializes the BCM, it also initializes the `tokio` runtime, which manages the asynchronous tasks and I/O operations. This setup is done once per pack, as mentioned previously.

Moreover, `tokio` is **multi-threaded**, meaning that it can utilize multiple OS threads to execute asynchronous tasks concurrently. Therefore, with one `tokio` runtime per pack, one worker can benefit from multiple threads to perform I/O operations concurrently (e.g., using multiple connections to the remote backend).

3.3.3 Actor model for communication

The user code executed by workers is programmed in a synchronous manner, while the BCM is designed to handle asynchronous communication. The synchronous nature of the user code is due to the **MPI programming model** being adopted, which is inherently synchronous. This design choice simplifies the programming model for users, as they can write their code without worrying about asynchronous communication mechanisms.

The discrepancy between the synchronous user code and the asynchronous communication mechanisms of the BCM must be bridged to ensure efficient communication. To bridge this gap, the BCM employs an **actor model** [40, 12]. The actor model is another concurrent computational model in which actors are independent entities that communicate by sending messages to each other. In Rust, actors are implemented as asynchronous tasks that communicate through message passing using `tokio` channels.

Each worker has its corresponding **actor** that interacts with the BCM through message passing to bridge the synchronous user code with the asynchronous communication mechanisms. The actor model ensures that the user code remains synchronous while the communication tasks are handled asynchronously by the BCM. In a future iteration, support for asynchronous user code execution may be added to further enhance the platform's performance.

The BCM in turn manages the communication between workers, both locally and remotely, using the actor model. It uses the **proxy** pattern [22] to abstract the communication logic of multiple backends (e.g., Redis, RabbitMQ, S3), allowing workers to interact with the BCM through a unified interface. This design is depicted in fig. 3.1. The intra-pack communication is handled as a special backend that uses shared memory, while the inter-pack communication is managed by remote backends.

3.3.4 Intra-pack communication

For **intra-pack** communication (local), the BCM leverages **in-memory queues** to facilitate data exchange between workers within the same pack. Since the Rust runtime spawns workers as threads within a single process, these workers share the same memory space. Consequently, traditional shared memory mechanisms such as `shm_open` or `mmap` are unnecessary. Instead, workers can pass memory pointers directly to one another.

Furthermore, Rust's inherent memory safety guarantees ensure that this pointer-based communication remains **thread-safe**. For instance, in a *broadcast* operation, the root worker can send a read-only memory pointer to all other local workers, allowing them to read the broadcasted data concurrently without the need for locks or synchronization mechanisms and preventing data races. If a worker needs to modify the received data, it can employ copy-on-write (COW) mechanisms.

The BCM uses `tokio` **channels** to implement intra-pack communication. `tokio` channels provide a mechanism for sending and receiving messages between asynchronous tasks. As mentioned in section 3.1.1, the BCM must differentiate between *direct* (one-to-one) and *broadcast* (one-to-many) messages to handle communication efficiently. To achieve this, the BCM uses two types of channels: `mpsc` (multi-producer, single-consumer) channels [14] and `broadcast` channels [13] (multi-producer, multi-consumer), respectively. The latter is used for *broadcast* operations, while the former is used for direct communication. The `broadcast` channel is also suitable for *single-producer, multi-consumer* scenarios, which is the case for the root worker in a *broadcast* operation.

The `tokio` channels meet the requirements of the BCM stated in section 3.1.1:

- **Shared memory utilization** and **zero-copy data transfer**: `tokio` channels allow workers to pass memory pointers directly to one another, enabling zero-copy data transfer.
- **Thread-safe communication**: `tokio` channels are thread-safe, allowing multiple workers to send and receive messages concurrently without any additional synchronization mechanisms.

3.3.5 Inter-pack communication

For **inter-pack** communication (remote), the BCM utilizes **remote backends** to transmit messages between workers located in different packs. Each pack maintains a shared connection pool to the remote backend, enabling all workers within the pack to send and receive messages concurrently. This approach maximizes the bandwidth utilization of the container, which is particularly advantageous in operations like *all-to-all*, where each worker needs to establish channels with every other worker.

When dealing with large messages, the data is divided into smaller **chunks** that are sent and received concurrently. This method optimizes network utilization by allowing receivers to start processing data as soon as the first chunk arrives, rather than waiting for the entire message to be transmitted to the backend. The chunk-based transmission approach helps in maintaining high throughput and reduces latency in data communication.

The BCM uses `tokio` **asynchronous tasks** to send and receive these message chunks. Tasks are spawned for each chunk, allowing multiple chunks from different messages and workers to be transmitted concurrently by the BCM. This approach ensures that the communication tasks do not block the execution of other operations, such as local communication.

The BCM supports multiple remote backends, and there is a good selection of **Rust libraries** that provide asynchronous APIs for these backends with `tokio` support. For instance, the Redis backend uses the `redis` crate [39], the RabbitMQ backend uses the `lapin` crate [38], and the S3 backend uses the official AWS SDK for Rust [37]. These libraries provide asynchronous APIs for interacting with the respective backends, ensuring that the BCM can efficiently manage remote communication tasks.

The approach of using `tokio` asynchronous tasks and libraries that support asynchronous I/O operations satisfies the requirements of the BCM stated in section 3.1.1:

- **Multiple backends**: The BCM supports multiple remote backends, each with its asynchronous API that integrates seamlessly with `tokio`.
- **Connection pooling**: The shared connection pool, specialized for each backend, manages the connections to the remote backend efficiently.

- **Chunked message transfer**: The BCM divides large messages into smaller chunks, enabling concurrent transmission and processing of data, which enhances network utilization and minimizes latency.

3.3.6 Backend interface for remote communication

The BCM is designed with **extensibility** in mind, enabling the integration of additional remote backends as needed. Currently, the BCM supports several remote backends, including Redis, DragonflyDB, RabbitMQ, and S3. This flexibility ensures that the middleware can be adapted to various system requirements and environments.

The backend interface within the BCM distinguishes between **direct** (one-to-one) and **broadcast** (one-to-many) messages, optimizing how messages are read from the backend server. This differentiation is crucial because direct messages are read only once, while broadcast messages must be read multiple times by different workers, so it is logical to optimize the broadcast message retrieval process.

Because of this distinction, the backend interface is straightforward, requiring only four operations to be implemented: `send`, `receive`, `broadcast`, and `broadcast_receive`. These operations are the interface of the *proxies* mentioned in section 3.3.3 and illustrated in fig. 3.1.

The `send` and `receive` operations are used for direct messages, while the `broadcast` operation is used for broadcast messages. Using these basic operations, the BCM can implement more complex collective operations, such as *all-to-all* and *reduce*, by combining multiple `send` and `receive` primitives.

The backend interface is designed to be flexible and extensible, allowing for the integration of additional remote backends as needed. Each remote backend has its own implementation of the backend interface, tailored to the specific requirements and capabilities of the backend.

The backend interface of the BCM meets the requirements outlined in section 3.1.1 by ensuring backend extensibility, allowing easy integration of additional backends. Using a unified backend interface, it effortlessly differentiates between direct and broadcast messages, uses appropriate backend commands for efficient message distribution and employs the most suitable data structures and commands for each backend. Additionally, it supports polling mechanisms for backends that require them, overall resulting in efficient communication management.

RabbitMQ backend

For instance, in RabbitMQ, the middleware utilizes **direct exchanges** for one-to-one messages and **fan-out exchanges** for one-to-many messages. This distinction ensures efficient message distribution and retrieval, as RabbitMQ's *fan-out* exchanges are designed specifically for broadcasting messages to multiple consumers.

Redis and DragonflyDB backends

In Redis and DragonflyDB, the middleware employs different data structures and commands to store messages based on the message type:

- **Direct messages (one-to-one)**. For direct messages, the middleware uses the Redis list key type for each pair of workers. The `RPUSH` command is used to send messages, while the `BLPOP` is used to receive them. If the queue is empty, `BLPOP` will block until a new message is available. When the source worker puts a message in the list, the destination worker will unblock and consume (delete) the message from the list.
- **Broadcast messages (one-to-many)**. For broadcast messages, the middleware uses string keys in Redis. The source worker uses the `SET` command to store the message. Multiple remote workers can then perform a `GET` operation to retrieve the message without deleting it and allowing concurrent access by all intended recipients.

In DragonflyDB, the middleware uses the exact same logic as Redis.

S3 backend

AWS S3 backend implementation presents a unique challenge as it lacks native **locking mechanisms**. As a result, workers must actively **poll** for messages. The receiver worker repeatedly lists the keys of the object storage, checking for new objects (messages) to consume. This polling mechanism, while not as efficient as blocking commands, ensures that messages are eventually received and processed.

To prevent excessive polling and reduce the number of requests to the S3 backend, the BCM implements a **request throttling** mechanism. This mechanism limits the rate at which workers can poll the S3 backend for messages, ensuring that the platform operates within the specified request limits. Additionally, the throttling mechanism limits the number of concurrent requests to the S3 backend.

The throttling mechanism is implemented using a semaphore and a **token bucket** algorithm inspired in part by the Rust `tokio` library [15]. The semaphore controls the number of concurrent requests, while the token bucket algorithm regulates the rate at which workers can poll the S3 backend.

Summary

Ideally, the remote communication backends should offer **locking mechanisms** to facilitate efficient message retrieval. This means that if a worker wants to receive a message that the source worker has not yet sent, the receiver worker should be able to block itself until the message is available. This approach is more efficient than polling, as it reduces the number of unnecessary requests and minimizes latency.

Another alternative is **push-based** communication, where the backend notifies the receiver worker when a new message is available. This approach is more efficient than

polling, as it eliminates the need for the receiver worker to continuously check for new messages. However, not all backends support push-based communication, so polling is often the only viable option.

3.3.7 Message reliability and ordering

To guarantee **at-least-once** delivery semantics, ensuring that no messages are lost, the BCM employs a sophisticated message-tracking system. This system maintains a count of direct messages sent between each pair of workers and for each collective operation. By including a unique ID in each message, the BCM can manage and correct issues related to duplicate or out-of-order messages effectively.

Message headers

Every message transmitted by the BCM includes a detailed header. This header contains critical information such as:

- **Burst ID**: The unique identifier of the current *burst* or *flare*.
- **Source and destination worker IDs**: The IDs of the sending and receiving workers. They are unique within the current *flare*.
- **Message type**: The type of message, which can be direct or collective.
- **Collective type**: If the message is part of a collective, this field specifies the type of collective operation.
- **Sequence number**: A unique sequence number (counter) is assigned to each message to ensure correct ordering.
- **Total number of chunks**: For chunked messages, this field indicates the total number of chunks in the message.
- **Chunk number**: For chunked messages, this field specifies the order of the current chunk within the message.

By including this information in the message header, the BCM can accurately process the messages in the correct order and detect any missing or duplicate messages.

Duplicate and out-of-order message handling

The BCM employs several mechanisms to handle duplicate and out-of-order messages:

Duplicate messages Duplicate messages can occur due to network issues or failures in the communication infrastructure, as only *at-least-once* delivery semantics are guaranteed. If a message with a counter less than the expected value is received, it is ignored and assumed to have been already processed, effectively eliminating duplicates.

Out-of-order messages As messages are transmitted asynchronously, they may arrive out of order.

- If a message with a counter greater than the expected value is received, it is temporarily cached. The middleware will then wait for the missing messages to arrive before processing the out-of-order message.
- When a worker attempts to receive a message, the BCM first checks the cache to see if the expected message is already there. If not, the worker will wait until the missing message arrives. This mechanism ensures that messages are processed in the correct sequence.

Chunked messages Large messages are divided into smaller chunks for transmission. These are also subject to duplicate and out-of-order issues.

- For large messages that are divided into smaller chunks, a memory region is pre-allocated for the entire payload. As chunks arrive, they are written to their designated offsets within this memory region.
- This approach allows the complete message to be reconstructed in the correct order once all chunks have been received, even if they arrive out of order.

In summary, the message tracking system and header information ensure that the BCM accomplishes the requirements presented in section 3.1.1, guaranteeing *at-least-once* delivery semantics and maintaining the correct order of messages. Even for large messages that are divided into chunks, the BCM can reconstruct the original message in the correct order, ensuring that workers process the data accurately.

3.3.8 Example *broadcast* implementation

To showcase how the BCM **transparently** handles local and remote communication, a simplified implementation of the *broadcast* operation is presented in listing 2. The *broadcast* operation is a collective communication operation that sends data from one worker to all other workers in the *burst*.

The *broadcast* operation is implemented as an asynchronous function that takes the data to be broadcast and the root worker ID as arguments. The root worker sends the broadcast message to both the local channel and the remote backend. Each *pack* has a group leader that

Listing 2 Simplified implementation of the *broadcast* operation in the BCM.

```rust
pub async fn broadcast(&mut self, data: Option<T>, root: u32) -> Result<Message<T>> {
    let counter = get_counter(CollectiveType::Broadcast);

    if self.worker_id == root {
        // Root worker sends the broadcast message
        let msg = create_message(self.worker_id, counter, data);

        // Send to local group
        self.local_broadcast.local_broadcast_send(msg).await;

        if self.enable_message_chunking {
            // Send chunked messages to remote workers
            let chunked_messages = chunk_message(msg, self.message_chunk_size);
            let futures = chunked_messages
                .into_iter()
                .map(|msg| self.remote_broadcast.remote_broadcast_send(msg))
                .collect::<FuturesUnordered<_>>();
            futures::future::try_join_all(futures).await;
        } else {
            // Send whole message to remote workers
            self.remote_broadcast.remote_broadcast_send(msg).await;
        }
    } else if !self.group.contains(root) && self.worker_id == self.group_worker_leader {
        // Group leader receives the message from remote and sends to local group
        // This function handles the chunking logic
        let msg = self.get_broadcast_message(root, counter).await;
        self.local_broadcast.local_broadcast_send(msg).await;
    }

    // All workers receive the broadcast message via the local channel
    let msg = self.local_broadcast.local_broadcast_recv().await;

    increment_counter(CollectiveType::Broadcast);

    Ok(msg)
}
```

receives the message from the remote backend and forwards it to the local group. Eventually, all workers receive the broadcast message via the local channel.

If chunking is enabled, the message is divided into smaller chunks, which are sent concurrently to the remote backend. Note that for the local channel, the message is sent as a single unit, as shared memory is used for intra-pack communication.

The implementation of the *broadcast* operation demonstrates how the BCM abstracts the complexities of local and remote communication from the user code. The user code can call the *broadcast* function without worrying about the underlying communication mechanisms, allowing for a seamless and efficient programming experience.

Chapter 4

Evaluation

4.1 Objectives

The evaluation of the *burst communication middleware* (BCM) is crucial to assess its impact on the performance of the platform. The main objective of the evaluation is to demonstrate the benefits of **group-awareness** and **locality** in the communication middleware. However, to ensure the feasibility and effectiveness of the BCM, it is important to first evaluate the invocation latency and worker simultaneity.

By evaluating the invocation latency, we can assure that the platform is capable of starting up a *burst* of workers in a timely manner. This is essential for the BCM to be effective, as it relies on the ability to launch workers quickly and efficiently. If a *burst* takes too long to start up, the benefits of the BCM may be negated by the increased latency. For example, if some workers in a *burst* take significantly longer to start up than others, it can lead to inefficiencies in inter-worker communication, as some workers may be waiting for others to become operational.

Additionally, worker simultaneity will be assessed to ensure that all workers in a *burst* operate with complete parallelism. This aspect is as important as the invocation latency, as it ensures that all workers in a *burst* start and function in unison. Without proper worker simultaneity, the BCM loses its significance: if workers are not coupled in time, the communication between them may be hindered.

4.2 Methodology

To achieve the objectives, a series of experiments were conducted. The experiments were designed to test and analyze various performance aspects of the platform and the BCM under different conditions. These performance aspects include the following:

- **Invocation latency:** The time it takes for a *burst* to start up and become operational is measured, compared to the traditional FaaS approach.

- **Simultaneity:** The ability of the platform to launch multiple workers concurrently is evaluated, ensuring they start and function in unison.
- **Code and data loading:** The influence of worker locality on the efficiency of loading code and data is examined, highlighting improvements over independent function invocations in FaaS.
- **Worker communication and collectives:** The effectiveness of inter-worker communication within a *burst* is assessed, focusing on the performance of the BCM in collective operations like broadcast and all-to-all.
- **Application performance:** The overall performance of BCM and the *burst computing* model is tested on three different applications: PageRank, TeraSort, and hyperparameter tuning.

The mentioned experiments were conducted using Amazon Web Services (AWS) within the same region (`us-east-1`) to maintain consistency and reliability in the results.

4.3 *Burst* group invocation

4.3.1 Setup

The *burst computing* platform operates on an Amazon Elastic Kubernetes Service (EKS) cluster. The control plane is hosted on `t4i.xlarge` VM (4 vCPUs, 16 GiB RAM). For the invokers, up to 20 `c7i.12xlarge` VMs (48 vCPUs and 96 GiB RAM) are used. This setup allows to support up to 960 workers simultaneously, with each worker having 1 vCPU.

4.3.2 Impact on *burst* invocation latency

To evaluate the effect of different packing granularities on *burst* invocation latency, a homogeneous packing policy is implemented. The results of this evaluation are presented in fig. 4.1, which illustrates the worker latency distribution for burst sizes of 48, 96, 480, and 960 workers.

A clear pattern emerges from the results: as the packing granularity increases, the start-up time for workers significantly decreases and the distribution of worker latencies becomes more uniform. For instance, with a *burst* size of 960, the time required for all workers to become operational decreases by a factor of 11.5 when moving from the smallest granularity (FaaS) to a granularity of 48. This improvement is primarily due to the time taken to create containers, which dominates the overall invocation latency.

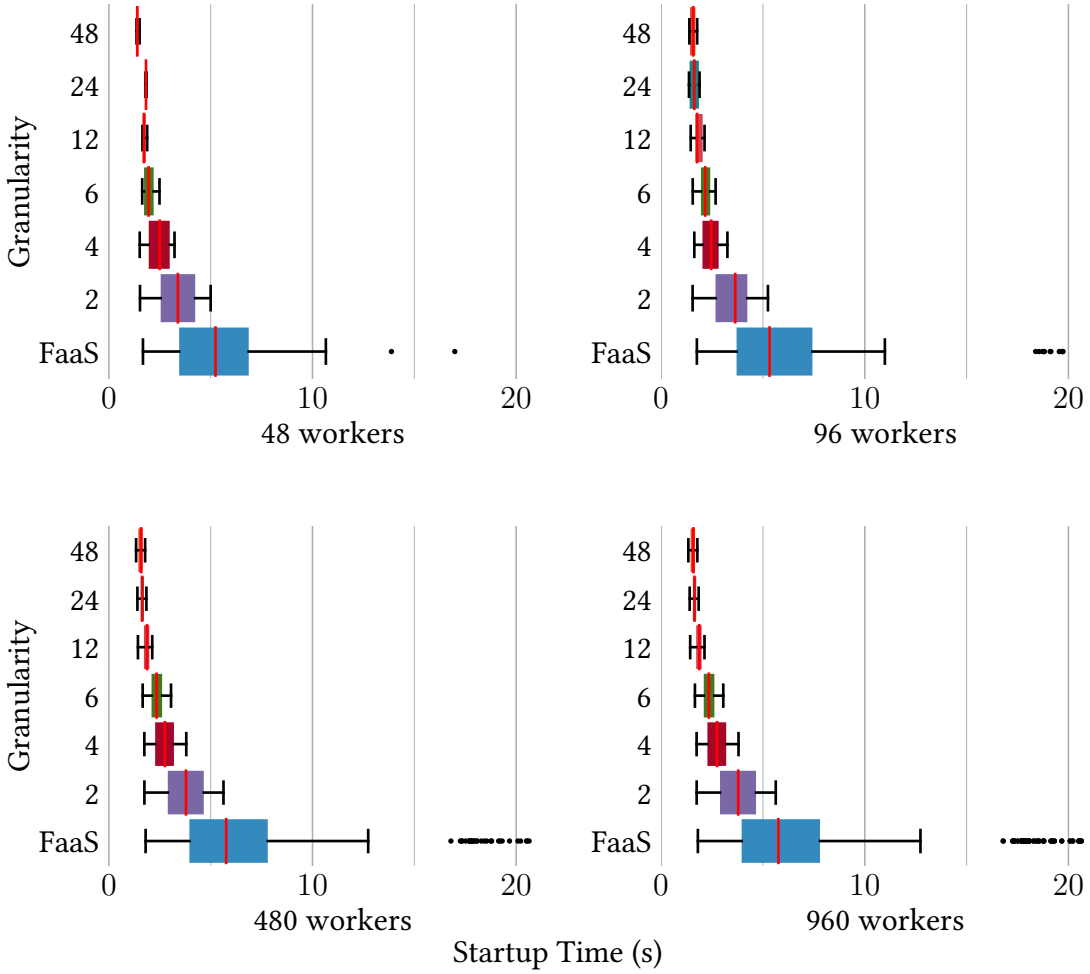


Fig. 4.1 *Burst* start-up time for different granularities and worker counts compared to FaaS (worker latency distribution).

4.3.3 Impact on worker simultaneity

One of the main objectives of the *burst computing* model is to ensure that all workers in a *burst* operate with complete parallelism, maximizing concurrency and minimizing wait times. For a *burst* to be effective, the tasks comprising it must start as simultaneously as possible. Any significant deviation in the initialization of some workers can lead to inefficiencies, especially when inter-worker communication is required.

To asses the worker simultaneity, an experiment was conducted comparing a *burst* job of size 960 on traditional FaaS with *burst computing* configured with a granularity of 48. Each worker in this experiment was programmed to perform a simple task: a 5-second sleep. The execution timelines of the workers are plotted and analyzed in fig. 4.2.

The analysis of the results reveals that the *burst computing* model significantly outperforms FaaS in terms of resource allocation speed and worker readiness. Workers are initialized much faster and more uniformly, which is crucial for ensuring effective parallelism.

A closer look at the dispersion of worker start-up times (also seen in fig. 4.1) further highlights this advantage. In the FaaS setup, the range of worker start-up times is 18.8 s, with a median average deviation (MAD) of 2.65 s. This wide range indicates substantial variability in when workers become ready, which in turn hampers full parallelism.

In contrast, the *burst computing* model with a granularity of 48 demonstrates a much narrower range of start-up times, just 0.44 s, with a MAD of only 0.1 s. This represents a 43× reduction in the range of start-up times and a 26.5× decrease in variability compared to FaaS.

The reduced dispersity in worker start-up times means that all workers can start executing their tasks almost simultaneously, achieving full parallelism from the onset. By guaranteeing worker simultaneity, *burst computing* ensures that inter-worker communication is feasible and efficient, even for large-scale *bursts*.

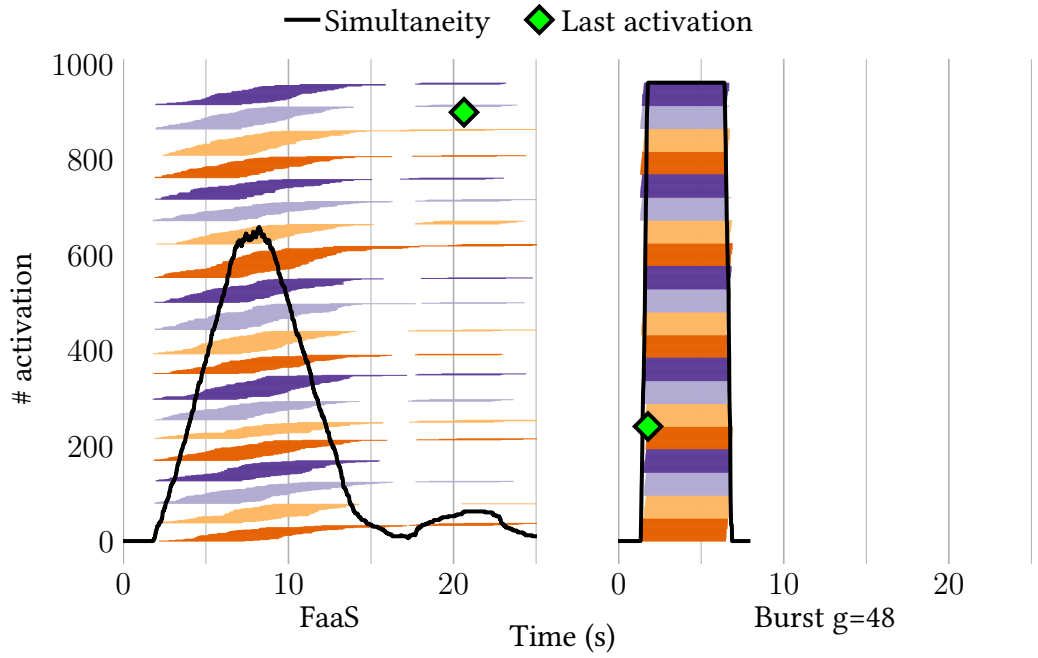


Fig. 4.2 Simultaneity in FaaS (left) vs *Burst* with granularity 48 (right) and 1000 workers. Each horizontal line represents the life-time of a worker and colors indicate different ivoker VMs.

4.3.4 Impact on code and data loading

Burst computing addresses a significant challenge in FaaS systems: the inefficiency of loading the same data across multiple invocations. This inefficiency is particularly problematic because it leads to redundant data transfers and increased latency. In *burst computing*, this issue is alleviated by implementing mechanisms that allow data to be downloaded once per pack, significantly reducing the overall data ingestion load.

Moreover, *burst computing* takes advantage of the parallelism within each pack to speed up data transfer. By using parallel object storage byte range reads *burst computing* can

leverage multiple workers to download different parts of the same object simultaneously. This optimization ensures that the data is loaded more efficiently compared to FaaS, where each worker must download the data independently.

To evaluate the effectiveness of the approach, experiments were conducted using different granularity setups. The results, presented in fig. 4.3, illustrate the performance improvements. Specifically, a burst of 96 workers was tasked with downloading a 1 GB object from Amazon S3. The comparison between FaaS and *burst computing* with a granularity of 48 shows a remarkable speedup of 32.6× in download time.

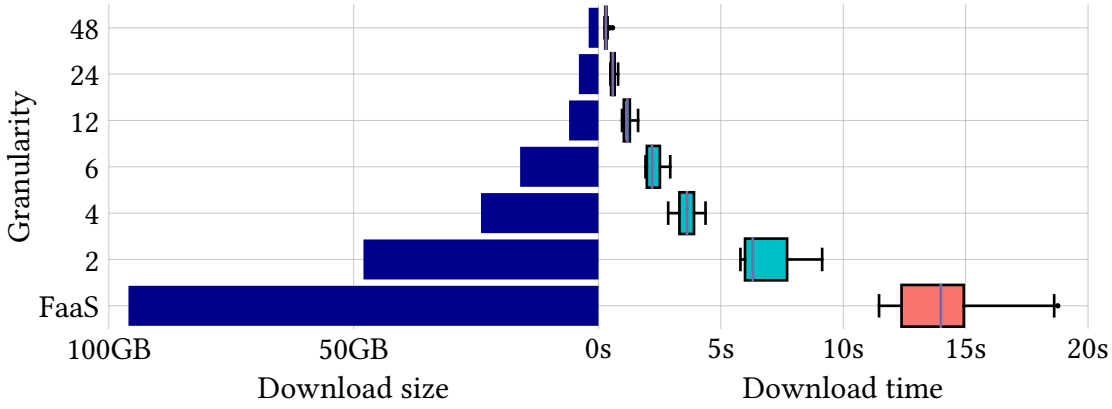


Fig. 4.3 A *burst* of 96 workers downloading a 1 GB object from S3. Download times are compared between FaaS and different granularities in *burst computing*.

4.3.5 Takeaway

The introduction of the `flare` mechanism significantly reduces the initialization time of workers in a *burst*, thus addressing the friction point (F1). This achieves, based on the results of the experiments, faster group initialization, which is 11.5× quicker compared to traditional methods, and higher worker parallelism, with a 26.5× reduction in worker dispersion. The increased simultaneity facilitates worker *packing*, which, in turn, enhances the **locality**

Improving locality accelerates data download processes within applications. The use of the proposed mechanisms in the *burst computing* model can lead to a 32.6× speed-up in data download times, addressing the third friction point (F3) associated with traditional FaaS platforms.

4.4 *Burst* inter-pack communication

Before evaluating the impact of the Burst Communication Middleware (BCM) on friction points (F2) and (F3), it is essential to verify the feasibility of an indirect communication model and identify a backend capable of handling burst loads at scale. To achieve this, the

throughput of various indirect communication backends was measured. The tested backends include Redis, DragonflyDB (a multi-threaded, Redis-compatible alternative), RabbitMQ, and Amazon S3. For Redis and DragonflyDB, two configurations were evaluated: using lists and using streams.

4.4.1 Message chunk size optimization

The BCM divides messages into several smaller chunks to optimize network utilization and enable parallel read and write operations. This chunking allows the receiving end to start processing the message as soon as the first chunk is available, enhancing the overall communication efficiency. However, determining the optimal chunk size involves balancing latency to the first byte and operational overhead. This optimal size can vary depending on the communication backend used.

To identify the best configuration, experiments were conducted to measure the throughput of transferring a 1 GiB message between two remote workers. The workers operated on two `c7i.large` VMs (2 vCPUs, 4 GiB RAM and up to 12.5 Gbps network bandwidth), while a `c7i.16xlarge` VM (64 vCPUs, 128 GiB RAM and 25 Gbps network bandwidth) served as the intermediate server. The results of these experiments are depicted in fig. 4.4.

The analysis revealed that different backends exhibited varying performance based on chunk size:

- **RabbitMQ:** RabbitMQ maintained a constant throughput and latency for larger chunk sizes but was limited by the AMQP protocol, which restricts payload sizes to a maximum of 128 MiB. This limitation impacts its suitability for handling larger messages in *burst computing* scenarios.

- **Redis and DragonflyDB:** both Redis and DragonflyDB performed optimally with a chunk size of 1 MB. DragonflyDB showed a slight advantage over Redis in terms of throughput and latency, likely due to its multi-threaded architecture, which allows for better utilization of server resources and handling of concurrent operations.

- **Amazon S3:** S3 demonstrated the lowest throughput and highest latency across all chunk sizes. This is primarily because object stores like S3 are not optimized for handling small file transfers. When dealing with chunk sizes of 1 MB or less, S3 quickly exceeds the allowed service request rate limits, resulting in significant performance degradation.

4.4.2 Maximum throughput and scalability

To evaluate how different communication backends handle parallel load and scale, the aggregated throughput between multiple pairs of workers communicating simultaneously is measured. In this experiment, workers are organized into two remote groups, with each

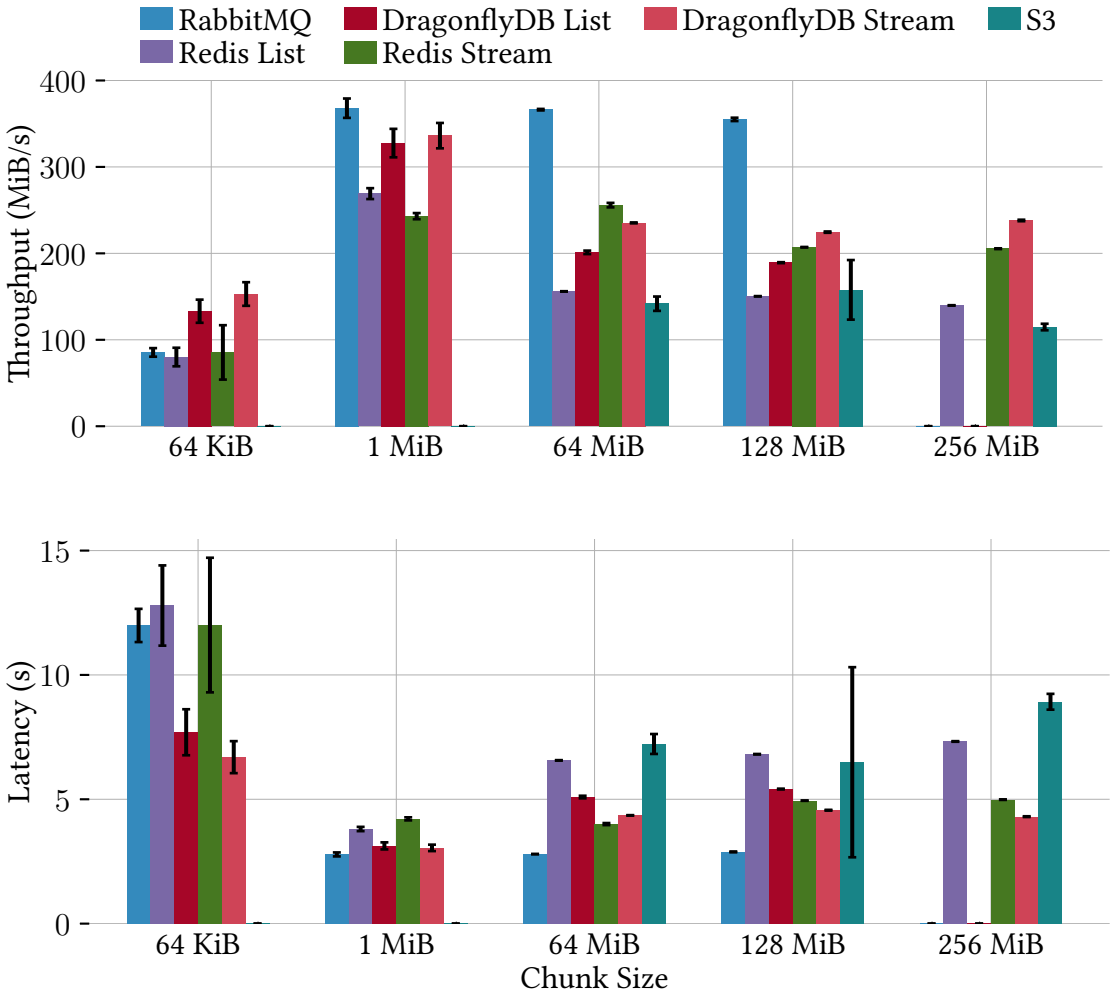


Fig. 4.4 Throughput and latency between two remote workers sending a 1 GiB payload chunked in different sizes. Median values with standard deviations are shown for 10 runs.

worker in group A sending a fixed-size message (256 MiB) to a corresponding worker in group B. The total volume of data transferred increases with the burst size, which ranges from 8 to 384 workers. Each backend uses the optimal chunk size determined in the previous micro-benchmark.

The experimental setup involves scaling the worker VMs according to the burst size: from `c7i.xlarge` VMs (4 vCPUs, 8 GiB RAM and up to 12.5 Gbps network bandwidth) for 8 workers to `c7i.48xlarge` VMs (192 vCPUs, 384 GiB RAM and 50 Gbps network bandwidth) for 384 workers. The communication server is consistently run on a `c7i.48xlarge` VM (192 vCPUs, 384 GiB RAM and 50 Gbps network bandwidth) to ensure sufficient resources for handling the communication load. The results of this experiment are presented in fig. 4.5.

The performance analysis showed varying scalability among the different backends:

- **RabbitMQ:** RabbitMQ reached a maximum throughput of 1 GiB/s but failed to scale effectively beyond this limit. This indicates that RabbitMQ might not be suitable

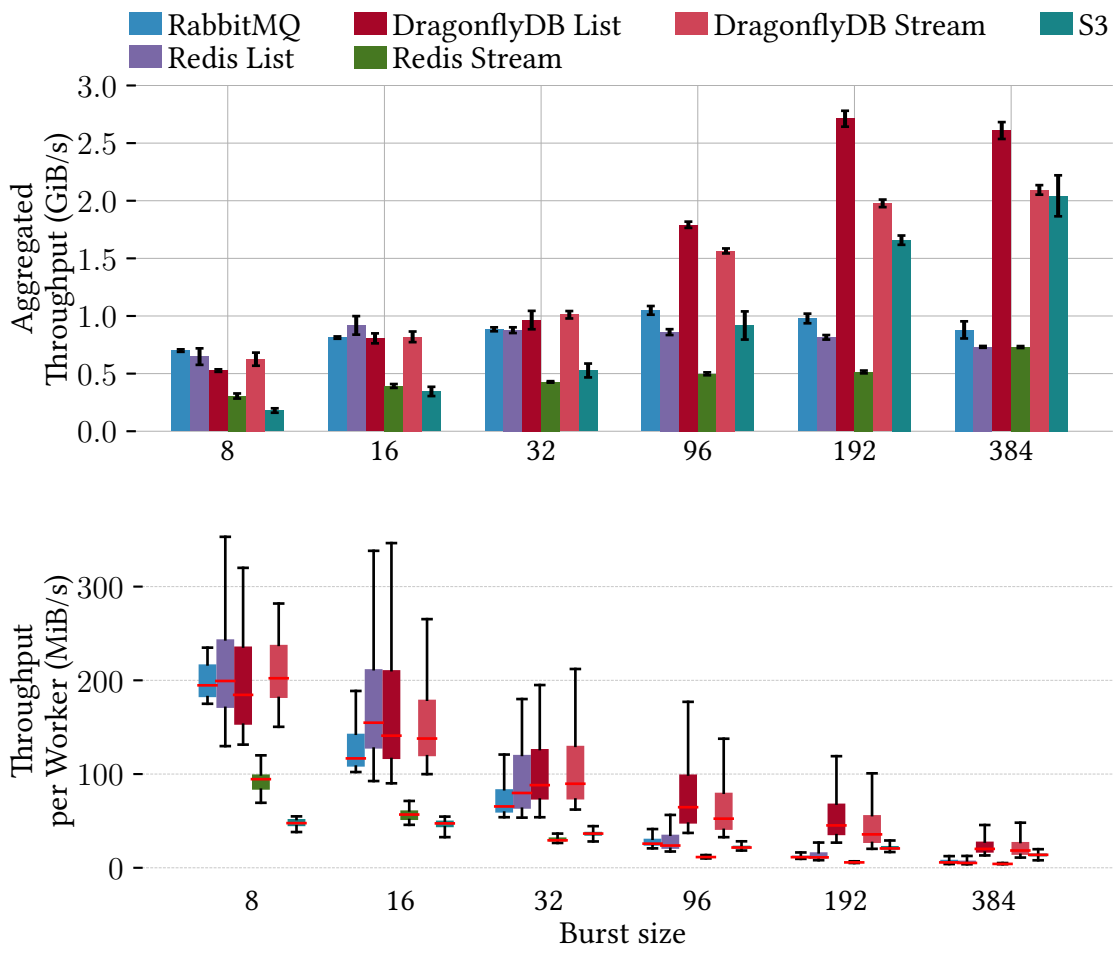


Fig. 4.5 Aggregated throughput and throghput per worker of two remote *packs*, A and B, of different sizes. Each worker from *pack* A sends a 256 MiB payload to another worker in *pack* B. Median values with standard deviations are shown for 10 runs.

for scenarios requiring high levels of parallel communication due to its inherent limitations in handling large volumes of data concurrently.

- **Redis and DragonflyDB:** in-memory stores Redis and DragonflyDB were tested with two configurations: using lists and using streams. The list-based approach outperformed the stream-based approach in both cases. Redis, being single-threaded, did not scale well with increasing parallelism, limiting its throughput. On the other hand, DragonflyDB, with its multi-threaded architecture, scaled effectively with parallelism and achieved the highest throughput, surpassing 2.5 GiB/s for large burst sizes.

- **Amazon S3:** Although S3 scaled with parallelism, its throughput remained slower compared to the in-memory stores. This is consistent with S3's design, which is optimized for larger, less frequent transactions rather than high-frequency, small-payload communications.

4.4.3 Takeaway

The results of the experiments demonstrate the feasibility of using an indirect communication model in *burst computing*, as it provides high throughput and scalability. The conducted micro-benchmarks reveal that some backends can sustain significant loads, handling up to 384 workers with individual connections effectively. Among the tested backends, DragonflyDB with a list-based approach and 1 MiB chunk size showed the best performance, making it the most suitable choice for the BCM in *burst computing*.

Given these findings, DragonflyDB List with 1 MiB chunk size will be used for the remaining evaluations.

4.5 *Burst* group collectives

This section examines the impact of worker locality on the performance of group collectives. The main focus of *burst communication middleware* (BCM) is to address the friction point F3 by optimizing communication between workers in a *burst*.

End-to-end latency is measured for collective operations, specifically focusing on the *broadcast* and *all-to-all* collectives. End-to-end latency is defined as the total time it takes for all workers to complete the collective. The objective of this evaluation is to observe how varying the packing granularity affects performance.

It's important to note that the *reduce* collective exhibits similar behaviour to the *broadcast* due to their analogous data movement patterns. Future work could explore other collectives like *gather* and *scatter*, which are expected to show similar behaviour to *all-to-all*.

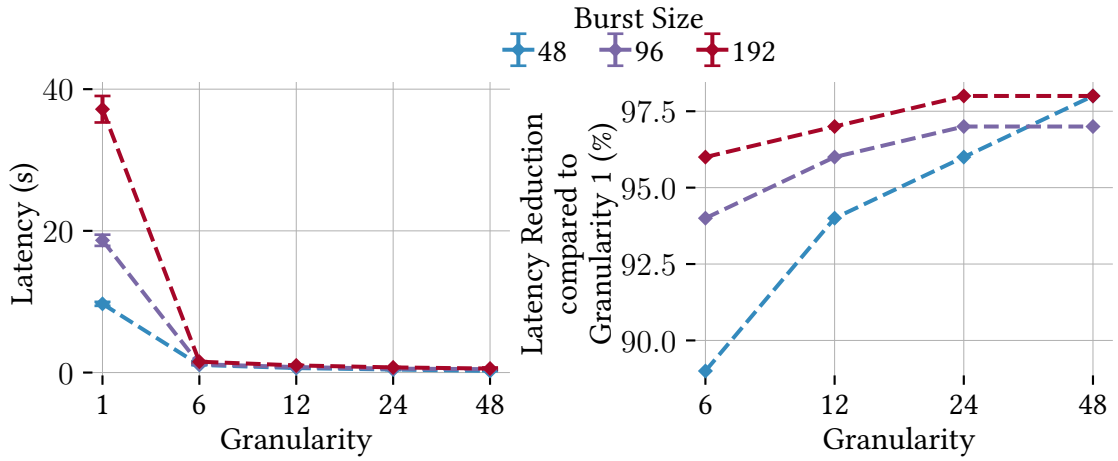
4.5.1 Experimental setup

The setup involves launching *bursts* of different sizes with varying packing granularities. The *bursts* are executed on one, two or four `c7i.12xlarge` VMs (48 vCPUs, 96 GiB RAM and 18.75 Gbps network bandwidth), corresponding to *burst* sizes of 48, 96, and 192 workers, respectively. The granularity ranges from 1 (FaaS) to 48. Each worker processes 256 MiB of data for each collective call. The backend server responsible for managing these operations runs on a `c7i.48xlarge` VM (192 vCPUs, 384 GiB RAM and 50 Gbps network bandwidth), which provides ample computational power and memory.

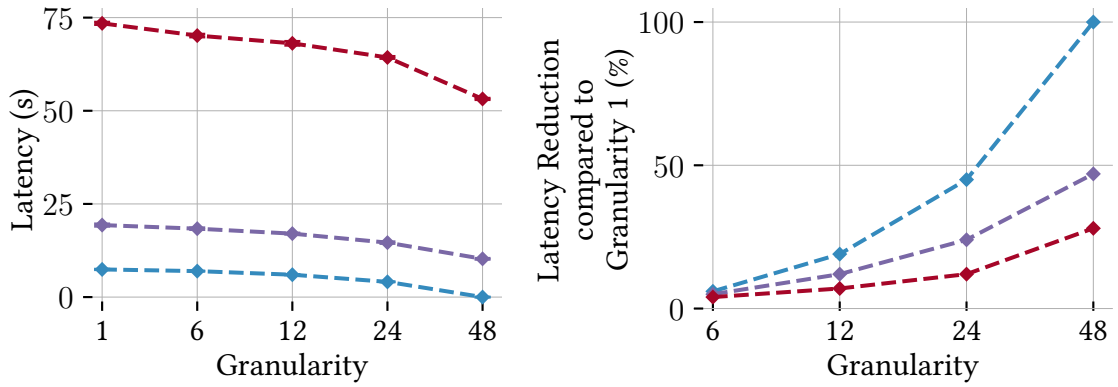
4.5.2 Results and analysis

The results of the experiments are presented in fig. 4.6. Overall, the latency of collective operations decreases significantly as the granularity increases. This reduction is primarily due to the decrease in remote communication, which is the main bottleneck in collective operations. When more data movement occurs locally within the same pack, the overall communication latency drops.

Moreover, if the amount of remote communication is reduced through locality, more bandwidth is available for data transfer, which further decreases latency. The cost of local communication is insignificant compared to remote communication.



(a) Broadcast



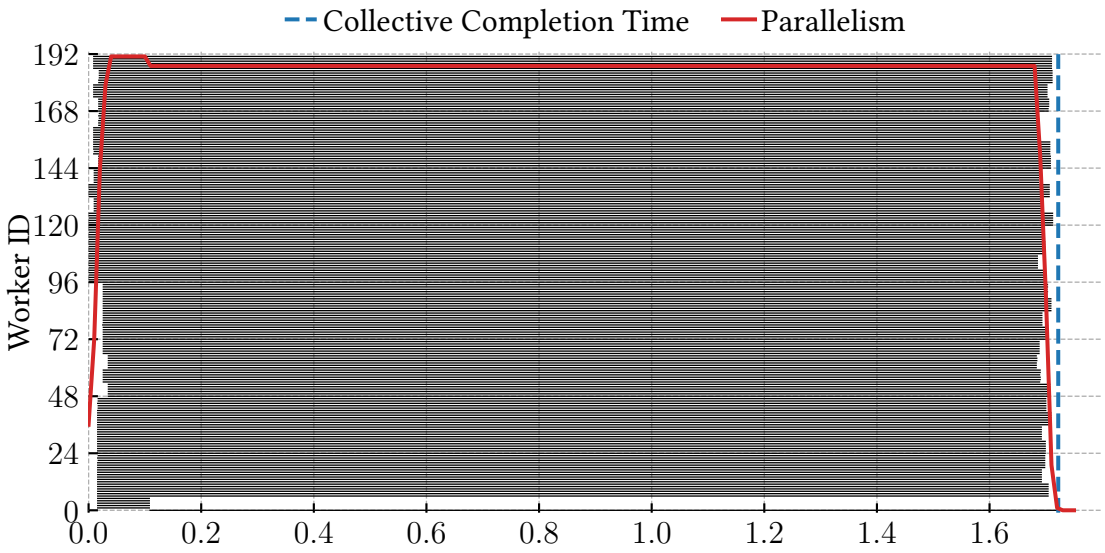
(b) All-to-all

Fig. 4.6 Latency and latency reduction percentage with respect to granularity 1 of *broadcast* and *all-to-all* operations for different granularities and burst sizes. Median values with standard deviations are shown for 10 runs.

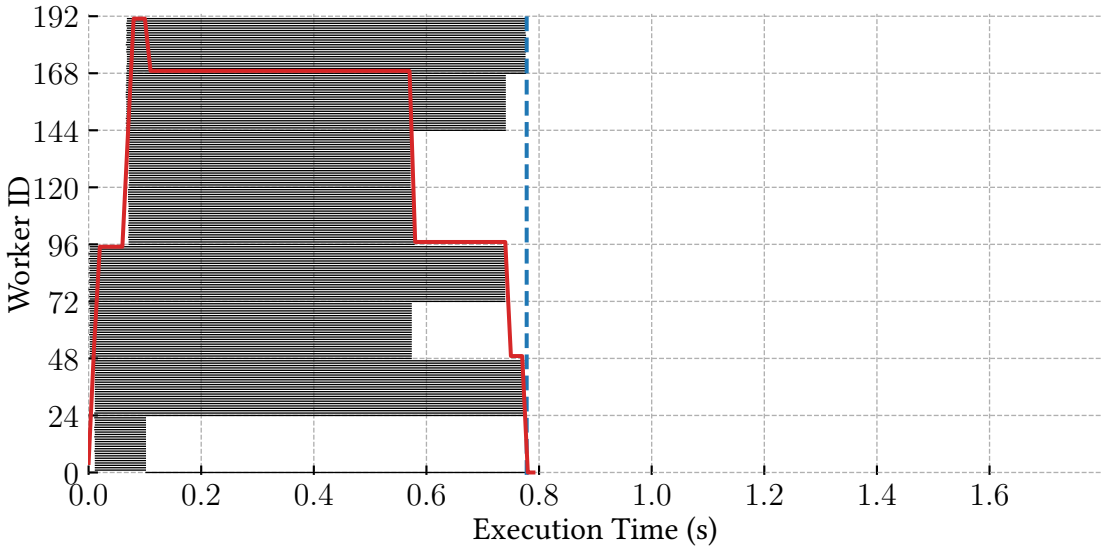
Broadcast

In the *broadcast* collective, a message is sent once but read once per pack. Consequently, the volume of remote data movement is directly proportional to the number of packs. For example, halving the number of packs reduces the amount of remote data traffic by half. This results in a rapid decrease in latency as the packing granularity increases, even for small granularities. The experimental data shows that at the highest granularity of 48 workers per pack, the latency reduction is nearly 90% as illustrated in fig. 4.6a.

The timeline of the *broadcast* collective operation in fig. 4.7 illustrates the impact of **locality** on the latency of the operation. The workers within the same pack finish at the same time, as the broadcast message is read once per pack. The root worker, which sends the



(a) Granularity 6



(b) Granularity 24

Fig. 4.7 Timeline of *broadcast* collective operation with a *burst* size of 192 workers and different granularities. Each worker is represented as a black line, and the parallelism at each instant of time is shown in red.

broadcast message, finishes last, as it must send the complete message to the intermediate server. It is important to note that the workers that belong to the same group as the root worker finish earlier than the workers in other groups, as they receive the message via local communication.

All-to-all

The *all-to-all* collective is more data-intensive than the *broadcast* because each worker must communicate with every other worker, resulting in a quadratic increase in data movement.

For a *burst* size of 192 workers, each worker sends 256 MiB of data to every other worker, totalling 48 GiB of data. Even if the *burst* is split into just two packs (granularity 96), half of the data must be transferred remotely. This high volume of remote data movement is evident in the latency results shown in fig. 4.6b. At the highest granularity of 48 workers per pack, the reduction in latency for *burst* sizes of 48, 96 and 192 workers is approximately 100%, 50%, and 25%, respectively.

4.5.3 Theoretical vs measured reduction

To further analyze the impact of packing granularity on collective operations, the theoretical reduction in remote data transfer is compared to the measured latency reduction. But first, a theoretical model is developed to predict the reduction in remote data transfer based on the packing granularity.

All-to-all collective

Assuming each worker sends 1 B of data to every other worker, the amount of data transferred remotely D_r for an *all-to-all* collective with a *burst* size of N workers and granularity of G can be calculated as shown in eq. (4.1).

$$D_r(N, G) = N \times (N - G) \tag{4.1}$$

For a *burst* with granularity 1, the amount of remote data transfer is maximized and becomes the baseline for comparison: $D_r(N, 1) = N \times (N - 1)$. From this, the theoretical reduction in remote data transfer R_r for different granularities can be calculated as a percentage with respect to granularity 1 using eq. (4.2).

$$R_r(N, G) = \frac{D_r(N, 1) - D_r(N, G)}{D_r(N, 1)} \times 100 \tag{4.2}$$

The equation can be further simplified to eq. (4.3). In table 4.1, the theoretical reduction in remote data transfer for *all-to-all* collective is calculated for different granularities and *burst* sizes using eq. (4.3).

$$R_r(N, G) = \frac{G - 1}{N - 1} \times 100 \tag{4.3}$$

The results of the comparison between the theoretical and measured reduction are presented in fig. 4.8 for the *all-to-all* collective. *Burst* sizes of 48 and 96 workers are considered, with granularities ranging from 6 to 48 and compared with granularity 1. The data shows that the measured latency reduction closely aligns with the theoretical reduction calculated prior to the experiments. This indicates that the theoretical model accurately predicts the performance improvements achieved by increasing the packing granularity.

Table 4.1 Theoretical reduction in remote data transfer for *all-to-all* collective operations. The reduction is calculated as a percentage with respect to granularity 1.

	Burst size			
Granularity	**48**	**96**	**480**	**960**
1	0.00	0.00	0.00	0.00
2	2.13	1.05	0.21	0.10
4	6.38	3.16	0.63	0.31
6	10.64	5.26	1.04	0.52
12	23.40	11.58	2.30	1.15
24	48.94	24.21	4.80	2.40
48	100.00	49.47	9.81	4.90
96		100.00	19.83	9.91
480			100.00	49.95
960				100.00

(a) *Burst* size 48 **(b)** *Burst* size 96

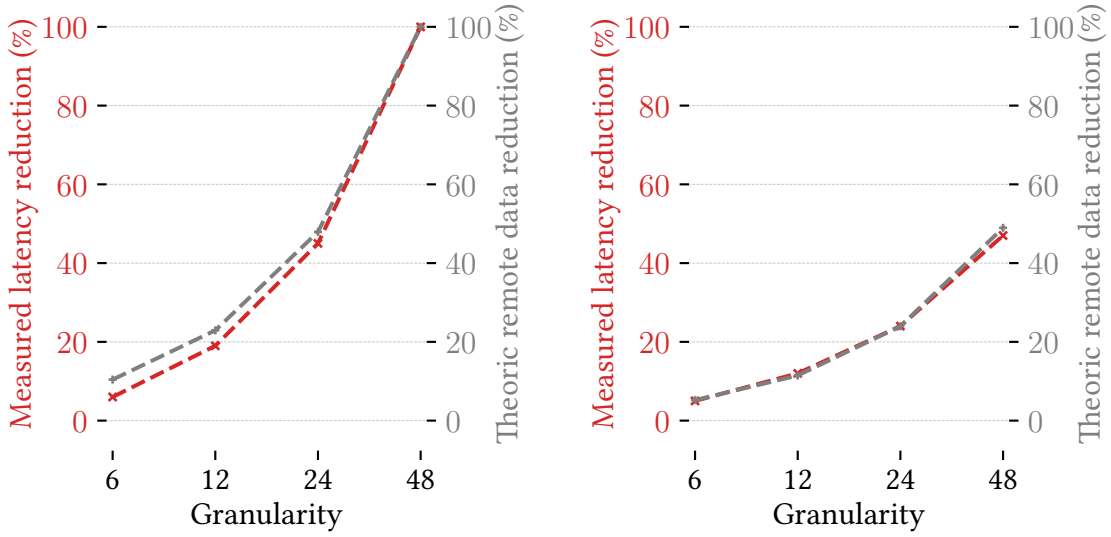


Fig. 4.8 Theoretical remote data transfer reduction compared to measured latency reduction in *all-to-all* operation for different granularities and burst sizes. The reduction is calculated as a percentage with respect to granularity 1.

Broadcast collective

The theoretical reduction in remote data transfer for the *broadcast* collective can be calculated with a similar approach as for the *all-to-all* collective.

Assuming the root worker sends a 1 B broadcast message and each pack reads it once, the amount of data transferred remotely D_r for a *broadcast* collective with a *burst* size of N workers and granularity of G can be calculated as shown in eq. (4.4).

$$D_r(N, G) = \frac{N}{G} - 1 \tag{4.4}$$

Table 4.2 Theoretical reduction in remote data transfer for *broadcast* collective operations. The reduction is calculated as a percentage with respect to granularity 1.

	Burst size			
Granularity	**48**	**96**	**480**	**960**
1	0.00	0.00	0.00	0.00
2	51.06	50.53	50.26	50.05
4	76.60	75.79	75.39	75.08
6	85.11	84.21	83.77	83.42
12	93.62	92.63	92.15	91.76
24	97.87	96.84	96.34	95.93
48	100.00	98.95	98.43	98.02
96		100.00	99.48	99.06
480			100.00	99.90
960				100.00

For a *burst* with granularity 1, the amount of remote data transfer is maximized and becomes the baseline for comparison: $D_r(N, 1) = N - 1$.

The theoretical reduction in remote data transfer R_r for different granularities can be calculated as a percentage with respect to granularity 1 using eq. (4.5).

$$R_r(N, G) = \frac{D_r(N, 1) - D_r(N, G)}{D_r(N, 1)} \times 100 \tag{4.5}$$

The equation can be further simplified to eq. (4.6). In table 4.2, the theoretical reduction in remote data transfer for *broadcast* collective is calculated for different granularities and *burst* sizes using eq. (4.6).

$$R_r(N, G) = \frac{NG - N}{NG - G} \times 100 \tag{4.6}$$

The comparison between the theoretical and measured reduction for the *broadcast* collective is presented in fig. 4.9 for *burst* sizes of 48 and 96 workers. The results show that the measured latency reduction closely aligns with the theoretical reduction, indicating that the theoretical model accurately predicts the performance improvements achieved by increasing the packing granularity.

4.5.4 Takeaway

Locality-aware group collectives significantly mitigate friction point **F3** by reducing the volume of remote data movement. By optimizing the packing granularity based on the communication pattern of the application, *burst computing* ensures that most data movement occurs locally, thereby improving the overall efficiency and performance of collective operations.

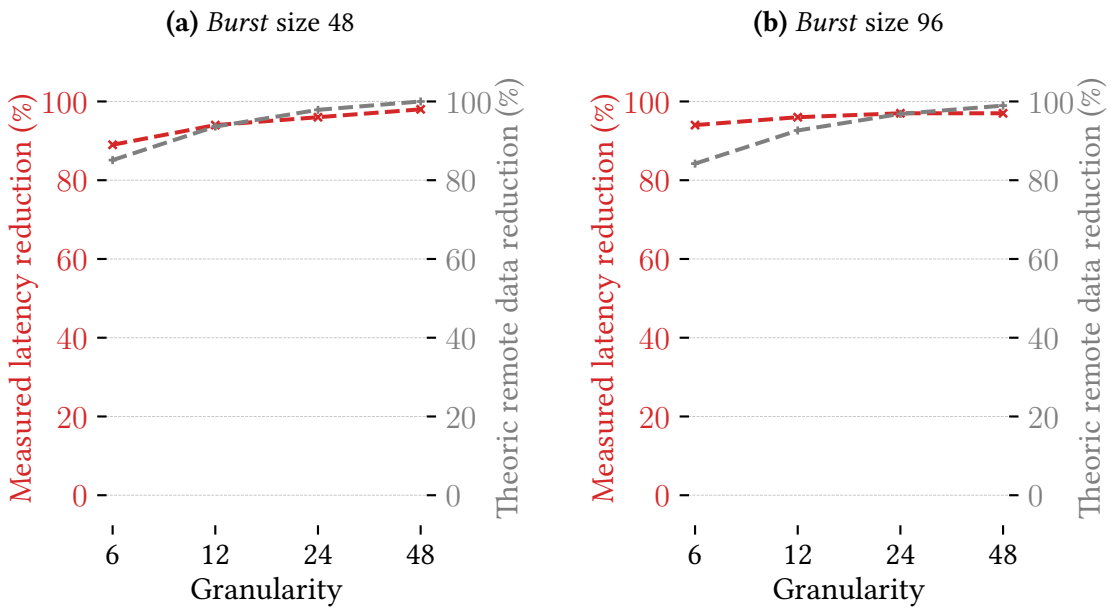


Fig. 4.9 Theoretical remote data transfer reduction compared to measured latency reduction in *broadcast* collective for different granularities and burst sizes. The reduction is calculated as a percentage with respect to granularity 1.

4.6 *Burst* applications

In this section, the overall performance of the BCM is evaluated on three real-world *bursts*: hyperparameter tuning, PageRank, and TeraSort. These applications clearly illustrate the three friction points and provide a comprehensive comparison between *burst computing* and FaaS-based approaches.

4.6.1 Hyperparameter tuning

Hyperparameter tuning is a common task in machine learning (ML) that involves evaluating different combinations of hyperparameter values to find the best-performing set for a given ML model. This task, particularly using a grid search technique, is inherently parallel, as each worker evaluates a different set of hyperparameters independently using the same training dataset. While FaaS can execute these parallel tasks, it faces significant inefficiencies due to redundant data downloads by each worker, even if multiple workers are co-located on the same invoker. Consequently, memory and network bandwidth are wasted, leading to increased latency and higher costs. *Burst computing* addresses this inefficiency by leveraging worker locality to optimize data download and resource utilization.

Setup

The hyperparameter tuning experiment is conducted using a grid search technique on a stochastic gradient descent (SGD) model. The model is implemented in a Python application using the `scikit-learn` (`sklearn`) library. The dataset used for this evaluation consists

Table 4.3 Time to start 96 workers and gather input data (ready for computation) in hyperparameter tuning for different burst granularity. Data download time is also shown for comparison.

Granularity	**1 (FaaS)**	**6**	**12**	**24**	**48**	**96**
Download time (s)	13.97	2.97	1.51	1.29	1.19	1.06
Ready time (s)	17.51	5.65	3.64	3.18	2.96	2.57

of 500 MiB of Amazon reviews, available on Kaggle[1], and stored in an S3 bucket. The baseline for comparison is AWS Lambda (granularity 1), with a memory configuration of 1769 MiB, which provides a full vCPU. As for the *burst computing* model, a `c7i.24xlarge` VM (96 vCPUs, 192 GiB RAM and 37.5 Gbps network bandwidth) is used.

Results and analysis

The results, summarized in table 4.3, demonstrate a clear reduction in the "Ready time" as the packing granularity increases. "Ready time" is defined as the total time from the initiation of the job (client-side invocation) to the point where all workers have downloaded the input data and are ready to start the computation. "Download time" is also shown for comparison, which is the time taken to download the input data from S3.

There are two primary factors contributing to the reduction in "Ready time" with *burst computing*:

1. **Group invocation primitive:** This primitive significantly speeds up the invocation time compared to FaaS. For example, the invocation time drops from around 4 s for FaaS to approximately 1.5 s with a granularity of 96.

2. **Data download optimization:** Unlike FaaS, where each worker downloads a separate copy of the data, *burst computing* enables workers co-located within the same pack to download the input data in parallel. This collaborative downloading dramatically reduces the input download time. In the case of FaaS, the download time is around 14 s, whereas with a granularity of 48 or 96, it reduces to about 1 s.

4.6.2 PageRank

PageRank is a prominent data analysis algorithm characterized by intensive worker coordination. It operates iteratively, performing extensive data aggregation across large datasets, making it an excellent candidate for benchmarking the communication capabilities of BCM. Specifically, the aim is to compare the performance of isolated workers in the FaaS model with the worker packing and locality features of the *burst* model in handling aggregations and iterative algorithms. The PageRank implementation used for this evaluation is adapted

[1] https://www.kaggle.com/bittlingmayer/amazonreviews

from the MapReduce version in the Hi-Bench benchmark suite [24]. The simplified version of the algorithm is shown in listing 1.

The MapReduce version of the PageRank algorithm on top of FaaS is not feasible as the high number of short iterations (stages) needed for the algorithm to converge makes it obviously slower than the *burst computing* model. Therefore, the comparison is centred around the *burst computing* model and various granularities.

Setup

For this experiment, four `c7i.16xlarge` VMs (64 vCPUs, 128 GiB RAM and 25 Gbps network bandwidth) are used to launch *bursts* of different sizes. The graph dataset, generated with Hi-Bench, consists of 50 million nodes, totalling approximately 30 GiB and is partitioned into 256 segments. The PageRank algorithm is executed for 10 iterations with a *burst* size of 256, and the granularity is varied from 1 to 64.

Results and analysis

The results illustrated in fig. 4.10 display the total execution of all iterations, segmented into three phases: 1. downloading input data from S3, 2. computing the ranking, and 3. inter-worker communication (collectives). The times for each phase are averaged across all workers and summed across all iterations.

Additionally, table 4.4 provides data on network traffic and the percentage reduction compared to granularity 1.

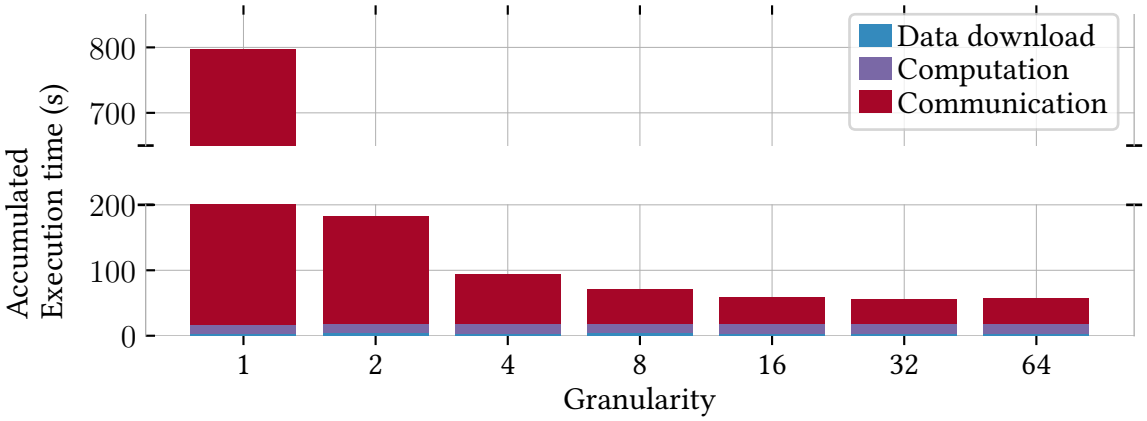


Fig. 4.10 Average aggregated time of each phase in PageRank for different granularities.

The results indicate that communication constitutes the majority of the execution time. This is due to the necessity of aggregating and sharing the rank vector at each iteration. In this configuration, the rank vector, which is 40 MiB in size, is sent, received, and aggregated in a tree pattern among the workers. After aggregation, it is broadcasted from the root worker to all other workers.

Table 4.4 Aggregated network traffic volume and percentage of traffic reduction compared to granularity 1, for different granularities in PageRank.

Granularity	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Traffic (GiB)	3068	1532	764	380	188	92	44
% Reduction	n/a	50.0%	75.0%	87.6%	93.8%	97.0%	98.5%

As granularity increases, the portion of communication that occurs remotely decreases. For instance, with a granularity of 2, only the first level of the binary reduction tree is local, and subsequent levels involve remote communication. With a granularity of 64, there are four packs, so remote communication is limited to the last two levels of the tree.

This optimization leads to a significant reduction in data traffic and execution time. Specifically, with a granularity of 64, the configuration achieves a 98.5% reduction in data traffic and a 13× speed-up in execution time compared to a granularity of 1.

Additionally, a detailed breakdown of the PageRank execution time for different granularity 4 and 32 is shown in fig. 4.11. The first three iterations are analyzed to highlight the impact of the packing granularity on the execution phases.

4.6.3 TeraSort

TeraSort is a benchmark workload adapted from the MapReduce model found in the HiBench suite [24], specifically chosen for its intensive data shuffle phase. The goal of this evaluation is to compare the performance of the traditional serverless MapReduce approach with a *burst computing* version, which leverages locality for the shuffle phase. The key advantages of *burst computing* in this context are two-fold:

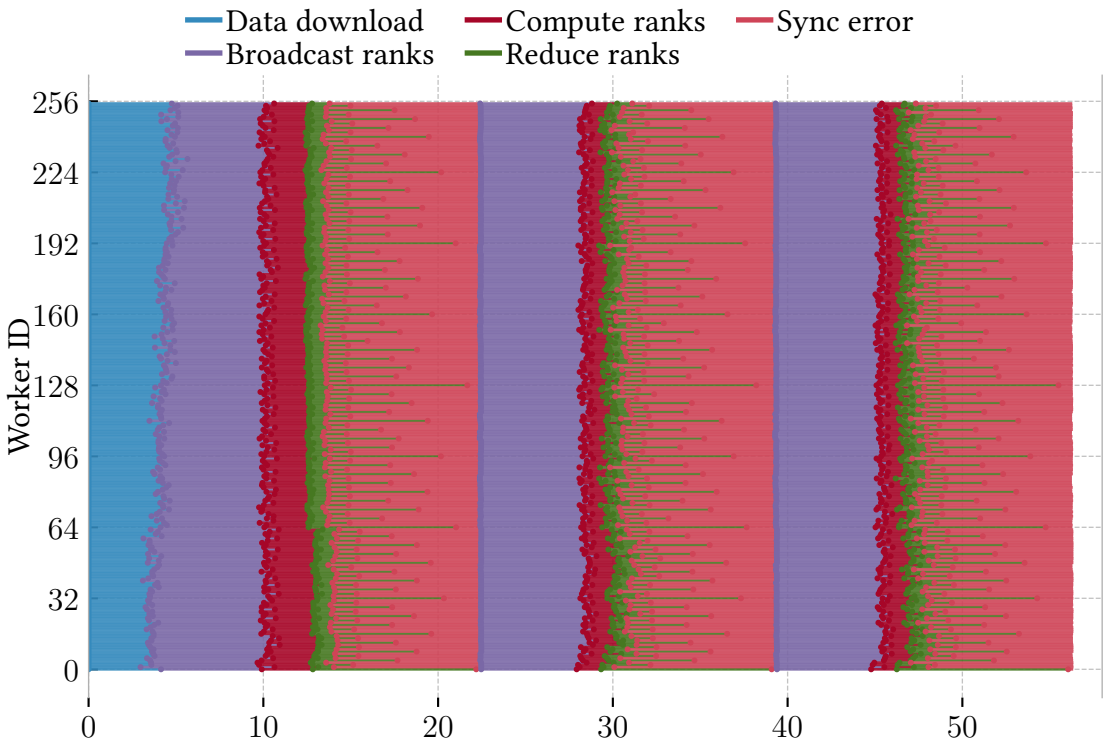
1. **No phase separation:** In the traditional MapReduce model, two rounds of function invocation are required for the `map` and `reduce` phases. In contrast, *burst computing* completes the task with a single `flare` invocation, eliminating the need for phase separation.

2. **Local shuffle:** The shuffle phase in MapReduce involves exchanging data between workers through object storage, whereas *burst computing* utilises the BCM's locality-aware *all-to-all* collective for data exchange.

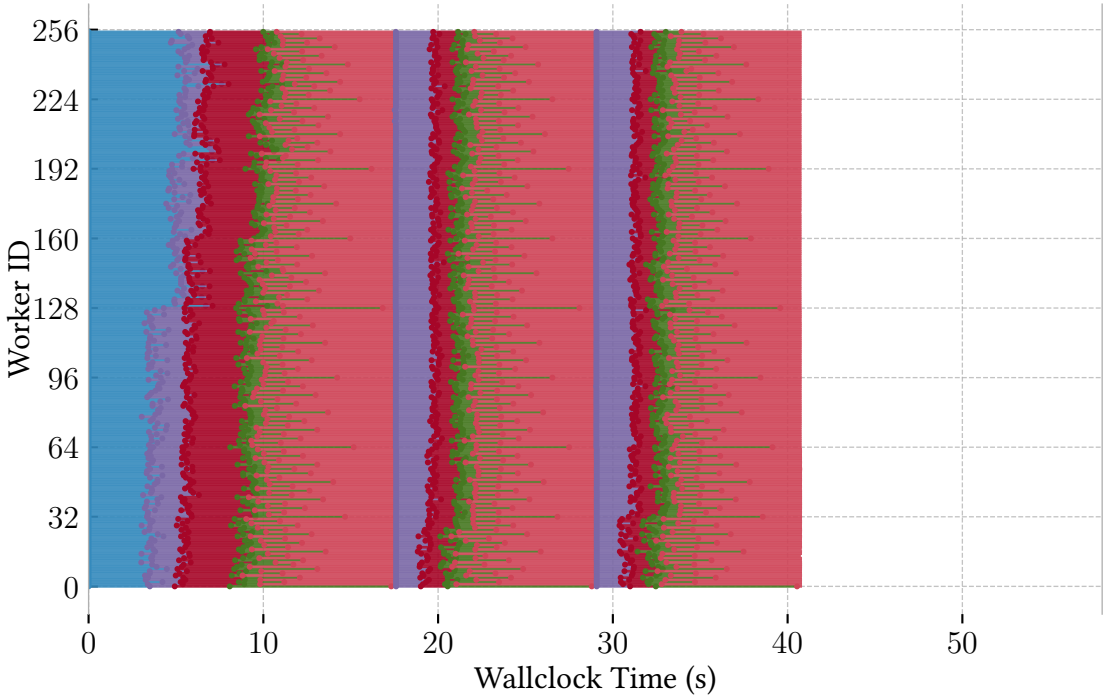
The TeraSort implementation used in this evaluation is adapted from the Hi-Bench suite and is written in Rust. It uses the bucket sort algorithm to sort the input data. The simplified version of the algorithm is shown in listing 3.

Setup

In this experiment, a 100 GiB dataset, generated with Hi-Bench, is partitioned into 192 segments and sorted. The *burst computing* platform is deployed on Amazon EKS, utilizing two `c7i.48xlarge` VMs (192 vCPUs, 384 GiB RAM and 50 Gbps network bandwidth) as



(a) Granularity 4



(b) Granularity 32

Fig. 4.11 PageRank execution time breakdown for granularities 4 (fig. 4.11a) and 32 (fig. 4.11b. The first 3 iterations are shown in detail.

the invokers and a `c7i.xlarge` VM (4 vCPUs, 8 GiB RAM and up to 12.5 Gbps network bandwidth) as the controller. The input data is stored in an Amazon S3 bucket.

Listing 3 Simplified source code of the TeraSort *work* function for *burst computing*, written in Rust. The access to *burst context* to perform the *all-to-all* collective is highlighted.

```rust
fn work(params: Input, burst: &BurstContext) -> Output {
    let num_partitions = params.num_partitions;
    let partition_idx = params.partition_idx;
    let sort_column = params.sort_column
    let segment_bounds: Vec<String> = params.segment_bounds;

    let chunk = get_chunk(&params);
    let df = DataFrame::from_chunk(&chunk);

    // Calculate indexes for each partition using binary search
    let mut indexes: HashMap<u32, Vec<u32>> = HashMap::new();
    for (idx, value) in df[sort_column].iter().enumerate() {
        let to = match segment_bounds.binary_search(&value) {
            Ok(x) => x,
            Err(x) => x,
        };
        indexes.to(to).or_default().push(idx);
    }

    let mut exchange_vec = vec![Bytes::new(); num_partitions];
    for (bucket, idxs) in indexes {
        // Get the rows that belong to this bucket taking the indexes calculated before
        let mut partition_df = df.take(&idxs);
        exchange_vec[bucket] = serialize(&partition_df);
    }

    let exchanged_vec = burst.all_to_all(exchange_vec)
    let mut exchanged_df = deserialize(&exchanged_vec);

    exchanged_df.sort_by(sort_column);

    upload_chunk(&exchanged_df, partition_idx);

    Output { partition_idx }
}
```

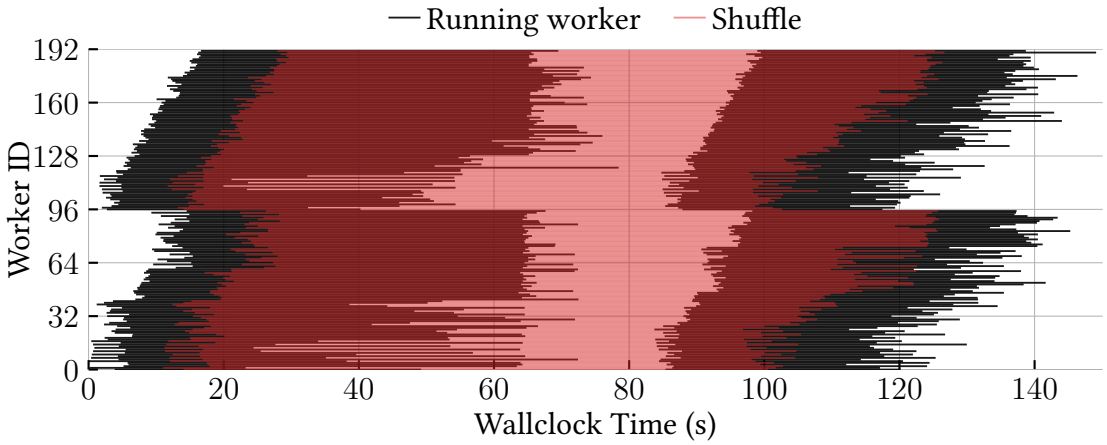
Results and analysis

The results of the TeraSort benchmark are displayed in fig. 4.12, comparing the timelines of the traditional serverless MapReduce approach with the *burst computing* model. The execution times of individual workers are represented by horizontal black bars, with *Worker IDs* along the vertical axis. The time taken for the shuffle phase is superimposed on the timeline in red, highlighting the difference in data exchange between the two approaches.

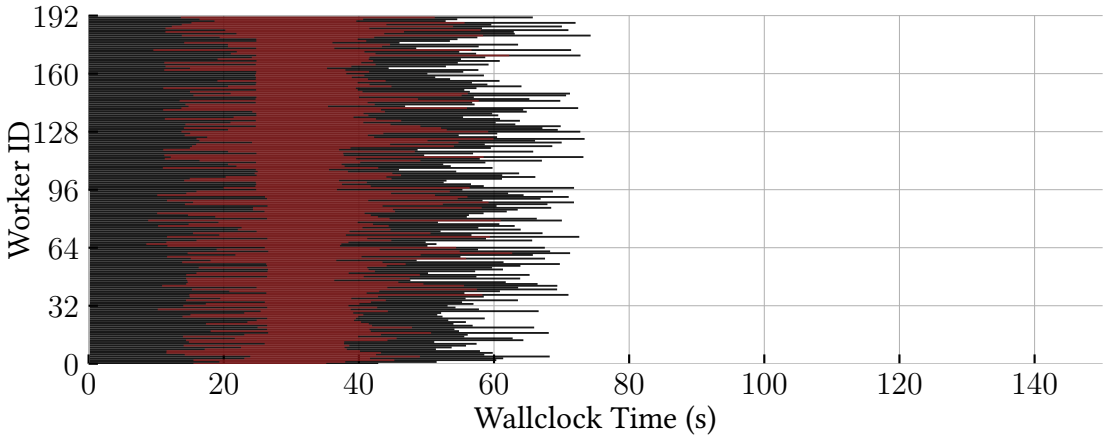
4.6.4 Takeaway

In the MapReduce version (fig. 4.12a), several inefficiencies are evident:

1. **Worker initialization:** There is significant variability in function start-up times, reflecting the issues previously shown in fig. 4.2.



(a) MapReduce



(b) *Burst*

Fig. 4.12 TeraSort execution timeline comparison between fig. 4.12a serverless MapReduce and fig. 4.12b *burst computing*. MapReduce version has two stages (`map` and `reduce`) while *burst* uses a single `flare`, exchanging data with the *all-to-all* collective.

2. **Synchronization:** A noticeable gap exists where no functions are running, caused by the need to synchronize between the `map` and `reduce` phases, which introduces additional latency.

3. **Outliers:** An outlier in the map phase (worker #121) significantly delays the entire workflow, highlighting the impact of stragglers and slow-starting functions.

These issues are effectively mitigated in the *burst computing* model (fig. 4.12b):

- The group invocation mechanism packs all workers into two containers of 96 workers each, leading to faster start-up times and ensuring simultaneous execution (parallelism), thus eliminating latency-induced outliers.

- Worker-to-worker collectives allow the workload to be processed in a single phase, avoiding the overheads of phase separation and synchronization.

- Locality-aware communication further reduces remote data transfer, as discussed in section 4.5.

This results in a substantial speed-up, with a particular execution achieving nearly a 2× improvement over the traditional MapReduce model, and an average speed-up of 1.91× across six runs.

The evaluation of real-world applications like hyperparameter tuning, PageRank, and TeraSort highlights how the three friction points explained in section 2.1.4 are addressed by *burst computing*:

1. Hyperparameter tuning exemplifies the duplication in worker initialization due to friction **F1**, which also slows down worker start-up in both PageRank and TeraSort.

2. The iterative nature of PageRank exposes friction **F2**, making it unfeasible in the FaaS model due to excessive stages, while TeraSort's performance is hindered by slower coordination. The *burst computing* model alleviates these issues by enabling workers to coordinate and share data in a single stage, rather than relying on multiple externally orchestrated stages.

3. Furthermore, both PageRank and TeraSort illustrate friction **F3**, with PageRank requiring extensive communication for vector aggregation and TeraSort necessitating a large-scale data shuffle. In both cases, *burst computing* significantly reduces these communication overheads through locality-aware data transfer, enhancing overall performance and efficiency.

Chapter 5

Related work

Several research efforts have focused on improving the performance of serverless computing platforms by addressing the challenges associated with communication. This section provides an overview of related work in the areas of communication middleware and collective communication in FaaS environments.

For instance, Jin et al. [26] proposed **Ditto**, a job scheduler for serverless analytics. Ditto prioritizes the placement of stages of functions with large shuffling traffic to leverage zero-copy intra-server communication for efficient shuffling. However, Ditto does not provide group-awareness or collective communication mechanisms, which are essential for burst-parallel jobs.

Communication and state sharing within FaaS platforms have been extensively explored in the literature. **Cloudburst** [44] reimagines FaaS platforms by introducing stateful functions that provide low-latency mutable state and communication. It uses an autoscaling key-value store for state sharing and benefits from data locality by co-locating caches alongside function executors. Shillaker and Pietzuch [43] introduced **Faasm**, a serverless runtime that supports sharing memory directly between functions to reduce data movement costs.

Both Cloudburst and Faasm focus on including shared data spaces where functions can manage common state collaboratively. Nonetheless, these solutions, while effective in their own right, do not directly address the communication challenges inherent to burst-parallel workloads. They operate orthogonally to the *burst communication middleware* (BCM) proposed in this thesis.

Additionally, other studies explore dedicated remote storage solutions to address state sharing. These include:

- **Pocket**: an elastic ephemeral storage system designed to meet the demands of serverless analytics applications [30].
- **Crucial**: a distributed shared memory system that enables state management and coordination in serverless applications [8].
- **Glider**: an ephemeral storage system that improves data movement in serverless analytics [6].

Specific works focused on communication mechanisms include **Boxer** by Wawrzoniak et al. [48] and **FMI** by Copik et al. [11]. Boxer implements function-to-function communication using conventional TCP/IP, enabling efficient data processing on serverless platforms. FMI, on the other hand, provides a high-performance library of point-to-point and collective communication operations for groups of FaaS functions, including techniques like NAT traversal.

These efforts, nevertheless, are constrained by the limitations of current FaaS platforms and do not leverage locality optimizations as effectively as the *burst communication middleware* (BCM) proposed in this thesis. Moreover, FMI could potentially be integrated into the BCM as a remote communication backend to expedite *pack-to-pack* data transfers, further enhancing communication efficiency.

Finally, HPC technologies directly relate to the *burst communication middleware* (BCM). **MPI** offers a rich set of collective communication operations to facilitate the execution of SPMD programs on distributed systems. However, the static membership of MPI groups hinders rapid auto-scaling, which is crucial for *burst* workloads that require quick scaling and dynamic resource allocation. While existing HPC approaches can inspire burst computing solutions, new runtimes are necessary to support the rapid setup and communication required for burst-parallel jobs. The key lies in overcoming the overhead associated with establishing **direct communications**, which can significantly impact the computation time in *burst* scenarios.

In conclusion, while various studies have contributed to enhancing communication, state sharing, and resource utilization in FaaS platforms, they fall short in addressing the specific needs of burst-parallel workloads. The *Burst Communication Middleware* (BCM) aims to fill this gap by leveraging locality to optimize inter-worker communication, offering a promising communication solution in cloud environments.

Chapter 6

Conclusions and future work

In this thesis, the design and implementation of the *Burst Communication Middleware* (BCM) has been introduced as a core component of the *burst computing* model, aimed at addressing the increased demand for handling sudden, burst-parallel workloads. This work thoroughly examined the limitations and challenges of current Function-as-a-Service (FaaS) technologies and demonstrated the effectiveness and versatility of the BCM in optimizing communication and data sharing among workers in a serverless environment. The design principles, architecture, and implementation details of BCM, highlighting its role in enhancing communication efficiency and reducing overheads in burst-parallel jobs have been extensively discussed.

Experiments and performance evaluations presented in this thesis underscore the considerable benefits of the BCM. By leveraging locality, the middleware achieves significant improvements in worker-to-worker communication through group collectives. The practical efficacy of the BCM has been validated in real-world scenarios, with performance evaluations demonstrating speed-ups of 13× for PageRank and 2× for TeraSort.

The complexity and sophistication of this thesis are further emphasized by the broad range of advanced technologies, methodologies, and tools employed throughout the implementation and evaluation of the BCM. The development involved using Rust, a modern systems programming language, as the primary implementation language for the BCM. Rust is known for its performance and safety features, however, it also presents a steep learning curve and challenges in terms of concurrency and memory management. The project also integrated several communication backends, which required a deep understanding of their respective APIs and mechanisms. Overall, a thorough comprehension of containerization, cloud, and serverless technologies such as Docker, Kubernetes and AWS Lambda was essential for the success of this work.

The thesis is part of a larger research effort on *burst computing*, encapsulated in a paper titled *"Burst Computing: Quick, Sudden, Massively Parallel Processing on Serverless Resources"* [7]. This research article has been submitted to the 2025 Proceedings of the European Conference on Computer Systems (EuroSys), and the BCM is a central component of the proposed *burst computing* model.

While Aitor Arjona provided guidance and direction for this work, the development of the BCM was conducted independently by me, including key design decisions and the overall architecture. Specifically, chapter 2 draws from the broader paper on *burst computing*, while chapter 3 details my original contributions and the technical implementation of the BCM. This independent work highlights the depth of my involvement and the technical challenges addressed in this thesis.

Future work

Several promising avenues for future research and development in the Burst Communication Middleware are evident. One significant area for exploration is the refinement and enhancement of the BCM itself. Future work could focus on optimizing the BCM's internal mechanisms to further reduce communication overheads and improve performance.

Another potential direction for future research is the implementation of additional communication primitives in the BCM. While the current implementation of BCM provides basic message passing and group collective operations, the addition of more advanced communication operations could further enhance the middleware's capabilities. For example, common collective operations such as *scatter* or *gather* could be implemented to facilitate data exchange among workers in a burst-parallel job.

Furthermore, the BCM could be extended by integrating support for additional communication backends. Currently, the BCM has support for Redis, RabbitMQ, and S3 as communication backends. Future work could explore the integration of other communication systems, especially stream-oriented, such as Kafka or NATS, and evaluate their performance in the context of burst-parallel workloads.

References

[1] Lixiang Ao et al. “Sprocket: A Serverless Video Processing Framework”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 263–274. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267815. URL: https://dl.acm.org/doi/10.1145/3267809.3267815 (visited on 07/03/2023).

[2] async-book. *Why Async? - Asynchronous Programming in Rust*. URL: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html.

[3] AWS. *New for AWS Lambda – 1ms Billing Granularity Adds Cost Savings*. 2020. URL: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/.

[4] AWS. *Serverless Computing – AWS Lambda Pricing – Amazon Web Services*. URL: https://aws.amazon.com/lambda/pricing/.

[5] Daniel Barcelona-Pons and Pedro García-López. “Benchmarking Parallelism in FaaS Platforms”. In: *Future Generation Computer Systems* 124 (Oct. 2020), pp. 268–284. DOI: 10.1016/j.future.2021.06.005.

[6] Daniel Barcelona-Pons, Pedro García-López, and Bernard Metzler. “Glider: Serverless Ephemeral Stateful Near-Data Computation”. In: *Proceedings of the 24th International Middleware Conference*. Middleware ’23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 247–260. ISBN: 9798400701771. DOI: 10.1145/3590140.3629119. URL: https://doi.org/10.1145/3590140.3629119.

[7] Daniel Barcelona-Pons et al. “Burst Computing: Quick, Sudden, Massively Parallel Processing on Serverless Resources”. In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys ’25. Rotterdam, Netherlands, 2025.

[8] Daniel Barcelona-Pons et al. “On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures”. In: *Proceedings of the 20th International Middleware Conference*. Middleware ’19. Davis, CA, USA: Association for Computing Machinery, Dec. 2019, pp. 41–54. ISBN: 978-1-4503-7009-7. DOI: 10.1145/3361525.3361535. URL: https://doi.org/10.1145/3361525.3361535.

[9] Benjamin Carver et al. “Wukong: a scalable and locality-enhanced framework for serverless parallel computing”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1–15. ISBN: 9781450381376. DOI: 10.1145/3419111.3421286. URL: https://doi.org/10.1145/3419111.3421286.

[10] Cloudflare. *What is Function-as-a-Service (FaaS)? | Cloudflare*. URL: https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/.

[11] Marcin Copik et al. “FMI: Fast and Cheap Message Passing for Serverless Functions”. In: *Proceedings of the 37th International Conference on Supercomputing*. ICS ’23. Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 373–385. ISBN: 9798400700569. DOI: 10.1145/3577193.3593718. URL: https://doi.org/10.1145/3577193.3593718.

[12] Tokio documentation. *Bridging with sync code | Tokio - An asynchronous Rust runtime*. URL: https://tokio.rs/tokio/topics/bridging.

[13] Tokio documentation. *channel in tokio::sync::broadcast - Rust*. URL: https://docs.rs/tokio/latest/tokio/sync/broadcast/fn.channel.html.

[14] Tokio documentation. *channel in tokio::sync::mpsc - Rust*. URL: https://docs.rs/tokio/latest/tokio/sync/mpsc/fn.channel.html.

[15] Tokio documentation. *Semaphore in tokio::sync - Rust*. URL: https://docs.rs/tokio/latest/tokio/sync/struct.Semaphore.html#rate-limiting-using-a-token-bucket.

[16] Tokio documentation. *tokio - Rust*. URL: https://docs.rs/tokio/latest/tokio/.

[17] Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[18] Sadjad Fouladi et al. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[19] Apache Software Foundation. *Apache OpenWhisk Documentation*. 2023. URL: https://openwhisk.apache.org.

[20] Apache Software Foundation. *apache/openwhisk-runtime-rust: Apache OpenWhisk Runtime Rust supports Apache OpenWhisk functions written in Rust*. URL: https://github.com/apache/openwhisk-runtime-rust.

[21] Apache Software Foundation. *apache/openwhisk: Apache OpenWhisk is an open source serverless cloud platform*. URL: https://github.com/apache/openwhisk.

[22] Refactoring Guru. *Proxy*. URL: https://refactoring.guru/design-patterns/proxy.

[23] Joseph M. Hellerstein et al. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. DOI: 10.48550/ARXIV.1812.03651. URL: https://arxiv.org/abs/1812.03651.

[24] Shengsheng Huang et al. "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis". In: *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 2010, pp. 41–51. DOI: 10.1109/ICDEW.2010.5452747.

[25] IBM. *What Is Function as a Service (FaaS)? | IBM*. URL: https://www.ibm.com/topics/faas.

[26] Chao Jin et al. "Ditto: Efficient Serverless Analytics with Elastic Parallelism". In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 406–419. ISBN: 9798400702365. DOI: 10.1145/3603269.3604816. URL: https://doi.org/10.1145/3603269.3604816.

[27] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019.

[28] Eric Jonas et al. "Occupy the cloud: distributed computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. New York, NY, USA: Association for Computing Machinery, Sept. 2017, pp. 445–451. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128601. URL: https://dl.acm.org/doi/10.1145/3127479.3128601 (visited on 05/02/2023).

[29] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. "Centralized Core-granular Scheduling for Serverless Functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 158–164. ISBN: 978-1-4503-6973-2. DOI: 10.1145/3357223.3362709. URL: https://doi.org/10.1145/3357223.3362709 (visited on 07/03/2023).

[30] Ana Klimovic et al. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[31] Ana Klimovic et al. "Understanding Ephemeral Storage for Serverless Analytics". en. In: 2018, pp. 789–794. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless (visited on 05/05/2023).

[32] Yilong Li, Seo Jin Park, and John Ousterhout. "MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds". en. In: 2021, pp. 593–611. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/li-yilong (visited on 07/03/2023).

[33] Fangming Lu et al. "Serialization/Deserialization-free State Transfer in Serverless Workflows". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. EuroSys '24. Athens, Greece: Association for Computing Machinery, 2024, pp. 132–147. ISBN: 9798400704376. DOI: 10.1145/3627703.3629568. URL: https://doi.org/10.1145/3627703.3629568.

[34] Ingo Müller et al. "Serverless Clusters: The Missing Piece for Interactive Batch Applications?" en. In: Apr. 2020. DOI: 10.3929/ethz-b-000405616. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/405616 (visited on 07/03/2023).

[35] Gerard París, Pedro García-López, and Marc Sánchez-Artigas. "Serverless Elastic Exploration of Unbalanced Algorithms". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. ISSN: 2159-6190. Oct. 2020, pp. 149–157. DOI: 10.1109/CLOUD49709.2020.00033.

[36] Wei Qiu et al. "User-guided Page Merging for Memory Deduplication in Serverless Systems". In: *2023 IEEE International Conference on Big Data (BigData)*. 2023, pp. 159–169. DOI: 10.1109/BigData59044.2023.10386487.

[37] Rust Package Registry. *aws-sdk-rust - crates.io: Rust Package Registry*. URL: https://crates.io/crates/aws-sdk-rust.

[38] Rust Package Registry. *lapin - crates.io: Rust Package Registry*. URL: https://crates.io/crates/lapin.

[39] Rust Package Registry. *redis - crates.io: Rust Package Registry*. URL: https://crates.io/crates/redis.

[40] Alice Ryhl. *Actors with Tokio – Alice Ryhl*. URL: https://ryhl.io/blog/actors-with-tokio/.

[41] Josep Sampé et al. "Serverless Data Analytics in the IBM Cloud". In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, pp. 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[42] Brandon Scheller. *Best practices for resizing and automatic scaling in Amazon EMR*. 2018. URL: https://aws.amazon.com/blogs/big-data/best-practices-for-resizing-and-automatic-scaling-in-amazon-emr/.

[43] Simon Shillaker and Peter Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". en. In: 2020, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker (visited on 07/03/2023).

[44] Vikram Sreekanti et al. “Cloudburst: stateful functions-as-a-service”. In: *Proceedings of the VLDB Endowment* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836. URL: https://doi.org/10.14778/3407790.3407836 (visited on 07/03/2023).

[45] Jovan Stojkovic et al. “MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA ’23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589069. URL: https://doi.org/10.1145/3579371.3589069.

[46] Ali Tariq et al. “Sequoia: enabling quality-of-service in serverless computing”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 311–327. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421306. URL: https://dl.acm.org/doi/10.1145/3419111.3421306 (visited on 07/03/2023).

[47] Rust Team. *Rust Programming Language*. URL: https://www.rust-lang.org/.

[48] Mike Wawrzoniak et al. “Boxer: Data Analytics on Network-enabled Serverless Platforms”. en. In: Jan. 2021. DOI: 10.3929/ethz-b-000456492. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/456492 (visited on 07/03/2023).

[49] Sebastian Werner and Stefan Tai. “A reference architecture for serverless big data processing”. In: *Future Generation Computer Systems* (2024). ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2024.01.029. URL: https://www.sciencedirect.com/science/article/pii/S0167739X24000360.