UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA


# Improving Parallel and Concurrent Programming in FreeST

Guilherme João Correia Lopes

**Mestrado em Engenharia Informática**

Versão Provisória

Dissertação orientada por:
Prof.ª Doutora Andreia Filipa Torcato Mordido Rodrigues
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2024

# Acknowledgments

I would like to express my gratitude to my supervisors, Andreia Mordido and Vasco T. Vasconcelos, for their invaluable guidance, understanding, and insightful contributions throughout this work. Their expertise and patience were pivotal in shaping this work and ensuring its successful completion.

I am profoundly grateful to my parents for their unwavering support and encouragement, which sustained me through both the highs and lows of this journey. To my siblings, thank you for lifting my spirits and providing much-needed distractions. I am forever indebted to my grandmother and aunt, whose enduring presence profoundly shaped my education and life. A special thanks to my girlfriend, whose constant support, motivation, and unwavering belief in me were invaluable; your presence ensured that my resolve never wavered. I also extend my heartfelt thanks to all my friends who consistently believed in me and provided their support.

Without the collective support of each of you, this work would not have been possible. Thank you all.

# Resumo

Nos últimos anos, a evolução tecnológica levou os fabricantes a adotar arquiteturas de processadores multi-core, promovendo o crescimento da *computação paralela e concorrente*. Este avanço trouxe inúmeros benefícios, incluindo melhorias no desempenho e na comunicação entre processos em sistemas distribuídos. No entanto, essa transição também introduziu desafios significativos, exigindo ferramentas e linguagens especializadas que possam lidar eficientemente com o paralelismo e a concorrência. Linguagens como Erlang e Go foram criadas especificamente para atender a essas necessidades, enquanto outras, como Java, evoluíram para incluir funcionalidades que facilitam a programação paralela e concorrente, fornecendo soluções práticas para novos paradigmas de programação.

Assim sendo, as linguagens de programação evoluíram lado a lado com a computação paralela e concorrente, incorporando ferramentas essenciais como módulos e frameworks para abordar diversos paradigmas de programação. Essas ferramentas permitem que programadores implementem soluções eficientes—aproveitando arquiteturas modernas de multiprocessadores—e simplificam o processo de desenvolvimento.

*FreeST* [3, 11] é uma linguagem de programação funcional concorrente baseada em troca de mensagens que implementa *tipos de sessão livres de contexto* [50]. *Tipos de sessão* [32, 33, 34] descrevem padrões de comunicação estruturados entre agentes em sistemas distribuídos, assegurando que todos os agentes envolvidos seguem rigorosamente o protocolo de comunicação definido. Apesar das suas capacidades promissoras, FreeST é uma linguagem académica num estágio embrionário do seu desenvolvimento. A sua aplicabilidade para cenários práticos do mundo real permanece limitada, pois oferece apenas primitivas básicas para programação concorrente e carece de ferramentas abrangentes que auxiliem no desenvolvimento de sistemas paralelos e concorrentes complexos. Consequentemente, FreeST exige um profundo conhecimento da linguagem e um esforço significativo para desenvolver sistemas paralelos e concorrentes de forma eficaz, resultando numa curva de aprendizagem acentuada que coloca desafios até mesmo para programadores familiarizados com a linguagem.

Esta tese visa abordar essas limitações, adaptando a linguaguem FreeST para a tornar mais viável em cenários práticos. Para isso, propomos expandir a linguagem através do desenvolvimento de novos módulos que integram alguns paradigmas emergentes na programação paralela e concorrente. Especificamente, são introduzidos três módulos principais: o módulo *Parallel*, o módulo *Futures* e o módulo *Streams*.

O módulo Parallel foi projetado para abordar o *paralelismo de dados* [37], fornecendo um ambiente de programação com um conjunto de abstrações que facilitam a implementação de *problemas embaraçosamente paralelos* [27]. Estas abstrações implementam padrões de comunicação baseados nas operações e mecanismos definidos pela especificação de *Message-Passing Interface* (MPI) [43, 36]. Este módulo permite que os desenvolvedores lidem de maneira mais eficiente com problemas que se beneficiam de divisões de trabalho triviais, melhorando a experiência de programação através de um ambiente conciso e eficiente. Para ilustrar a sua aplicação prática, comparamos a implementação do método de *Monte Carlo* [45] para estimar $\pi$ em FreeST e a implementação usando este módulo. Este exemplo demonstra as vantagens deste módulo em lidar com paralelismo de dados e problemas embaraçosamente paralelos de forma eficiente.

Em FreeST o resultado de computações assíncronas é descartado. Para recuperar o resultado, é necessário estabelecer um canal entre a computação assíncrona e a thread principal. O módulo Futures foi desenvolvido para abstrair essa complexidade do gerenciamento de computações assíncronas, permitindo que os desenvolvedores criem e manipulem *futures* [1, 14], que representam resultados de computações realizadas de forma assíncrona. Isso simplifica a recuperação dos resultados dessas computações e facilita a abordagem de problemas que se beneficiam de estratégias de *divisão e conquista* [21] concorrentes, de forma semelhante ao modelo *ForkJoin* [20, 39, 40]. Nesta tese, implementamos o exemplo da *sequência de Fibonacci* [21] de forma a detalhar como este módulo pode ser usado para implementar algoritmos de divisão e conquista de maneira mais intuitiva e eficiente.

Embora FreeST seja adequado para programação com *streams* [10, 17] devido à natureza dos tipos de sessão, a linguagem carece de um conjunto de ferramentas que abstraiam e simplifiquem esse paradigma de programação. O módulo Streams foi desenvolvido para suprir essa necessidade, fornecendo abstrações e alguns mecanismos baseados na linguagem *StreamIt* [51, 9] que incentivam práticas de programação mais estruturadas e bem definidas. Isso resulta em código menos convoluto especialmente em cenários mais complexos. A tese explora a implementação do algoritmo *quicksort* [30, 21] utilizando o módulo Streams, demonstrando as suas vantagens e pontencialidades, e faz uma comparação entre o módulo e a linguagem StreamIt.

A validação destas contribuições foi realizada por uma avalição através da condução de três questionários—um por cada módulo proposto—e por uma análise da redução de *linhas de código* (LoC) pela utilização do módulo Parallel. Em cada questionário, os participantes realizaram um exercício a utilizar o respetivo módulo, avaliaram alguns parâmetros em uma escala de 1 a 10, e forneceram feedback mais detalhado através de respostas abertas. Apesar de identificarem várias limitações dentro de cada módulo, principalmente devido a restrições inerentes ao FreeST, os questionários receberam feedback muito positivo. Isso indica uma recepção favorável e valida a utilidade e a eficácia das contribuições realizadas.

A comparação de LoC entre implementações a usar apenas FreeST "puro" e implementações a usar o módulo Parallel, mostra uma redução substancial na complexidade e quantidade de código necessário, evidenciando a sua eficácia em simplificar a abordagem a paralelismo de dados e em

implementar problemas embaraçosamente paralelos.

Em termos de trabalho futuro, sugerimos várias direções para continuar a expandir e refinar o FreeST. Melhorias nos módulos propostos podem ser guiadas pelas limitações identificadas durante o seu desenvolvimento, com o objetivo de criar módulos mais consistentes, flexíveis, e aptos a lidar com uma variedade mais ampla de cenários. Além disso, as percepções dos participantes dos questionários apontam várias oportunidades de aprimoramento, especialmente em termos de usabilidade. Também encorajamos a exploração de outros paradigmas e conceitos de programação para ampliar ainda mais as capacidades de FreeST para programação paralela e concorrente. Exemplos incluem a *Programação Reativa Funcional* (FRP) [44], o *modelo de atores* [29, 28] e a *geração de números pseudoaleatórios* (PRNG) [15].

Em suma, este trabalho contribui para o avanço da linguagem FreeST, tornando-a mais prática e eficiente para programação paralela e concorrente. A introdução de módulos específicos para abordar diferentes paradigmas de programação não apenas expande as capacidades da linguagem, mas também demonstra a sua aplicação prática e viabilidade em resolver problemas no mundo real. As melhorias na linguagem e os novos módulos desenvolvidos representam um passo importante para tornar a FreeST uma opção viável e atraente para desenvolvedores que enfrentam desafios de programação paralela e concorrente.

**Palavras-chave:** Tipos de sessão, Concorrência, Problemas embaraçosamente paralelos, Futuros, Streams

# Abstract

Technological advancements led manufacturers to shift from single-processor to multi-core architectures, fostering the rise of parallel and concurrent computing. This transition has yielded substantial benefits, such as improved performance and enhanced interprocess communication in distributed systems. However, it has also introduced challenges requiring specialized tools and languages to handle parallelism and concurrency efficiently and make these tasks easier for developers. Languages like Erlang and Go were purpose-built to meet these demands, while others, such as Java, have evolved to incorporate features facilitating parallel and concurrent programming, providing accessible solutions for emerging paradigms.

FreeST is a modern, message-passing concurrent functional language featuring context-free session types to ensure strict adherence to communication protocols. While session types were designed for concurrent programming, FreeST, despite its strengths, stumbles upon limitations stemming from its embryonic state and the lack of comprehensive tools for addressing various parallel and concurrent programming paradigms. Consequently, FreeST demands substantial domain knowledge and effort to develop parallel and concurrent systems effectively, posing a steep learning curve. Our goal in this work is to provide a set of tools that improve and ease parallel and concurrent programming in FreeST.

This thesis proposes the development of three modules aimed at enhancing FreeST's capability to handle diverse parallel and concurrent programming paradigms and concepts: data parallelism and embarrassingly parallel problems, futures and divide-and-conquer strategies, and stream programming. These modules are designed to leverage FreeST's unique features and expand its practical utility in real-world applications, making it a more versatile and effective tool for addressing complex parallel and concurrent computing challenges. To validate our contributions, we employ surveys and a Lines of Code (LoC) comparison.

**Keywords:** Session types, Concurrency, Embarrassingly parallel problems, Futures, Streams

# Contents

**List of Figures** **xiii**

**List of Tables** **xv**

**1 Introduction** **1**

**2 Background** **5**

2.1 Terminology and concepts . . . . . 5

2.2 Programming paradigms . . . . . 7

2.2.1 Embarrassingly parallel problems . . . . . 7

2.2.2 Futures and promises . . . . . 8

2.2.3 Divide-and-conquer . . . . . 9

2.2.4 Streams . . . . . 10

2.3 Concurrency in programming languages . . . . . 10

2.4 Related work . . . . . 12

2.4.1 Message-Passing Interface . . . . . 12

2.4.2 ForkJoin . . . . . 17

2.4.3 StreamIt . . . . . 19

**3 The FreeST programming language** **23**

3.1 Session types . . . . . 23

3.2 The math client example . . . . . 27

3.3 Challenges . . . . . 28

**4 Parallel and concurrent modules for FreeST** **31**

4.1 The Parallel module . . . . . 31

4.1.1 Implementing Monte Carlo in FreeST . . . . . 31

4.1.2 Identifying parallel patterns . . . . . 33

4.1.3 Design and implementation . . . . . 34

4.1.4 Implementing Monte Carlo with the module . . . . . 38

4.1.5 Interprocess communication: Challenges and attempts . . . . . 39

4.2 The Futures module . . . . . 40

4.2.1 Design and implementation . . . 41
4.2.2 Addressing divide-and-conquer . . . 42
4.2.3 Implementing the Fibonacci sequence . . . 42
4.2.4 Comparison to ForkJoin and challenges . . . 43
4.3 The Streams module . . . 44
4.3.1 Design and implementation . . . 45
4.3.2 Implementing the quicksort algorithm . . . 48
4.3.3 Comparison to the StreamIt programming language . . . 49

**5 Evaluation** **51**
5.1 Surveys . . . 51
5.1.1 Design . . . 52
5.1.2 Results and analysis . . . 53
5.2 Lines of Code comparison . . . 56

**6 Conclusion and future work** **59**

**References** **64**

**A Modules' implementation code** **65**
A.1 Parallel module . . . 65
A.2 Futures module . . . 69
A.3 Streams module . . . 69

**B FEP-0009: Add new concurrency-related primitives to FreeST** **73**

**C Surveys** **77**
C.1 Parallel module . . . 77
C.2 Futures module . . . 86
C.3 Streams module . . . 92

**D Survey insights** **99**
D.1 Parallel module . . . 99
D.2 Futures module . . . 100
D.3 Streams module . . . 100

# List of Figures


2.1 Sequence diagram of the $\pi$ estimation through the Monte Carlo method . . . . . 8
2.2 MPI - Point-to-point C code example . . . . . 13
2.3 MPI - Illustration of the barrier pattern . . . . . 14
2.4 MPI - Illustration of the broadcast pattern . . . . . 14
2.5 MPI - Illustration of the scatter pattern . . . . . 15
2.6 MPI - Illustration of the gather pattern . . . . . 15
2.7 MPI - Illustration of the reduce pattern . . . . . 15
2.8 MPI - Illustration of the allgather and allreduce MPI patterns . . . . . 15
2.9 ForkJoin - A representation of the *fork* and *join* operations . . . . . 17

3.1 Linear channels - One-to-one representation . . . . . 26
3.2 Shared channels - Many-to-one representation . . . . . 26

4.1 Parallel - Implementation of the Monte Carlo in FreeST . . . . . 32
4.2 Parallel - The manager-workers communication framework . . . . . 36
4.3 Parallel - Illustration of our implementation of the reduce operation . . . . . 37
4.4 Parallel - Implementation of the Monte Carlo using the Parallel module . . . . . 39
4.5 Futures - Divide-and-conquer process relationship in FreeST . . . . . 42
4.6 Streams - Example of the quicksort algorithm . . . . . 48

5.1 Comparison of survey results across all modules . . . . . 54
5.2 LoC comparison for pure FreeST and Parallel module implementations . . . . . 56

# List of Tables

3.1 FreeST - Session type constructors . . . . . 24
3.2 Operations to interact with channels in FreeST . . . . . 27

# Chapter 1

# Introduction

Traditionally, software was written to execute sequentially. However, rapid technological advancements led to physical limitations such as thermal constraints and clock speed, prompting processor manufacturers to pivot from enhancing single-processor performance to developing multi-core architectures. This shift marks the transition from monolithic architectures to *parallel and concurrent computing*, introducing a series of advantages such as increased performance and enhanced interprocess communication in distributed systems.

Communication in distributed systems and parallel and concurrent programming is crucial for connecting the world. It enables different systems and processes to work together seamlessly, facilitating efficient data sharing and task coordination across multiple locations. This interconnectedness not only boosts application performance and supports real-time collaboration but also drives innovation in diverse fields, from global communications to complex scientific computations. Ensuring effective system communication lays the foundation for a more integrated and responsive technological landscape, ultimately bridging gaps and fostering global connectivity.

Effective communication in parallel and concurrent computing is crucial for several reasons. It ensures proper task coordination and synchronization across multiple processors, maintains data consistency, optimizes performance by reducing latency and efficiently utilizing bandwidth, and enhances fault tolerance and reliability by managing errors and maintaining system consistency.

Despite these benefits, parallel and concurrent programming introduces its own set of challenges. The growing demand for that programming paradigm in distributed systems has led to the development of specialized tools and programming languages designed to handle parallelism and concurrency efficiently. Languages such as Erlang [2] and Go [4] were created with these needs in mind while existing languages like Java [6] have evolved to include features that facilitate concurrent programming. The increasing accessibility of these tools has fostered the widespread adoption and establishment of this paradigm in distributed systems.

Moreover, these programming languages rely on *typing systems* and *parallel and concurrent programming models* to minimize the chances of system failure. Regardless of the number of processors in use, it is crucial that a program behaves as expected. A predefined communication protocol must be strictly followed to prevent process cooperation from breaking down, ensuring it will never diverge from its definition.

### Programming paradigms

Programming languages have evolved alongside the advancement of parallel and concurrent computing, integrating essential tools such as modules and frameworks tailored for emerging *programming paradigms*. These tools enable developers to implement efficient solutions across diverse problem domains, leveraging modern multi-processor architectures.

Frameworks addressing programming paradigms have become integral components of programming languages. They empower developers to handle complex computational tasks effectively, enhancing scalability and performance in real-world applications. By integrating these tools, languages not only streamline development processes but also ensure that applications are optimized to meet the challenges posed by concurrent processing.

In essence, modern programming languages have adapted to support robust parallel and concurrent paradigms, crucial for fully utilizing multi-processor systems and managing the complexities of concurrent programming.

### The FreeST programming language

*FreeST* [3, 11] is a modern, message-passing concurrent functional language featuring *context-free session types*, ensuring strict adherence to communication protocols among participants.

*Session types* [32, 33, 34] were proposed to describe structured communication patterns and designed for concurrent settings. With the help of a powerful static type-checker, the communication behaviour of agents in distributed systems can be verified, guaranteeing that all agents involved in a communication strictly follow the protocol. However, protocols allowed by traditional session types are limited. As proposed by Thiemann and Vasconcelos [50], context-free session types solve this limitation by breaking tail recursion and enabling non-regular recursion.

Despite effectively addressing structured communication patterns with session types, FreeST remains an embryonic and academically oriented language with limitations in practical, real-world applications. Currently, FreeST provides only basic primitives for concurrent programming, lacking comprehensive frameworks and tools for diverse programming paradigms that facilitate the development of complex parallel and concurrent systems.

Consequently, FreeST demands substantial domain knowledge and effort to develop parallel and concurrent systems effectively, posing challenges even for programmers familiar with the language. This results in a steep learning curve and substantial effort to establish essential boilerplate code, diverting attention from addressing a problem's logic. For instance, if a developer wants to distribute data among multiple processes to compute an algorithm in parallel, they must establish the interprocess communication framework before concentrating on the algorithm itself. This constraint significantly limits FreeST's viability as a practical choice to address the broader spectrum of challenges in parallel and concurrent programming.

To adapt FreeST for real-world parallel and concurrent programming, it is essential to expand the language, making it more versatile and practical for solving parallel and concurrent problems. This requires integrating modules that address well-known parallel and concurrent programming

paradigms, creating a rich environment that meets programmers' needs. Our goal is to make parallel and concurrent programming in FreeST easier, thus making FreeST a more viable option for real-world scenarios.

In this thesis we aim to adapt well-known parallel and concurrent programming paradigms to FreeST, developing new tools that leverage the language's features. We explore topics such as *data parallelism* [37], *embarrassingly parallel problems* [27], *futures* [1, 14], the *divide-and-conquer paradigm* [21], and *stream programming* [10, 17]. We aim to design coherent and valuable tools by analyzing how FreeST interacts with these concepts, while remaining true to the language's unique characteristics and philosophies.

**Contributions** This work focuses on the design and implementation of three modules, proposing their integration into the FreeST programming language to simplify how programmers approach these concepts and improve their efficiency:

- The *Parallel module* addresses data parallelism and provides a programming environment with a set of abstractions that facilitate the implementation of embarrassingly parallel problems.
- The *Futures module* provides abstractions to launch *asynchronous computations*, equipping FreeST with a novel synchronization method that opens doors to address the divide-and-conquer paradigm.
- The *Streams module* abstracts the inherent concurrent stream programming paradigm in FreeST through a series of definitions and abstractions, promoting more structured and well-defined programming practices.

**Outline** The remainder of this document is organized as follows: Chapter 2 contextualizes this work within relevant terminology, distributed system concepts, and concurrent programming patterns and paradigms. It also includes an overview of some programming languages and related work; Chapter 3 introduces the FreeST programming language, detailing its features, innovations, advantages, and concurrency challenges; Chapter 4 discusses the design, implementation, and usage examples of the developed modules; Chapter 5 describes the validation process for our contributions and presents the respective results; finally, Chapter 6 is reserved for conclusions and suggestions for future research in parallel and concurrent programming within the FreeST programming language.

# Chapter 2

# Background

This chapter provides the necessary foundation for understanding the topics discussed in subsequent chapters. We delve into key parallel and concurrent terminology and concepts, offer an overview of some programming languages featuring concurrent capabilities, and dedicate a section to examining related work that has influenced our design and implementation decisions.

## 2.1 Terminology and concepts

### Parallelism and concurrency

In multicore systems, where a single processing chip contains multiple computing cores, multithreaded programming becomes essential to provide a mechanism for efficiently utilizing these cores and enhancing concurrency.

Two fundamental terms in this context are *parallelism* and *concurrency* [41, 47]. Although often used interchangeably, they are, by definition, distinct concepts.

**Parallelism** Involves leveraging the computational hardware multiplicity, i.e., multiple computing cores, to increase the overall efficiency of a computation. A parallel system is capable of executing multiple tasks simultaneously, so it is possible to achieve computation results earlier by distributing the workload across different processes.

**Concurrency** A program-structuring technique where multiple threads of control execute computations simultaneously instead of sequentially, creating the perception of interleaved results and effects for the user.

### Communication models

Interprocess communication can be managed using several well-known models, with the two most essential being the *shared-memory* and *message-passing* models [47].

**Shared-memory** Processes create and gain access to common regions of memory owned by other processes by agreeing to remove a few memory access restrictions. This model establishes a region of memory shared by the cooperating processes for interprocess communication.

**Message-passing** Processes exchange messages seeking to transfer information between them, coordinate activities, and share data by sending and receiving messages.

The shared-memory model in distributed systems faces synchronization challenges, such as ensuring data consistency and avoiding conflicts in data access. This model is predominant in multiprocessing environments where processes operate on the same machine, enhancing communication efficiency and convenience. Conversely, the message-passing model is efficient for exchanging smaller amounts of data because there is no need to avoid conflicts, and synchronization is easier to manage. However, it entails a higher communication overhead than the shared-memory communication model.

**Communicating Sequential Processes** The message-passing model was significantly popularized by Tony Hoare's *Communicating Sequential Processes* (CSP) [31], a formal language developed in the late 1970s for describing interaction patterns in concurrent systems. CSP introduced a rigorous framework for process synchronization through message exchanges, establishing message-passing as a robust and reliable communication model.

### Synchronization

There are significant differences in synchronizing cooperating processes through the shared-memory and message-passing communication models [27, 47, 49].

In the shared-memory model, multiple processes share a common memory space, requiring synchronization to coordinate accesses and prevent *race conditions*. Race conditions occur when several processes access and manipulate the same data concurrently, influencing the result by the order in which access occurs. Key synchronization mechanisms in shared-memory systems include:

- *Mutual exclusion*: Processes must synchronize access to shared resources to prevent race conditions using locks, semaphores, or other synchronization primitives.
- *Atomic operations*: Provides read-modify-write operations without explicit locking.
- *Deadlock avoidance*: Implementing techniques to prevent or detect scenarios where processes wait indefinitely for each other to release locks.

In the message-passing model, processes synchronize by coordinating the sending and receiving of messages, which can be either *blocking* (synchronous) or *nonblocking* (asynchronous):

- *Synchronous*: Both sending and receiving operations are blocking—the sender blocks until the receiver retrieves the message, and the receiver blocks until a message arrives—ensuring synchronization with every message.
- *Asynchronous*: The send operation is non-blocking, allowing the sender to continue execution without waiting for the receiver's response, while the receive operation can either be blocking or non-blocking. In the non-blocking variant, the receiver continues executing after calling the receive operation, whereas in the blocking variant, the receiver blocks until a message arrives.

## 2.2 Programming paradigms

Regardless of the communication model and type of synchronization, every concurrent programming language addresses certain widely-known programming paradigms to enhance the parallel and concurrent programming experience. Some languages provide specialized tools to handle them, while others adapt through their existing core features.

### 2.2.1 Embarrassingly parallel problems

An *embarrassingly parallel problem* [27] where minimal effort is required to separate the problem into multiple parallel tasks. This is often the case where there is little to no dependency or need for communication or results among the parallel tasks.

Essentially, these problems can be divided into independent chunks of work that can be distributed across several threads, executing concurrently, thereby achieving greater performance.

**Data parallelism** Embarrassingly parallel problems are well-suited for *data parallelism* [37], where data is divided into subsets and processed simultaneously by multiple processors. Since each task operates independently on different data chunks, data parallelism maximizes resource utilization and speeds up computation. This approach is ideal for applications like image processing and simulations, where the same operation is performed on different data segments in parallel, enhancing efficiency and performance.

#### Estimating $\pi$ through the Monte Carlo method

The *Monte Carlo method* is a classic example of an embarrassingly parallel problem [45]. This technique relies on repeated random sampling to obtain numerical results for deterministic problems. One common application is the estimation of $\pi$.

Consider a scenario where darts are randomly tossed at a square dartboard defined by the opposing vertices $(-1, -1)$ and $(1, 1)$. Within this square is an inscribed circle with a radius of 1, resulting in an area of $\pi$. Assuming the darts are uniformly distributed and consistently land within the square's boundaries, the ratio of darts landing inside the circle to the total number of darts should approximately satisfy the following equation:

$$\frac{number\ in\ circle}{total\ number\ of\ tosses} = \frac{\pi}{4}$$

We are perfectly capable of parallelizing this problem. Multiple threads can independently toss the same amount of darts, each returning the *number in circle* to the main thread. This parallel approach is significantly more efficient than a sequential one. The following diagram illustrates the parallelized problem divided among three threads, where $x = total\ number\ of\ tosses\ /\ 3$:

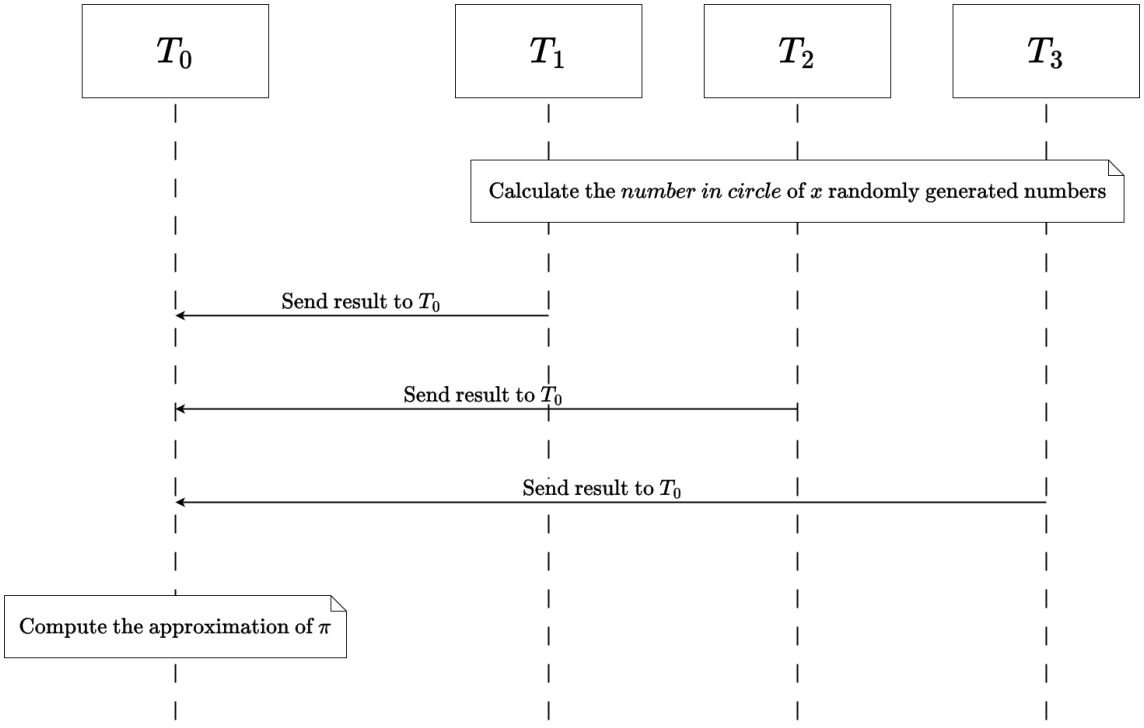



Figure 2.1: Sequence diagram of the $\pi$ estimation through the Monte Carlo method

Henceforth, any reference to the Monte Carlo method will refer to its use in estimating $\pi$.

### 2.2.2 Futures and promises

The terminology and implementations of futures and promises can vary across programming languages and paradigms, making precise definitions challenging.

A *future* [14] is a concept originally introduced as the representation of a promise to deliver the value of an expression—that was given to the evaluator—at some later time. A process is created for each newly created future to evaluate a given expression. When the value of a future is explicitly needed, if the evaluation has finished, it is immediately available; otherwise, the process waits until it completes.

In the broadest sense, both futures and *promises* can be seen as a value that will eventually become available [1]. When a task starts, a future is returned immediately, allowing the program to continue executing; the future will eventually hold the task's result upon completion. However, there are essential differences between these concepts:

- A future is a read-only placeholder for a result yet to be computed asynchronously.
- A promise is a writable, single-assignment variable that completes a future. It can complete a future with a value to indicate success or an exception to indicate failure.

In Scala, futures and promises are non-blocking, utilizing callbacks instead of typical blocking operations. However, in many implementations, futures are blocking; once a process requires its result, it must block its computation and wait until the future is completed.

To support these functionalities, futures libraries typically provide methods to create and return futures, check if a task has been completed, block until the result is available, or attach callbacks that will be invoked when the result is ready.

Common applications of futures and promises are parallelism and concurrency, asynchronous I/O operations, web development, asynchronous event handling, data streaming, deferred execution, and resource management.

### 2.2.3 Divide-and-conquer

*Divide-and-conquer* [21] is a simple yet powerful programming paradigm in which an algorithm recursively breaks down problems into smaller sub-problems until they become simple enough to solve directly. The solutions to the sub-problems are then combined to solve the original problem.

In other words, there is a *base case*—solving the problem directly—and the *recursive case*, performing the following three steps:

1. **Divide**: Break down a problem into two or more smaller sub-problems.
2. **Conquer**: Solve the sub-problems recursively.
3. **Combine**: Merge the sub-problems' results into the solution of the original problem.

A divide-and-conquer algorithm resembles a tree structure. The initial task splits into branches that continue to divide until they reach the base cases (the *leaves*). The results from these leaves are combined, propagating back up the tree until the final result is obtained.

Common problems addressed by divide-and-conquer include sorting algorithms and problems based on a *recurrence relation*. A recurrence relation is an equation where the $n^{\text{th}}$ term of a sequence derives from some combination of its preceding terms. A classic example is the *Fibonacci sequence*.

#### The Fibonacci sequence

In mathematics, the Fibonacci sequence [21] defines each number as the sum of the two preceding ones. The sequence commonly starts from $0$ and $1$, with the first few values being $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144\}$.

We can formally define the Fibonacci numbers using the following recurrence relation (for $n > 1$):

$$F_0 = 0, \quad F_1 = 1, \tag{2.1}$$

$$F_n = F_{n-1} + F_{n-2} \tag{2.2}$$

Since each number in the Fibonacci sequence equals the sum of the two preceding ones, we can break down the problem $F_n$ into the two sub-problems $F_{n-1}$ and $F_{n-2}$. The solutions to these sub-problems are then combined to provide the solution to the original problem. The base cases of the Fibonacci sequence correspond to Equation 2.1, while the recursive case aligns with Equation 2.2. This description matches the divide-and-conquer paradigm.

### 2.2.4 Streams

In programming languages, *streams* [10, 17] are sequences or flows of data elements that can be continuously and sequentially read from or written to. They are widely used for handling input and output operations, especially when dealing with large volumes of data or when data is produced or consumed over time.

Streams abstract the underlying details of reading and writing data, making it easier for programmers to work with different data sources and destinations without worrying about the specific implementation details. They provide a uniform interface to interact with various data sources, such as files, network sockets, standard I/O, and more.

Functions that process data by maintaining endpoints to send and receive data elements within streams are known as *filters*. *Pipelines* are formed by linking multiple filters in a sequence, enabling efficient and systematic data processing as it flows through different stages. This stream composition enhances the flexibility and expressiveness of stream-based applications, facilitating complex data processing tasks in a scalable manner.

Moreover, streams are commonly used in parallel and concurrent programming scenarios. They play a significant role in efficiently handling data processing tasks, especially when working with multiple cores, threads, or processing units.

## 2.3 Concurrency in programming languages

Concurrent programming languages enable the simultaneous execution of processes or threads, structuring a program to handle multiple tasks at once.

There are numerous concurrent programming languages, each with unique characteristics, features, use cases, and advantages. This section reviews some relevant concurrent programming languages, categorized by their communication model and synchronization methods.

Concurrent programming languages generally fall into two categories:

- **General-purpose languages with support for concurrency**: These languages have primary purposes beyond concurrency but offer libraries and frameworks for concurrent programming. Examples include Rust [7, 18] and Java [26, 27]. Moreover, Haskell [5, 41] is

another poignant example, as it is an essential reference in functional programming for its powerful concurrency tools.

- **Languages designed for parallelism and concurrency**: These languages were developed to address parallelism and concurrency, offering specialized primitives for such tasks. Examples include Go [22], Erlang [13], and StreamIt [8].

When comparing programming languages, it is important to distinguish them based on their programming paradigm. The two main paradigms are:

- **Imperative languages**: These languages, such as Rust, Java, and Go, instruct a machine on how to change its state.

- **Declarative languages**: These languages, such as Erlang and Haskell, declare properties of the desired result without specifying how to compute it. Erlang and Haskell are functional languages expressing results through a series of function applications.

Except for Erlang, most of these languages support shared-memory and message-passing communication models but tend to be more suited for one of them. Below is an overview of how each language approaches these communication models:

**Rust** Rust does not prescribe a specific communication model. As a low-level system programming language, Rust prioritizes control to achieve more nuance and efficiency in handling concurrency, providing synchronization mechanisms for shared-memory communication. Moreover, the language supports (mainly) asynchronous channels for message-passing.

**Java** Java, a versatile language, primarily uses shared-memory communication through built-in primitives and packages. While it can implement message-passing (e.g. the *producer-consumer* [27] design pattern or the *actor model* [29, 28]), it is not designed for this model by default, requiring additional workarounds and effort.

**Haskell** Haskell offers powerful concurrency through its *Concurrent Haskell* library [35, 41], which provides abstractions to launch and manage lightweight threads. These threads can communicate by sharing memory via *MVars* or using *software transactional memory* (STM) for atomic memory operations. Haskell also supports message-passing between threads through buffered (asynchronous) channels. Additionally, it uses *sparks*, lightweight units of computation scheduled to a pool for execution when processors are available.

**Go** Go features *goroutines* (lightweight threads) and channels, primarily using message-passing. Go's channels are synchronous by default but can be made asynchronous by adjusting the buffer size. Moreover, Go also supports shared-memory and some of its mechanisms through packages.

**Erlang** Erlang is an asynchronous message-passing programming language that employs the *actor model*, where processes communicate via *mailboxes* instead of channels.

**StreamIt** Unlike the others, StreamIt is a streaming-oriented language designed for real-time data processing. It emphasizes pipeline parallelism, where filters execute concurrently within a stream. We will discuss it more in-depth in the next section.

## 2.4 Related work

Parallel and concurrent programming has seen the development of various tools and techniques to address the previously discussed concepts and patterns. This section provides an in-depth overview of existing tools and approaches directly related to the goals and ambitions of this thesis, helping to contextualize our contributions and the decisions behind them.

### 2.4.1 Message-Passing Interface

The *Message-Passing Interface* (MPI) [43, 36] is a message-passing library interface specification that primarily addresses the message-passing parallel programming model. It facilitates the exchange of data between processes through *cooperative operations*, harnessing data parallelism effectively. MPI is a specification, not an implementation or a language, in which all operations are expressed as functions, subroutines, or methods aligned with language bindings. As a reference, programming languages such as C and Fortran support popular MPI implementations.

MPI supports the *multiple instruction multiple data* (MIMD) paradigm, where different processors execute different instructions on different data sets simultaneously. Within MIMD, MPI also accommodates the *single program multiple data* (SPMD) model, in which processors execute the same program but on different data subsets, possibly following different execution paths [37]. In MPI, processes execute the same program but might have different inputs or data subsets to work on or take different execution paths based on their rank.

In a distributed communication environment, the benefits of standardization are significant, as higher-level routines and abstractions build upon low-level message-passing routines. MPI aims to establish a widely used standard for writing message-passing programs by enabling efficient communication, supporting heterogeneous environments, and providing a reliable communication interface.

MPI's capability to harness data parallelism makes it particularly effective for embarrassingly parallel problems, where tasks can be divided into independent subsets, enabling concurrent processing with minimal inter-task communication.

This overview focuses on explaining and demonstrating MPI through its implementation in C.

#### Groups and communicators

In order to define communication contexts and groups, we need to introduce a few notions first.

*Communicators* specify the communication context for a communication operation. Each communication context establishes an independent "communication world", ensuring that messages sent in different contexts do not interfere.

A *group* (or group of processes) defines an ordered collection of processes, each associated with a unique *identifier* (i.e., a *rank*). Although groups might be manipulated separately from communicators, only communicators can be used in communication operations; hence, groups are often used within a communicator to describe the participants in a "communication world", distinguishing them through their respective ranks.

Furthermore, communicators are divided into *intra-communicators* for operations within a single group of processes and *inter-communicators* for operations between two groups of processes. Intra-communicators are most commonly used for message-passing in MPI. In contrast, inter-communicators might be useful when an application is built by composing parallel modules, allowing these modules to communicate. This overview focuses on intra-communicators.

#### The point-to-point and collective communication

MPI defines a few relevant communication types, particularly *point-to-point* and *collective* communication.

In MPI, the definition of blocking and non-blocking communications hold a different meaning from our synchronization terminology, and we will not discuss those nuances of the specification.

**Point-to-point communication** Is the most basic mechanism of MPI, involving direct sending and receiving of messages between processes. In C, this communication is achieved through the `MPI_Send` and `MPI_Recv` operations, varying from implementation to implementation. To understand how this mechanism works, consider the following code example:

```
1  int world_rank;
2  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
3  int world_size;
4  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
5
6  int number;
7  if (world_rank == 0) {
8      number = -1;
9      MPI_Send(&number, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD);
10 } else if (world_rank == 1) {
11     MPI_Recv(&number, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
12     printf("Process 1 received number %d from process 0\n", number);
13 }
```

Figure 2.2: MPI - Point-to-point C code example

First, the program sets up the communication world using `MPI_Comm_rank` and `MPI_Comm_size` to establish each process's world size and rank. Then, if the executing pro-

cess holds rank zero, it initializes a number to $-1$ and sends it to process one using `MPI_Send`. If the executing process holds rank one, it calls `MPI_Recv` to receive and print the number.

MPI supports different communication modes, such as *standard*, *buffered*, and *synchronous*, catering to different needs for synchronization and buffering. `MPI_Send` satisfies the standard mode, letting MPI decide whether or not to buffer outgoing messages. `MPI_BSend` addresses the buffered mode, and the operation's completion does not depend on a matching receive. At last, in the synchronous mode, `MPI_SSend` completes successfully only if a matching receive is called. Therefore, point-to-point communication allows synchronous and asynchronous communication depending on the user's needs.

**Collective communication** Involves all participants in a group of processes associated with a communicator. The core patterns this communication covers are:

**Barrier** Forms a barrier where no processes in the communicator can pass until all of them call the operation, i.e., it behaves as synchronization across all processes in a group.

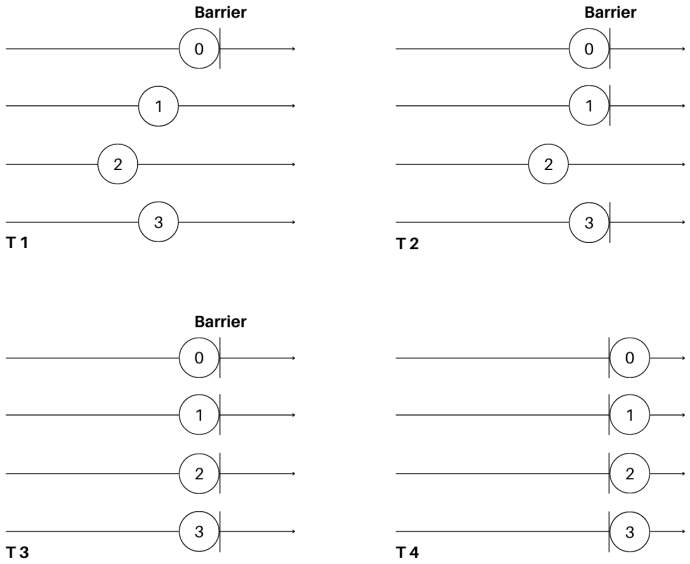



Figure 2.3: MPI - Illustration of the barrier pattern

**Broadcast** Sends the same data from one process to all processes.

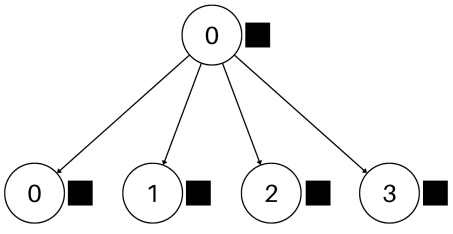



Figure 2.4: MPI - Illustration of the broadcast pattern

**Scatter** Sends data from one process to all processes, distributing chunks of an array to different processes.

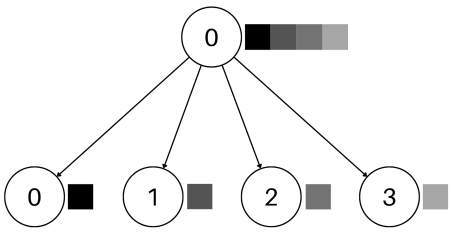



Figure 2.5: MPI - Illustration of the scatter pattern

**Gather** Receives elements from all processes and gathers them into a single process.

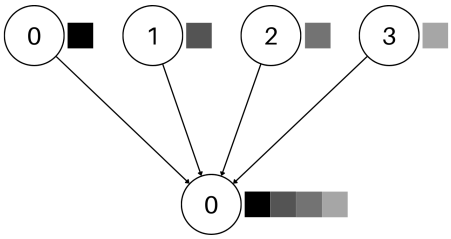



Figure 2.6: MPI - Illustration of the gather pattern

**Reduce** Takes an array of elements on each process and sends its reduced result to a single process. Examples could be folding a list by applying a combining function, such as summing numbers.

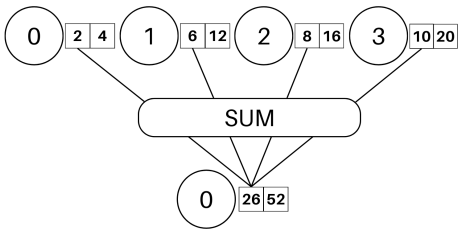



Figure 2.7: MPI - Illustration of the reduce pattern

**Allgather and allreduce** Variation of gather and reduce where all processes receive the result.

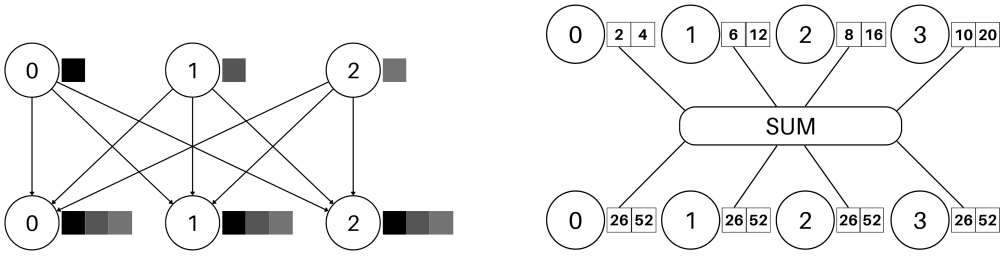



Figure 2.8: MPI - Illustration of the allgather and allreduce MPI patterns

All participants in the communication must call the collective operation. This holds for both intra-communicators and inter-communicators.

As observed above, some patterns have a single sending or receiving process called *root*. Usually, arguments in the function of the operations allow specifying which rank is the root process, so the operation knows to whom to attribute a different behaviour. For example, in the scatter operation, specifying the root as the process of rank 0 ensures that whenever that process calls the operation, it will distribute the contents of a structure to the remaining participants while they receive a chunk of the original contents.

### Implementing Monte Carlo

A practical example might help provide a clearer understanding of collective communication. Consider the following MPI implementation of the Monte Carlo method described in Section 2.2.1:

```
1  MPI_Bcast(&n_points, 1, MPI_INT, 0, MPI_COMM_WORLD);
2
3  int points_per_proc = calculate_points_per_process(n_points, size);
4  local_count = calculate_local_count(points_per_proc);
5
6  MPI_Reduce(&local_count, &total_count, 1, MPI_INT, MPI_SUM, 0,
       MPI_COMM_WORLD);
7
8  if (rank == 0) {
9      calculate_and_print_pi(total_count, n_points);
10 }
```

This program demonstrates how to divide work across processes, perform computations on a broadcasted value, and reduce the results to a final result. Unlike the representation in Fig. 2.1, MPI implementations also use its root process to solve the problem. The program takes the following steps:

1. The process of rank 0 broadcasts the number of points to all processes.
2. Each process calculates its portion of tosses (points) and the corresponding count of points inside the circle.
3. Reduce all local counts to get the total count.
4. Process 0 calculates and prints the estimation of $\pi$.

While there are more nuanced and specific collective operation variants, this demonstration focuses on the essential aspects of MPI; thus, we will refrain from discussing them.

At last, it is important to note that collective communication calls can use the same communicator as point-to-point communication since MPI guarantees that messages exchanged through collective communication will not be confused with those exchanged through point-to-point communication.

### 2.4.2 ForkJoin

*ForkJoin* [20] is a technique for solving problems recursively by splitting them into subtasks running in parallel. This involves breaking down a part of the problem and then waiting and converging to compose the final results. It is the idea of constructing and managing queues of tasks and worker threads as concurrent versions of divide-and-conquer algorithms. Lea, who developed ForkJoin more in-depth, also proposed a Java implementation [39, 40].

There are two operations in ForkJoin: `fork` and `join`. The `fork` operation starts a new parallel subtask, while the `join` operation impedes the current task from proceeding, waiting until the forked subtask has finished its work. The most well-known algorithms are recursive in order to employ divide-and-conquer by splitting tasks until they are simple enough to be solved sequentially.

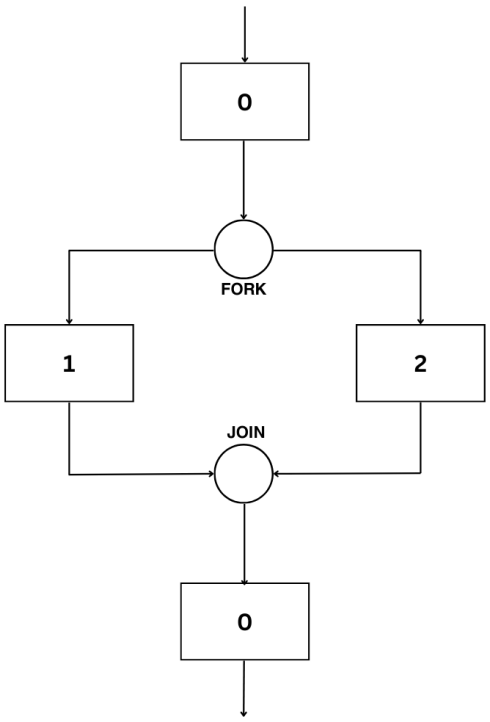



Figure 2.9: ForkJoin - A representation of the *fork* and *join* operations

These tasks have simple and regular synchronization and management requirements and more efficient scheduling tactics than general-purpose threads. While tasks are instances of a lightweight executable class, threads are not. Thread frameworks are often too heavy for most ForkJoin programs because the cost of constructing and managing a thread can exceed the computation time of a task itself. However, removing overhead or tuning thread scheduling is impossible since threads form the basis of many other concurrent and parallel programming styles.

A control and management entity sets up the worker threads pool and initiates threads as needed. Each thread processes tasks from a queue, managed and executed by the queuing and scheduling discipline.

A ForkJoin framework includes a *work-stealing* scheduler. Work-stealing is a tactic to optimize thread usage by allowing a worker thread with an empty scheduling queue, i.e., no local tasks to execute, to attempt to take one from another (randomly) chosen worker. This process is called *stealing*.

### Task granularity

Another important factor of ForkJoin is the *task granularity* of regular and irregular parallel programs, which refers to the number and size of tasks created to perform a given computation. If the granularity is too coarse and tasks are too large, there will be insufficient parallelism. Conversely, if the granularity is too thin, excessive context switching between tasks reduces efficiency.

Task granularity can be controlled by limiting new task creation and executing workloads sequentially. This is achieved through algorithms defining criteria for executing tasks sequentially or concurrently. The most well-known algorithms are *MaxTasks*, *MaxLevel*, *Adaptive Tasks Cut-off*, *Load-Based*, and *Surplus Queued Task Count*. Other algorithms, such as *Max Queue Size*, *StackSize*, and *MaxTasks with StackSize*, may be used depending on the specific problem and requirements [19, 23, 24].

### Implementing the Fibonacci sequence

The ForkJoin framework provides a parallel approach for implementing the Fibonacci sequence. Below is the Java implementation:

```
1  public class ForkJoinFib extends RecursiveTask<Integer> {
2      private int n;
3
4      public ForkJoinFib(int n) { this.n = n; }
5
6      @Override
7      protected Integer compute() {
8          if (n == 0 || n == 1) return n;
9
10         ForkJoinFib f1 = new ForkJoinFib(n - 1);
11         f1.fork();
12
13         ForkJoinFib f2 = new ForkJoinFib(n - 2);
14         f2.fork();
15
16         return f1.join() + f2.join();
17     }
18
19     public static void main(String[] args) {
20         ForkJoinFib fib = new ForkJoinFib(47);
21         fib.compute();
22     }
23 }
```

The `main` function initializes a `ForkJoinFib` object with the number 47 and invokes the task, which executes the `compute` function on an independent thread.

The critical part is the `compute` function, where the Fibonacci sequence calculation occurs. The function splits the problem into two sub-problems by creating two new `ForkJoinFib` objects. It forks the sub-problems, allowing them to process in parallel. The first sub-problem computes the Fibonacci number for $n-1$, while the second computes the Fibonacci number for $n-2$, representing the computation of the preceding numbers. The results from these sub-problems are combined by joining the first task and adding it to the second result.

This process repeats in each branch until it reaches a base case (0 or 1), returning the base value directly. As tasks complete, their results are combined, gradually building the solution from the base cases to the final result.

While this implementation effectively demonstrates parallelism, it is relatively inefficient regarding time complexity and thread management, resulting in high memory usage. Enhancements in granularity control and task management can improve its performance.

### 2.4.3 StreamIt

In contrast to the little language support for practical, large-scale stream programming, *StreamIt* [51, 9] is a programming language designed specifically for modern stream programming.

Streaming applications process continuous streams of data, such as multimedia processing, signal processing, and network packet processing.

Ultimately, the StreamIt programming language aims to simplify the coding of signal processing and other streaming computations while performing stream-specific optimizations for high-performance programming. The language has two main goals: providing high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain and serving as a standard machine language for grid-based processors.

StreamIt programs are stream graphs containing blocks with a single-input and a single-output, describing the function of atomic blocks and the structure of composite blocks.

#### Filters

*Filters* are the most basic units of StreamIt, where all computation takes place. Each filter has a `work` function, allowing a filter to communicate with other blocks through the input/output channels (FIFO queues). These channels support three operations:

- `pop()`: removes an item from the end of the channel and returns its value.
- `peek()` or `peek(i)`: returns, without removing, the value of the last item in the channel or the item $i$ spaces from the end of the channel, respectively.
- `push(x)`: writes $x$ on the front of the channel.

Filters also have an `init` function, called at initialization time, responsible for establishing the filter's initial state and specifying its I/O types and data rates.

### Constructs

In order to compose filters, StreamIt provides three distinct constructs: *Pipeline*, *SplitJoin* and *FeedbackLoop*. These constructs specify predefined ways to connect filters into single-input and single-output blocks.

The most basic stream composite is the Pipeline, which allows for building a filter sequence. Filters are added to the Pipeline through calls to the `add` function.

SplitJoins are used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. Incoming data passes through a splitter, is redistributed to the child streams for processing, and then recombined through a joiner into a single output stream. Like in Pipelines, calls to `add` specify the filters of a SplitJoin. Furthermore, three splitter types determine how to distribute items from the input of the SplitJoin to the parallel components:

- `duplicate`: replicates each data item and sends a copy to each parallel stream.
- `roundrobin`: distributes each item to one of the child streams in order. It is possible to specify the number of items distributed to each child.
- `null`: indicates that no parallel components require any input, and there are no input items to split.

Similarly, two joiner types specify how to receive the outputs of the parallel streams on the output channel of the SplitJoin: `roundrobin` and `null`, both behaving according to their respective splitter types.

Finally, FeedbackLoop provides a way to create cycles (loops) in the stream graph. Each FeedbackLoop contains the following components:

- A body stream (`setBody`): the block around which a backwards "feedback path" is created.
- A loop stream (`setLoop`): performs some computation along the feedback path.
- A splitter (`setSplitter`): distributes data between the feedback path and the output channel at the bottom of the loop.
- A joiner (`setJoiner`): merges items between the feedback path and the input channel at the top of the loop.

In other words, a FeedbackLoop has a body stream whose output passes through a splitter; one branch of the splitter leaves the loop, and the other goes to the loop stream. The output of the loop stream and the loop input go through a joiner to the body's input.

### Implementing the moving average algorithm

In statistics, a *moving average* is a calculation to analyze data points by creating a series of averages of different selections of the complete data set. This technique can, for instance, be used

as a forecasting method [12]. The following snippet of code is a simplified implementation of the moving average using the StreamIt programming language:

```
1  void->void pipeline MovingAverage {
2      add IntSource();
3      add Averager(10);
4      add IntPrinter();
5  }
6
7  int->int filter Averager(int n) {
8      work pop 1 push 1 peek n {
9          int sum = 0;
10         for (int i = 0; i < n; i++)
11             sum += peek(i);
12         push(sum / n);
13         pop();
14     }
15 }
```

The `MovingAverage` pipeline comprises of three primary filters: the `IntSource`, `Averager` and `IntPrinter`. Here is how it operates:

- `IntSource`: Generates a continuous stream of integers.
- `Averager`: Computes the moving average over a window size of ten. As each integer arrives in the stream, it includes this integer in its computation, maintaining a sliding window of the last ten integers. It then calculates the average of these integers and outputs each result.
- `IntPrinter`: Receives and prints all computed moving averages.

The `Averager` filter dynamically adjusts its computation with each incoming integer. It continuously recalculates the moving average based on the most recent ten integers in the stream, ensuring that `IntPrinter` receives and prints each computed average in real time.

# Chapter 3

# The FreeST programming language

In this chapter, we delve into the essential characteristics of the FreeST programming language, laying the groundwork for understanding the nuances of its unique features and challenges concerning parallel and concurrent programming.

*FreeST* [11] stands out as an asynchronous message-passing concurrent functional programming language, distinguished by its powerful type system centred around *session types*. Session types enable the specification of communication protocols. FreeST's type system governs the interactions on communication channels, ensuring that if a program is typable, i.e., passes the type checker, it correctly adheres to the specified protocol.

Implemented in Haskell, FreeST closely resembles the language's syntax and leverages its concurrency and channel features.

## 3.1 Session types

The most distinctive characteristic of the FreeST programming language is the integration of *session types*.

Session types, which form the core of communication through channels in FreeST, were initially introduced by Honda et al. [32, 33, 34]. They focus on binary (two-party) sessions with well-defined protocols between two endpoints, guaranteeing that communication errors never occur within a session. The session communication protocols are defined based on the four session-type constructors outlined in Table 3.1.

This defines a kind of *point-to-point* message-passing communication in a channel and guarantees that a communication never goes wrong by ensuring that messages are sent and received in the exact specified order.

Each participant in the communication holds an endpoint of opposite types, describing the actions required to follow the protocol.

To understand how it works, let us consider a *one-to-one* client-server communication example. In this scenario, the client wishes to send an integer to the server and receive another.

Accordingly, from the client's perspective, the protocol is as follows:

```
ClientChannel = !Int;?Int
```

| **Constructor** | **Description** |
|---|---|
| $!T$ | Message sending, send $T$. |
| $?T$ | Message receiving, receive $T$. |
| $\oplus\{l_i : T_i\}$ | Internal choice. Branch selection, select $l_i$, and continue as $T_i$. |
| $\&\{l_i : T_i\}$ | External choice. Branch matching, match $l_i$, and continue as $T_i$. |
| $T;U$ | Sequencial composition. Do $T$, then $U$. |

Table 3.1: FreeST - Session type constructors

Whereas, from the server's perspective, the appropriate protocol is:

```
ServerChannel = ?Int;!Int
```

Note that these protocols are each other's reflections, i.e., the server's protocol is the opposite of the client's. Furthermore, this means the communication is not one-sided: if the client sends a value, the server receives it. This relation is known as *duality*, where each protocol is dual to the other. In other words, we can rewrite the definition for `ServerChannel` as follows:

```
ServerChannel = dual ClientChannel
```

While the basic constructors are somewhat intuitive, the remaining two are more interesting and allow for more expressive and elaborate scenarios. Selection and branching introduce what can be interpreted as the power of options within protocols. Given the previous example, based on the client's decisions, we can now rewrite the protocol to work with several options offering different behaviours. For instance, we are allowed to have the following protocol:

```
ClientService = ⊕{IsOdd: !Int;?Bool,
                  Succ: !Int;?Int}
```

This protocol allows the client to choose between two options, triggering two different continuation behaviours.

Accordingly, the server-side protocol of the communication needs to match the client's protocol, i.e., it should be dual to it:

```
ServerChannel = dual ClientChannel
```

Or, in more explicit terms:

```
ServerService = &{IsOdd: ?Int;!Bool,
                  Succ: ?Int;!Int}
```

If the client selects the branch `IsOdd`, the server serves it by receiving an integer and sending a boolean back. If the `Succ` branch is selected, the server receives and sends an integer instead. So, in this sense, the server's protocol is dual to the client's.

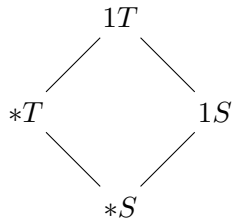
These protocols are ensured to be fulfilled by a type checker, guaranteeing that every element

in a channel is consumed by sending or receiving values accordingly until the channel is closed. If this is not confirmed, the program is considered not well-typed (or typable).

Another notable feature of session types is recursion, allowing the creation of recursive protocols combined with branching. This capability is particularly powerful in scenarios involving protocols for binary trees. However, in regular session types a limitation arises due to the need for an auxiliary stack to reconstruct structures like binary trees. To address this limitation, Thiemann and Vasconcelos [50] proposed *context-free session types*, breaking tail recursion and enabling non-regular recursion. Notably, FreeST implements context-free session types.

### Session types in FreeST

FreeST relies on polymorphism to type its channels properly, requiring a kinding system to check the formation of types. A *kind* consists of a pair composed of a basic kind and a multiplicity. Basic kinds distinguish functional types ($T$) from session types ($S$). Multiplicities control the number of usages of a value in a given context: exactly one (*linear* - 1) or zero or more (*unrestricted* - $*$). We have the following subkinding relation:




This subkinding relation allows an unrestricted type when a linear one is required, and session types ($1S$, $*S$) can be used in place of an arbitrary type ($1T$).

For the message-passing communication, FreeST supports *linear* and *shared* channels, which work with linear ($1S$) and unrestricted ($*S$) session type protocols, respectively. Linear channels restrict communication to *one-to-one* relations, while shared channels eliminate this restriction, allowing for all types of relations: *one-to-many*, *many-to-one*, *many-to-many*, and *one-to-one*.

Initially, FreeST only supported linear session types. Barros et al. [16] proposed the implementation of shared channels for the FreeST programming language, introducing *session initiation*, which combines both linear and shared session types. This addition breaks with the restriction of the limited one-to-one communication pattern and allows communication between more than two parties, making FreeST a much more flexible and mature language prepared for various real-world situations and scenarios.

Returning to the client-server example, its implementation depends on the number of participants in the communication. If it is known that it will remain a one-to-one relation, we can use linear channels; otherwise, shared channels are necessary.

In FreeST, the one-to-one version can be expressed using the following linear session types:

```
type ClientService = ⊕{IsOdd: !Int;?Bool,
                       Succ: !Int;?Int}
```

```
type ServerService = dual ClientService
```

To illustrate this, we can use the visual representation method proposed by Barros [16]:


Figure 3.1: Linear channels - One-to-one representation

The many-to-one version, on the other hand, requires the following unrestricted session type:

```
type Server = *?ServerService
```

This can be visually represented as follows:

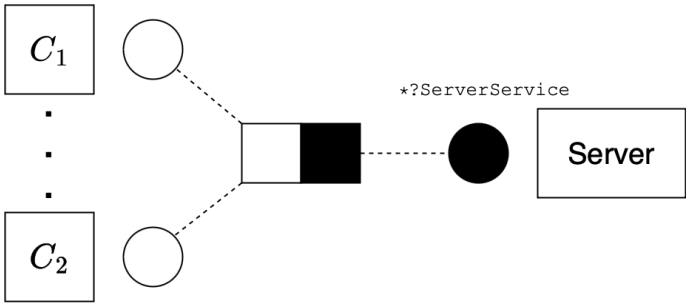



Figure 3.2: Shared channels - Many-to-one representation

It may be tempting to assume that shared channels are always superior to linear channels since they eliminate relation restrictions on communication, but this is not necessarily the case. Although shared channels can combine both types through session initiation to distribute linear endpoints to other processes, they restrict expressiveness by offering the same interface indefinitely, Furthermore, being "first come, first served", can be a "double-edged sword", eliminating overhead but introducing unpredictability in the communication order. Thus, choosing between linear and shared channels depends on a specific scenario and requirements.

### Essential FreeST features

As a concurrent programming language, FreeST provides the `fork` primitive to launch new lightweight threads. Additionally, it includes the `forkWith` abstraction, which simultaneously creates a new channel and launches a new asynchronous computation holding one of the endpoints.

To interact with channels, FreeST provides the operations described in the Table 3.2. Alternatively to the `match` operation, FreeST supports pattern matching akin to other functional programming languages.

Moreover, FreeST includes the `dualof` primitive to write the dual type of a session type.

| Operation | Description |
|---|---|
| **new** $T$ | Creates a channel of a given type $T$. |
| **send** $e$ $c$ | Sends the value of an expression $e$ through a channel endpoint $c$. |
| **receive** $c$ | Receives a value from a channel endpoint $c$. |
| **select** $l$ $c$ | Internal choice. Selects a branch $l$ in a $c$ channel endpoint's protocol. |
| **match** $c$ **with** $\{l_i \rightarrow e_i\}$ | External choice. Uses the channel endpoint $c$ to serve the branch $l_i$ selected by the opposite endpoint. |
| **close** $c$ | Closes a channel $c$. |
| **wait** $c$ | Waits for a channel $c$ to be closed. |

Table 3.2: Operations to interact with channels in FreeST

#### Function application operators

Similarly to other functional programming languages, FreeST provides the `$` and `▷` operators to facilitate function composition. Their primary purpose is to reduce the need for parentheses, improving readability and making nested function calls cleaner. The `$` operator allows expressions like `f $ g $ h x` instead of `f (g (h x))`, creating clean pipelines of function applications.

Conversely, the `▷` operator enables forward function application, allowing expressions such as `x ▷ h ▷ g ▷ f`. This alignment with natural left-to-right reading makes the data flow through functions more readable, resembling a pipeline of transformations. In FreeST, this is particularly useful in situations like `close (send y (send x c))`, which can be rewritten as `c ▷ send x ▷send y ▷close`.

## 3.2 The math client example

To gain a clear understanding of FreeST's session types, we will explore the *math client* example [25] to see how they work in practice.

The math client is a well-known example of a client communicating with a server offering two choices: `Plus` to add two numbers and `Eq` to verify that two numbers are equal. Both options require the client to close and finish the communication.

In FreeST, we can write the following protocol to establish the starting point for this example:

```
type MathClient = ⊕{Plus: !Int;!Int;?Int,
                    Eq: !Int;!Int;?Bool} ; Close
```

This protocol defines the entire communication logic of the program. If a client chooses the `Plus` option, it must send two integers and receive another in return. Conversely, if the client chooses the `Eq` option, it must send two integers and receive a boolean in return. The server side

of the communication holds the dual endpoint and is responsible for responding to the client's choice.

A possible implementation for this protocol is as follows:

```
1  mathServer : dualof MathClient → ()
2  mathServer (Plus c) = let (x, c) = receive c in
3                        let (y, c) = receive c in
4                        c ▷ send (x + y) ▷ wait
5  mathServer (Eq   c) = let (x, c) = receive c in
6                        let (y, c) = receive c in
7                        c ▷ send (x == y) ▷ wait
8
9  mathClient : Int → Int → MathClient → Int
10 mathClient x y c = let (n, c) = c ▷ send x ▷ send y
11                                   ▷ receive in
12                    close c; n
13
14 main : Int
15 main = let c = forkWith mathServer in
16        mathClient 2 4 c
```

This implementation contains two essential components: the `mathClient` and `mathServer` functions, encapsulating each side's communication behaviour and logic.

The `main` function serves as the entry point responsible for executing these components. First, it launches a new concurrent computation using the `forkWith` abstraction, applying a dual `MathClient` endpoint to the `mathServer` function and executing it concurrently. With the `mathServer` executing concurrently and ready to respond to the client's operations, the `mathClient` is executed sequentially in the main thread to prevent the program from terminating precociously.

The `mathClient` implements a possible client by choosing the `Plus` option. Accordingly, the `mathServer` implements both options and is prepared to serve them. By selecting the `Plus` option, the client sends two integers, which the server receives. Then, the server adds the integers and sends the result back to the client responsible for receiving it. Finally, the communication is closed.

Examining the code closely, we observe the dual relationship inherent in these communications. When the client sends data to the channel through the `send` primitive, the server must respond accordingly by receiving the data with the `receive` primitive, and vice versa. To conclude the communication, the client follows the protocol and calls the `close` primitive, while the server, being dual to the client, responds with the `wait` primitive.

## 3.3 Challenges

FreeST was born as a means to materialize the idea of implementing context-free session types in a programming language, emphasizing its intricate and powerful type checker and innovative core features rather than usability and real-world applications. As an academic-oriented language, FreeST remains in an embryonic state, presenting obstacles that demand a deep understanding of

session types and linearity, resulting in a steep learning curve.

While FreeST supports asynchronous communication through message-passing, it reaches a wall when addressing well-known parallel and concurrent programming paradigms and their applications.

As discussed in Chapter 2, concepts such as data parallelism, embarrassingly parallel problems, futures, divide-and-conquer, and streams are essential in parallel and concurrent programming, encompassing a wide range of applications. These concepts pose several challenges that hinder FreeST's practical usage and undermine its parallel and concurrent capabilities.

**Data parallelism** Implementing embarrassingly parallel problems in FreeST reveals several difficulties, such as establishing and maintaining a one-to-many communication framework with linear channels to distribute and retrieve data from asynchronously executing processes. Additionally, various patterns to parallelize these problems often lead to repetitive and complex boilerplate code and difficulties with linearity, which could be abstracted.

**Futures** In FreeST, the return value of an asynchronous computation is always discarded, requiring developers to manually establish and manage channels to retrieve results. Implementing futures could abstract these complexities, allowing accessible management of asynchronous computations and facilitating the approach to problems that benefit from the previously unintuitive divide-and-conquer paradigm.

**Streams** Although FreeST is well-suited for stream programming due to the nature of session types, it lacks a toolset that abstracts and simplifies this programming paradigm to support more structured and well-defined programming practices. This limitation results in convoluted and repetitive code, particularly in more complex scenarios.

This thesis addresses these challenges. The following chapter will further demonstrate how these challenges manifest and provide an in-depth introduction and explanation of our proposals to solve them in a realistic, pertinent, and accessible manner.

# Chapter 4

# Parallel and concurrent modules for FreeST

This chapter introduces three modules specifically designed to address the challenges of parallel and concurrent programming discussed in previous chapters. Each module targets specific issues, providing practical solutions to enhance and facilitate the parallel and concurrent programming experience in FreeST.

Throughout this chapter, examples accompany each section to demonstrate the identified challenges and showcase how these tools effectively resolve them. By the end of this chapter, we will have a comprehensive understanding of the proposed modules and their contributions to enhancing parallel and concurrent programming practices in the FreeST programming language.

## 4.1 The Parallel module

As discussed in Section 2.2, embarrassingly parallel problems are known as trivial examples of problems that benefit from parallelism, being very easy to parallelize.

This section explores how addressing data parallelism and implementing these problems in the FreeST programming language is still no trivial task. We introduce a module designed to address these challenges, simplifying the programming experience and providing a concise and efficient programming environment.

### 4.1.1 Implementing Monte Carlo in FreeST

To illustrate how FreeST handles embarrassingly parallel problems, let us examine the implementation of Monte Carlo in Fig. 4.1. Assume the existence of the `calculateDartsInCircle` and `calculatePi` functions. This implementation closely follows the structure and interprocess communication depicted in Fig. 2.1, comprising four main components:

- The `ParallelStream` protocol: Defines the communication between a centralizing component and the concurrent processes. It describes that one side sends an integer (number of darts), receives another integer (number of darts in the circle), and waits for the other side to terminate the communication.

```
type ParallelStream = !Int;?Int;Wait

process : dualof ParallelStream → ()
process c = let (points, c) = receive c in
            let localCount = calculateDartsInCircle points in
            c ▷ send localCount ▷ close

estimatePi : Int → ParallelStream → ParallelStream
                                  1→ ParallelStream 1→ Int
estimatePi nPoints w1 w2 w3 =
        let pointsPerProc = nPoints / 3 in

        let w1 = send pointsPerProc w1 in
        let w2 = send pointsPerProc w2 in
        let w3 = send pointsPerProc w3 in
        let (y1, w1) = receive w1 in
        let (y2, w2) = receive w2 in
        let (y3, w3) = receive w3 in
        wait w1; wait w2; wait w3;

        let totalCount = y1 + y2 + y3 in
        calculatePi nPoints totalCount

main : Int
main = let w1 = forkWith process in
       let w2 = forkWith process in
       let w3 = forkWith process in
       manager 999 w1 w2 w3
```

Figure 4.1: Parallel - Implementation of the Monte Carlo in FreeST

- The `main` function: The program's entry point, responsible for launching three concurrent processes to compute the number of darts in the circle concurrently. It also sequentially executes the `estimatePi` function, preventing the program from terminating prematurely.
- The `estimatePi` function: Acts as the central component managing communications with the three concurrent processes. It follows the defined protocol by broadcasting the number of darts each process will toss ($999/3$), receiving the number of darts in the circle from each process, and estimating $\pi$.
- The `process` function: Represents the computation performed by the concurrent processes. It follows the protocol dual to `ParallelStream` by receiving a number of darts from `estimatePi`, calculating the number of darts in the circle through the `calculateDartsInCircle` function, and sending the result back.

This implementation presents several noteworthy issues. Primarily, the code exhibits repetition and lacks scalability, confining itself to a predetermined number of processes. As the need for increased parallelism arises, expanding the workforce becomes cumbersome due to the absence of generalized communication endpoint management. Additionally, a *one-to-many* relation is

inherent here, representing a recurring pattern in addressing data parallelism and implementing embarrassingly parallel problems in FreeST.

Furthermore, utilizing FreeST often demands a deep understanding of linearity, introducing a significant time-consuming layer that diverts attention from problem-solving efforts. The same goes for implementing embarrassingly parallel problems, exacerbating the scalability concerns.

In response to these challenges, the forthcoming module aims to rectify these shortcomings, providing a conducive environment for streamlined development.

### 4.1.2 Identifying parallel patterns

To abstract the complexities inherent in addressing data parallelism, our initial step involves identifying recurring patterns typical to this paradigm.

During the process of testing FreeST's capabilities and limitations in handling such problems, we discerned two primary patterns for data distribution:

- Broadcasting the same data across concurrent processes.
- Evenly distributing the contents of a list among concurrent processes.

Conversely, we also identified two primary methods for gathering data from concurrent processes:

- Receiving and appending (sub)lists, resulting in a final list.
- Receiving and applying a combining function to basic values, resulting in a final value of the same type. This process is known as *reduction*.

Furthermore, we observed that realizing this conceptualization requires a centralizing entity to coordinate the identified behaviours, underlining the inherent *one-to-many* relationship. In other words, due to the nature of session types, a distinct component is necessary to distribute and gather data while following a protocol to which other processes will respond. This can be seen in the previous implementation, where the manager acts as the centralizing entity managing communications, while the concurrent processes simply respond to the defined protocol and calculate the number of darts inside the circle.

Section 2.4.1 demonstrates how these pattern descriptions align seamlessly with essential MPI operations for collective communication, crucial for data parallelism:

- Operations for primary data distribution types are represented by *broadcast* and *scatter*, depicted in Figs. 2.4 and 2.5, respectively.
- Operations for gathering data are illustrated by *gather* and *reduce* in Figs. 2.6 and 2.7, respectively.

Additionally, MPI's collective communication reveals other, albeit less essential, operations beneficial for developers tackling embarrassingly parallel problems and some specific scenarios. These include *barrier*, *allgather* (a gather followed by a broadcast), and *allreduce* (a reduce followed by a broadcast), illustrated in Figs. 2.3 and 2.8, respectively.

### 4.1.3 Design and implementation

In MPI, processes distinguish their behaviour based on a rank (identifier). For example, the root process, which also participates in the parallel execution, holds the rank 0. Each process calls an MPI operation abstraction and informs it of its rank. This allows every process to execute the same function while enabling specific processes to execute unique code by wrapping it in a `if (rank == x)` statement, where $x$ is a particular process rank.

In our module, the entity referred to as the root selects operations, distributes, and gathers data, whereas other processes respond to operation calls and contribute to solving the problem. This means the root and the remaining processes have distinct behaviours that must be defined in separate functions. This distinction is necessary due to the restrictions imposed by linear session types, as linearity only allows a linear resource to be held by one party at a time. Consequently, in a communication with two parties, each must hold one of the (linear) endpoints of a channel. Thus, our approach avoids using identifiers and ensures all processes execute the same code. For consistency, the root is not integrated into the problem-solving but solely manages and oversees communications.

**Terminology** We depart from the traditional root-processes terminology used in the original MPI specification. Instead of a root process that participates in the computation, our approach designates the root as a manager. This manager functions as the single central originating or receiving point in the communication. Consequently, we adopt the terminology *manager-workers* to describe their relationship, translating the one-to-many relation to manager-workers.

The initial step in implementing this module involves establishing the communication framework, which includes defining the protocol describing all possible communications between a manager and workers.

We first considered whether to use linear or unrestricted session types. We chose linear session types because, while shared channels are more suitable for one-to-many communications, they lack control over data distribution, operating on a "first come, first served" basis and leading to unpredictability. Although linear channels may incur more overhead than shared channels, they offer the necessary expressiveness and control for our purposes.

Having decided on using linear session types and linear channels, we defined the session types to outline the required communication patterns in this module:

```
type ManagerStream = ⊕{ Broadcast: ![Int]
                      , Scatter  : ![Int]
                      , Gather   : ?[Int]
                      , Reduce   : ?Int
                      , Done     : Wait} ; ManagerStream

type WorkerStream = dualof ManagerStream
```

The `ManagerStream` session type follows a recursive protocol, allowing developers to se-

lect collective operations as needed until `Done` is selected, terminating the communication. The manager, holding endpoints of this type, is responsible for selecting an operation (branch).

Conversely, the `WorkerStream` session type serves the collective operations. Each worker maintains an endpoint of this type and is responsible for answering the operation selected by the manager accordingly.

The protocol does not explicitly cover the `barrier`, `allgather` and `allreduce` operations since they are combinations of two other operations already covered by the protocol.

Currently, the protocol restricts communications to integers and lists of integers because FreeST does not yet support polymorphic session types. Additionally, the `broadcast` operation is limited to exchanging lists of integers only to provide a more generalized solution, enabling the broadcasting of both single values (wrapped in a list) and lists (e.g., `allgather`). In the future, it will be possible to broadcast shared channel endpoints, which is just one example of the interesting use cases that could arise.

To achieve scalability, we must define an algebraic data type (ADT) for the list of endpoints maintaining a communication channel between the manager and each worker rather than managing them independently:

```
data Comm = WNil () | Worker ManagerStream Comm
```

This data type can be considered analogous to MPI communicators, specifying a communication context. Only communicators can be used in communication operations. The `manager` holds an instance of `Comm`, enabling communication with each worker, who holds the respective `WorkerStream` endpoint to respond accordingly.

Having these pieces established, we need an abstraction to facilitate initialization. In other words, we need an abstraction which creates and concurrently launches the group of a given number of workers and executes the manager, establishing and executing the *manager-workers* communication framework:

```
initialize : (Comm → a) → (WorkerStream → ()) → Int → a
```

The `initialize` abstraction takes three arguments: a function encapsulating the manager's behaviour, a function encapsulating the workers' behaviour, and an integer specifying the number of workers to create. It sets the communication channels between the manager and the workers, constructs a `Comm` instance with the respective `ManagerStream` endpoints, and assigns it to the manager. Each worker is given its corresponding `WorkerStream` endpoint. Thus, the manager can communicate with each worker through the `Comm` instance, which provides access to the communication channels with each worker. The Fig. 4.2 visually represents the resulting framework.

The manager oversees communication with all workers while each worker manages its own endpoint. Consequently, the manager's and workers' functions will have different signatures because they maintain and manage distinct states: the manager holds an `Comm` instance and workers hold a `WorkerStream` endpoint.

Finally, to call the operations and parallelize a problem, we provide the following abstractions:

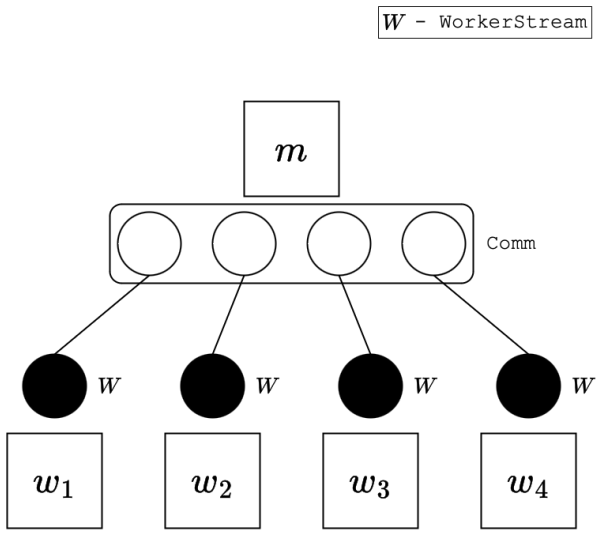



Figure 4.2: Parallel - The manager-workers communication framework

```
mbarrier : Comm → Comm
wbarrier : WorkerStream → WorkerStream

mbroadcast : [Int] → Comm → Comm
wbroadcast : WorkerStream → ([Int], WorkerStream)

mscatter : [Int] → Comm → Comm
wscatter : WorkerStream → ([Int], WorkerStream)

mgather : Comm → ([Int], Comm)
wgather : [Int] → WorkerStream → WorkerStream

mreduce : (Int → Int → Int) → Int → Comm → (Int, Comm)
wreduce : Int → WorkerStream → WorkerStream

mallgather : Comm → ([Int], Comm)
wallgather : [Int] → WorkerStream → ([Int], WorkerStream)

mallreduce : (Int → Int → Int) → Int → Comm → (Int, Comm)
wallreduce : Int → WorkerStream → ([Int], WorkerStream)

mdone : Comm → ()
wdone : WorkerStream → ()
```

Note that there are two abstractions for each operation. Managers should call abstractions with the *m* prefix, while workers should call abstractions with the *w* prefix. Abstractions with the *m* prefix take the manager's `Comm` instance, whereas those with the *w* prefix take the `WorkerStream` endpoint held by each worker.

Thus, the manager and workers hold complementary roles and exhibit distinct behaviours. This is reflected in this separation of an operation into two abstractions. For example, consider the scatter operation, which distributes a list across multiple workers. To perform this operation, the manager must call `mscatter`, distributing the list by sending a chunk to each worker through the

`ManagerStream` endpoints in the `Comm` instance. Correspondingly, each worker calls `wscatter` to receive its chunk from the manager through the `WorkerStream` endpoint:

```
1 worker : WorkerStream → ()
2 worker c = let (xs, c) = wscatter in
3            wdone c
4
5 manager : [Int] → Comm → ()
6 manager xs comm = comm ▷ mscatter xs ▷ mdone
```

Both sides must call their respective abstraction to close the communication: `mdone` for managers and `wdone` for workers.

The complete implementation code for this module can be found in Appendix A.1.

### Comparison with MPI operations

It is important to highlight some unique differences between the operations in our implementation and those described in the MPI specification:

- The *scatter* operation: Unlike the `MPI_Scatter` function in MPI (see Fig. 2.5), which requires developers to manually specify the size of each chunk, our `mscatter` abstraction simplifies this process by automatically dividing the list into evenly sized chunks as best as possible. For example, distributing $[1, 2, 3, 4, 5, 6, 7]$, where the length is not a multiple of three, across three workers will result in chunks $[1, 2, 3]$, $[4, 5]$, and $[6, 7]$.
- The *reduce* operation: The `MPI_Reduce` function in MPI (see Fig. 2.7) takes an array of elements from each process and sends its reduced result to the root process. Conversely, our `mreduce` abstraction expects the manager to receive a single integer from each worker and apply a combining function to them:

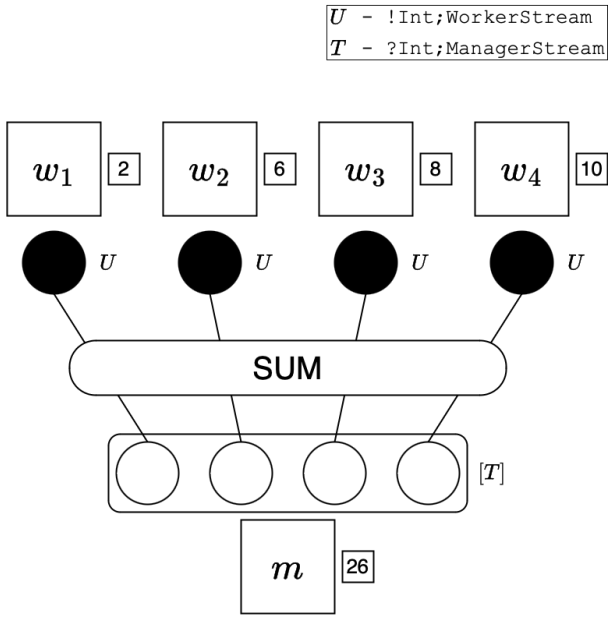



Figure 4.3: Parallel - Illustration of our implementation of the reduce operation

#### Parallelized higher-order functions

Our module also implements a parallelized version of higher-order functions on lists, which optimize heavier transformations:

```
nmap : Int → (Int → Int) → [Int] → [Int]

nfoldl : Int → (Int → Int → Int) → Int → [Int] → Int

nfoldr : Int → (Int → Int → Int) → Int → [Int] → Int

nfilter : Int → (Int → Bool) → [Int] → [Int]

nzipWith : Int → (Int → Int → Int) → [Int] → [Int] → [Int]

nzipWith3 : Int → (Int → Int → Int → Int) → [Int] → [Int]
                                           → [Int] → [Int]
```

A notable aspect of these functions is that we implement them using our own module. For example, here is the implementation of the `nfoldl` function:

```
1  foldlWorker : (Int → Int → Int) → Int → WorkerStream → ()
2  foldlWorker f z c = let (xs, c) = wscatter c in
3                      c ▷ wreduce (foldl f z xs) ▷ wdone
4
5  foldlManager : (Int → Int → Int) → Int → [Int] → Comm → Int
6  foldlManager f z xs comm = let (x, comm) = comm ▷ mscatter xs
7                                             ▷ mreduce f z in
8                             mdone comm; x
9
10 nfoldl : Int → (Int → Int → Int) → Int → [Int] → Int
11 nfoldl n f z xs = initialize (foldlManager f z xs)
12                              (foldlWorker f z) n
```

The only difference from the traditional higher-order functions is the inclusion of an additional parameter to specify the number of workers for parallel execution.

For instance, to sum the elements of an arbitrary list `xs` in parallel, we can use the following code:

```
main : [Int]
main = nfoldl 3 (+) 0 xs
```

### 4.1.4 Implementing Monte Carlo with the module

In Fig. 4.1, we implemented a parallel version of Monte Carlo using pure FreeST's capabilities; in contrast, Fig. 4.4 showcases a re-implementation of the same problem utilizing the described module, highlighting the differences.

Let us analyze the three components present in the code:

- The `main` function: Once again, this serves as the program's entry point and calls the `initialize` function, passing the `manager` and `worker` functions as arguments. It forks three workers and creates three channels, assigns a `WorkerStream` endpoint to each, and

```
worker : WorkerStream → ()
worker c = let (points, c) = wbroadcast c in
           let localCount = calculateDartsInCircle (head points) in
           c ▷ wreduce localCount ▷ wdone

manager : Int → Comm → Int
manager nPoints comm = let pointsPerProc = nPoints / 3 in
                       let comm = mbroadcast [pointsPerProc] comm
                       let (comm, totalCount) = mreduce (+) 0 comm in
                       mdone comm; calculatePi nPoints totalCount

main : Int
main = initialize (manager 999) worker 3
```

Figure 4.4: Parallel - Implementation of the Monte Carlo using the Parallel module

constructs a `Comm` instance. Subsequently, it executes the `manager` function sequentially, providing it with the `Comm` instance.

- The `manager` function: Manages communication with the workers, distributing and gathering resources. It uses the `Comm` instance to select the operation to execute. In this case, the manager uses the `mbroadcast` abstraction to broadcast the number of darts ($999/3$) each worker will toss. Then, it calls the `mreduce` abstraction to receive the number of darts in the circle from each process and applies the `(+)` combining function to sum the values. Finally, it closes its side of the communication with each worker through the `wdone` abstraction estimates $\pi$.

- The `worker` function: Encapsulates the workers' behaviour, responding to the manager's calls and solving the problem in parallel. First, it calls the `wbroadcast` abstraction to receive the number of darts to toss. Then, it calculates the number of darts in the circle using the `calculateDartsInCircle` function and sends the result back to the manager through the `wreduce` abstraction. Finally, in accordance with the manager, it closes its side of the communication with the `wdone` abstraction.

In comparison, our module allows a more focused and condensed implementation with better-defined blocks of code. Additionally, there is no need to define a protocol, as the module predefines a protocol describing all possible communications.

While it imposes some usage rules on developers, this example demonstrates that our module reduces implementation effort and avoids difficulties with linearity and scalability, resulting in fewer lines of code. It provides an environment that enables developers to focus on solving problems in a structured and organized manner.

### 4.1.5 Interprocess communication: Challenges and attempts

We have discussed some limitations of the current implementation, particularly its lack of support for exchanging and manipulating values other than integers or lists of integers. Moving forward,

we can explore some ideas that arose during development.

As previously explained, MPI assigns identifiers (ranks) to concurrent processes, allowing them to differentiate their behaviour based on rank. This enables a combination of interprocess communication through point-to-point and collective communication, where processes in the same group can communicate with each other based on their rank while performing collective operations. Our module, however, does not support this feature, as workers only maintain endpoints to communicate with the manager, inhibiting direct interprocess communication.

Combining both communication types is crucial for addressing a broader range of problems, such as *odd-even sorting* [38] and *finite differences* [46] in parallel, which require data exchange between processes. Currently, our module cannot implement these types of problems. However, given that our work is already based on the MPI specification, we attempted to expand our module's capabilities to include interprocess communication.

In this attempt, we create a shared channel for each process, assigning a shared endpoint ($*?T$) for receiving data and an identifier during initialization. We then use a dictionary-like ADT to map ranks to their respective dual shared endpoint ($*!T$), enabling communication with any process based on its rank. Similar to MPI, we offered point-to-point communication abstractions and built collective communication abstractions on top of these. Additionally, each process could execute the same function and distinguish unique behaviour based on rank, eliminating the manager-workers dichotomy.

Unfortunately, this approach proved to be unviable. The module's framework and its provided abstractions became overly convoluted and complex, resulting in a confusing and unintuitive experience for developers. Furthermore, it was unsafe, as processes could easily cheat, and some synchronization problems appeared, leading to unexpected results. These issues contradicted FreeST's philosophy of keeping communication simple, intuitive, and secure. The complex and time-consuming development ultimately led us to abandon the idea.

## 4.2 The Futures module

The concept of futures introduces a new paradigm for writing and launching asynchronous computations in FreeST. A future typically represents the result of a computation performed asynchronously, offering a different approach than the current methods used in FreeST.

In FreeST, data exchange between parties, including asynchronous computations, revolves around session types and channels. Traditionally, asynchronous computations launched through the `fork` primitive have their return values discarded. If a developer wishes to retrieve the result of an asynchronous computation, they must create a channel between the main thread and the asynchronous computation to exchange that data.

Integrating an abstraction for futures into the FreeST programming language addresses this limitation by eliminating the need for explicit channel creation between the main thread and asynchronous computations, resulting in simpler and more intuitive code. Consequently, developers can launch and manage asynchronous operations more efficiently, promoting a new and more

efficient way of handling asynchronous computations in FreeST.

FreeST, being a purely functional language, emphasizes immutability and pure functions. In this paradigm, the notion of promises, which are writable entities that change state over time, is not well-suited. Instead, FreeST favours constructs that align with its functional nature, where values and computations are predictable and side-effect-free. Thus, we can directly use constructs that handle futures.

### 4.2.1 Design and implementation

To implement futures in FreeST, we benefit from channels which are somewhat analogous to a promise, as they can store a value that will be computed later. Accordingly, the `?a;Wait` protocol represents this component, serving as the representation of a future. In the near future, we anticipate implementing the following session type, as FreeST currently does not yet support polymorphic session types:

```
type Future a = ?a;Wait
```

To launch a new asynchronous computation and create and return a future, we provide the following abstraction:

```
future : (() → a) → ?a;Wait
```

It takes a thunk—a deferred computation or, more specifically, an expression that has not yet been evaluated—and returns a future. It works by creating a new channel and passing `!a;Close`, which is dual to the protocol representing a future, to the forked computation. The new asynchronous computation involves forcing the evaluation of the thunk and then sending its return value to the channel.

The `future` abstraction returns a future from which the result of the asynchronous computation can be retrieved. To achieve this, we provide the following abstraction:

```
block : ?a;Wait → a
```

It receives a future and returns the value it is holding. If the value is not yet available, the current thread will block until it becomes available.

Additionally, we provide a more situational abstraction that delays the execution of a computation until another computation has been completed:

```
delay : (() → a) → (() → b) → ?b;Wait
```

This abstraction takes two thunks and asynchronously chains them, ensuring the first is evaluated before the second. The result of the first computation is discarded, and a future holding the result of the delayed computation is returned.

The implementation details of each abstraction in this module can be found in Appendix A.2.

### 4.2.2 Addressing divide-and-conquer

Parallelizing the divide-and-conquer programming paradigm in FreeST is not straightforward by default. In divide-and-conquer algorithms, a thread must retrieve and merge the results of the following two forked sub-problems. In FreeST, this translates into maintaining a channel for each asynchronous recursive call to retrieve data when results are available. The resulting relationship between processes can be visually represented as follows:

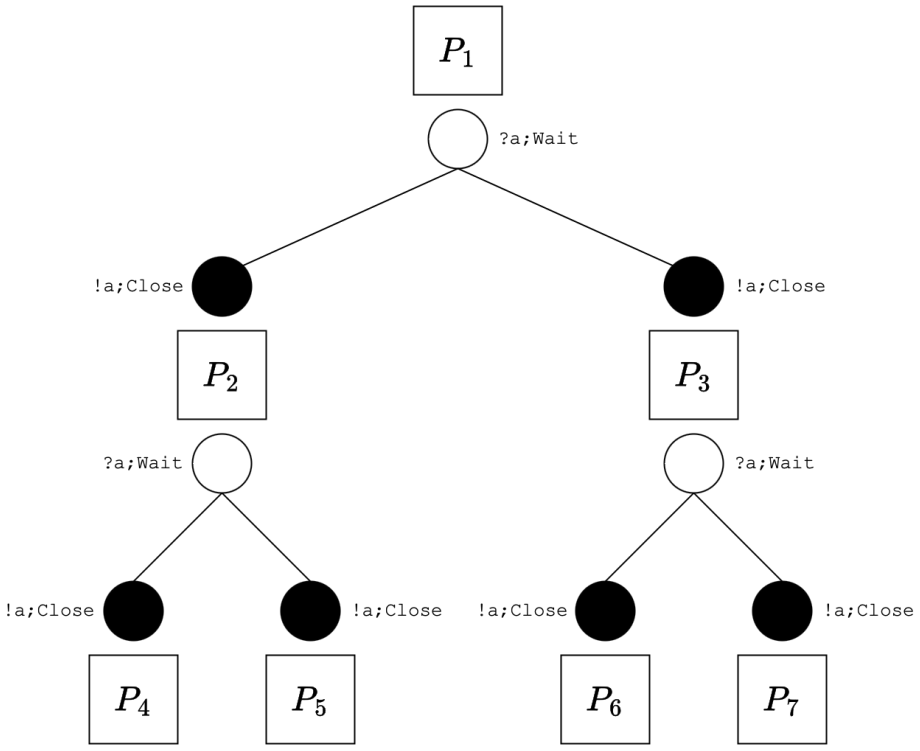



Figure 4.5: Futures - Divide-and-conquer process relationship in FreeST

Thus, implementing divide-and-conquer algorithms is not very intuitive and can become quite complex.

To address this issue, we can use futures to implement divide-and-conquer algorithms. Futures abstract the required channel management, making retrieving an asynchronous process's return value easier. We can divide the problem into two sub-problems, compute them asynchronously using the `future` abstraction, and retrieve the results with the `block` abstraction, combining them to give a solution to the original problem.

### 4.2.3 Implementing the Fibonacci sequence

As discussed in Sections 2.2.3 and 2.4.2, the Fibonacci sequence computation is a classic example of a recursive algorithm that benefits from divide-and-conquer strategies.

This subsection explores how to leverage futures in FreeST to implement a parallel divide-and-conquer version of the Fibonacci. Consider the following implementation:

```
1 pFib : Int → Int → Int
2 pFib n | n == 0     = 0
3        | n == 1     = 1
4        | otherwise = let f1 = future (\_:() → pFib (n - 1)) in
5                      let f2 = future (\_:() → pFib (n - 2)) in
6                      block f1 + block f2
```

To understand how the divide-and-conquer paradigm applies to a concurrent implemention of the Fibonacci sequence using futures, let us break down the `pFib` function:

- **Divide**: In the recursive case, the problem is divided into two sub-problems by creating two futures that concurrently execute the same function. One future computes `pFib (n - 1)` while the other computes `pFib (n - 2)`, representing the computations of Fibonacci's proceeding numbers. This pattern continues recursively until a base case is reached, where `n` is either 0 or 1.

- **Conquer**: In each recursive call, the computation of `pFib (n - 1)` and `pFib (n - 2)` is carried out in parallel, with the `block` abstraction waiting for the results of these futures.

- **Combine**: The results are then combined by summing the values returned by these futures. This approach ensures efficient computations by leveraging concurrent execution, starting from the base cases and moving up to the initial problem.

This example demonstrates the power of futures in facilitating concurrent computation and how they can be seamlessly integrated into divide-and-conquer algorithms to enhance performance in FreeST.

### 4.2.4 Comparison to ForkJoin and challenges

ForkJoin is a concurrent divide-and-conquer approach centred on the mechanisms of *forking* and *joining*. Our module implements abstractions that function as analogous mechanisms: we use the `future` abstraction similarly to the `fork` abstraction, and the `block` abstraction akin to the `join` abstraction. This enables the implementation of divide-and-conquer algorithms in a similar manner. However, ForkJoin follows specific scheduling mechanisms and thread management techniques, which reveal fundamental differences compared to our module's approach to divide-and-conquer.

In Section 2.4.2, we discussed the ForkJoin framework's work-stealing scheduler, wherein a thread with an empty scheduler queue attempts to take a task (lightweight thread) from another thread. In contrast, FreeST does not implement its own scheduler; instead, it relies on Haskell's *work-pushing* scheduler [42]. Work-pushing is a technique in which, when a thread's queue has more than one task and there are idle threads, it distributes some tasks to other idle threads. This difference is minor since both techniques behave very similarly, and it does not negatively affect our goal. On the contrary, FreeST enables a similar concurrent divide-and-conquer approach to ForkJoin.

#### Task granularity

Regarding task granularity control, FreeST currently lacks the necessary functionalities to implement most mechanisms. The only two viable options are a "limitation number", adjusted to the specific problem, and the *MaxLevel* mechanism, which developers can easily implement through an additional parameter in the function. However, the remaining mechanisms mentioned in Section 2.4.2 are either overly complicated and unintuitive to implement (e.g., *MaxTasks*, which is implementable through a shared channel) or simply unattainable due to the absence of primitives enabling developers to retrieve real-time information about the scheduler's status, such as the number of active threads, their queues, or the total number of tasks created.

To address this limitation in future work, we have identified an opportunity stemming from the lack of control over the scheduler. Our proposal involves integrating bindings to specific Concurrent Haskell primitives, thereby enhancing FreeST's capabilities and enabling the implementation of more task granularity control mechanisms. We have initiated this enhancement by creating a *FreeST Enhancement Proposal* (FEP) in the project's repository, which can be found in Appendix B.

## 4.3 The Streams module

FreeST employs channels as the fundamental mechanism for data exchange among agents. These channels are typically used for the sequential and continuous exchange of data elements, allowing them to be read from or written to in a stream-like fashion. Consequently, stream programming emerges as a fundamental and often implicit programming paradigm within FreeST.

To formalize this concept, we can define a recursive protocol that describes the transmission of elements of a specified type until one of the participating agents decides to terminate the transmission:

```
type IntStream = ⊕{More: !Int;IntStream, Done: Wait}
```

This protocol represents a stream of integers. Agents holding an endpoint of type `IntStream` can either select the `More` operation to continue feeding integers to the stream or the `Done` operation to conclude the transmission.

Moreover, FreeST's concurrent nature empowers agents to process data asynchronously, thereby enhancing the efficiency and responsiveness of data processing tasks. For instance, consider a scenario where one entity generates integers into a stream while another retrieves and prints these integers. We can implement this scenario as follows:

```
1 main : ()
2 main = let (w, r) = new @IntStream () in
3         fork (\_:() → intSource w);
4         intPrinter r
```

The `new` primitive creates a stream and returns endpoints `w` (writer) and `r` (reader) of types `IntStream` and `dualof IntStream`, respectively. Here, `intSource` concurrently feeds inte-

gers into the stream using the `w` endpoint, while `intPrinter` sequentially retrieves and prints integers at the same time using the `r` endpoint.

Therefore, developing a dedicated streams module in FreeST to define and abstract programming patterns for managing data streams among distinct agents could significantly enhance the stream programming experience. By establishing standardized streaming practices tailored to the language's characteristics, such a module would streamline programming workflows, inspire innovative problem-solving approaches, and alleviate issues associated with repetitive and complex code.

### 4.3.1 Design and implementation

The first step in developing a standardized streams module involves establishing the protocol for stream programming:

```
type OStream = ⊕{More: !Int;OStream, Done: Wait}
type IStream = dualof OStream
```

This protocol introduces a naming convention to differentiate behaviours related to streams: `O` (Output) for sending integers into a stream and `I` (Input) for retrieving integers from a stream. Although this mirrors our previous protocol, descriptive naming enhances clarity.

Similar to the Parallel module, our stream protocol currently limits communications to integer exchanges due to FreeST's lack of support for polymorphic session types.

**Filters** In the context of these session types, any function that handles streams and processes its data through these types can be considered a filter. Essentially, a filter is any function that maintains an `OStream` endpoint to send integers into a stream and/or an `IStream` endpoint to receive integers from a stream. For instance, in the previous example, both `intSource` and `intPrinter` are filters.

**Pipelines** Pipelines are formed by linking two or more filters with streams. To link two filters, one must hold an `OStream` endpoint and the other an `IStream` endpoint. The second filter can then link with a third stream through the same process, and so on. This chaining process continues, with each subsequent filter linking to the next through a one-to-one relationship between endpoints. Unfortunately, a universal `pipeline` abstraction is not feasible due to the impossibility of defining a consistent signature for all possible filter definitions. Nonetheless, the previous example successfully demonstrates a pipeline consisting of two filters.

Building upon the `OStream` and `IStream` session types, we designed several abstractions and patterns to create a robust environment for stream programming.

To facilitate basic but essential stream manipulation, we developed the following auxiliary abstractions:

```
sendS : Int → OStream → OStream
```

```
waitS : OStream → ()

forward : IStream → OStream 1→ OStream

fromList : [Int] → OStream → ()

toList : IStream → [Int]
```

Each abstraction serves the following purpose:

- `sendS` and `waitS` abstract the `OStream` protocol's operations. The former sends an integer into a given stream and returns the continuation of the protocol, while the latter waits for the stream to be closed.
- `forward` sends the contents of one stream into another, closing the former and returning the continuation of the latter.
- `fromList` and `toList` complement each other. The former feeds the contents of a list into a given stream, whereas the latter (re)constructs a list from the contents of a stream.

These abstractions form the foundation of our module, enabling straightforward communication patterns that can be useful for basic serialization and deserialization tasks.

#### Splitters and joiner

To leverage FreeST's potential and enhance the flexibility and expressiveness of our streams module, we implemented patterns inspired by constructs from the StreamIt programming language (discussed in Section 2.4.3). These patterns enable the composition of asynchronous computations, allowing for more complex stream processing pipelines.

We adapted StreamIt's *SplitJoin* construct to the FreeST programming language. *Splitters* distribute data between two streams, automating the process of linking a `OStream` endpoint to two `IStream` endpoints without requiring the programmer to manage data distribution manually. Correspondingly, the joiner merges the contents of two streams and forwards them into another stream.

We offer three different types of splitter abstractions, each with similar signatures and behaviours:

```
splitSDup : IStream → OStream 1→ OStream 1→ ()

splitSAlt : IStream → OStream 1→ OStream 1→ ()

splitSWith : (Int → Bool) → IStream → OStream 1→ OStream 1→ ()
```

- `splitSDup` duplicates elements from a stream and sends them to two different streams.
- `splitSAlt` alternates elements between two streams, distributing data in a round-robin fashion.

- `splitSWith` uses a predicate function to determine the distribution of elements. It sends the elements that satisfy the predicate to the first stream and those that do not to the second stream.

For the joiner, we provide the following abstraction:

```
joiner : IStream → IStream 1→ OStream 1→ ()
```

This abstraction functions similarly to the joiner mechanism in the StreamIt programming language in the sense that it merges and forwards the contents of two streams into another stream in a round-robin manner. Although splitters and joiners can be used together, they are independent abstractions and do not necessarily need to be used in conjunction.

### Higher-order functions with streams

Finally, this module provides some higher-order filters for data manipulation within streams:

```
mapS : (Int → Int) → IStream → OStream 1→ ()

foldlS : (Int → Int → Int) → Int → IStream → Int

foldrS : (Int → Int → Int) → Int → IStream → Int

filterS : (Int → Bool) → IStream → OStream 1→ ()
```

- `mapS` applies a function to each element in a stream and feeds the result into the output of another stream.

- `foldlS` and `foldrS` use a combining function to systematically combine all elements in a stream, returning the result.

- `filterS` filters elements in a stream based on a given predicate and feeds the passing elements into another stream.

Consider the previously discussed scenario with an integer generator and a printer. We can use `filterS` to only print the even numbers:

```
1 main : ()
2 main = let r  = forkWith intSource in
3        let r2 = forkWith (filterS even r) in
4        intPrinter r
```

In this setup, `intSource` and `filterS` run concurrently. `intSource` generates integers, which `filterS` then processes to retain only even numbers, which `intPrinter` prints.

See Appendix A.3 for the detailed implementation of this module's abstractions and definitions.

### 4.3.2 Implementing the quicksort algorithm

*Quicksort* [30, 21] is an efficient and well-known sorting algorithm that follows the divide-and-conquer strategy to sort a list of elements in ascending or descending order.

The quicksort algorithm works as follows:

1. **Choose a pivot**: Select an element from the list as the pivot. In the simplest implementation, the first or last element are often chosen as pivot.

2. **Partitioning**: Reorder the list such that all elements less than the pivot come before it, and all elements greater than or equal to the pivot come after it. The pivot is now in its final sorted position.

3. **Recursion**: Recursively apply quicksort (steps 1 and 2) to the two sublists on either side of the pivot until reaching the base case (e.g., a sublist contains zero or one element).

4. **Merge**: As the recursive calls return, concatenate the sorted sublists around the pivot to form the fully sorted list.

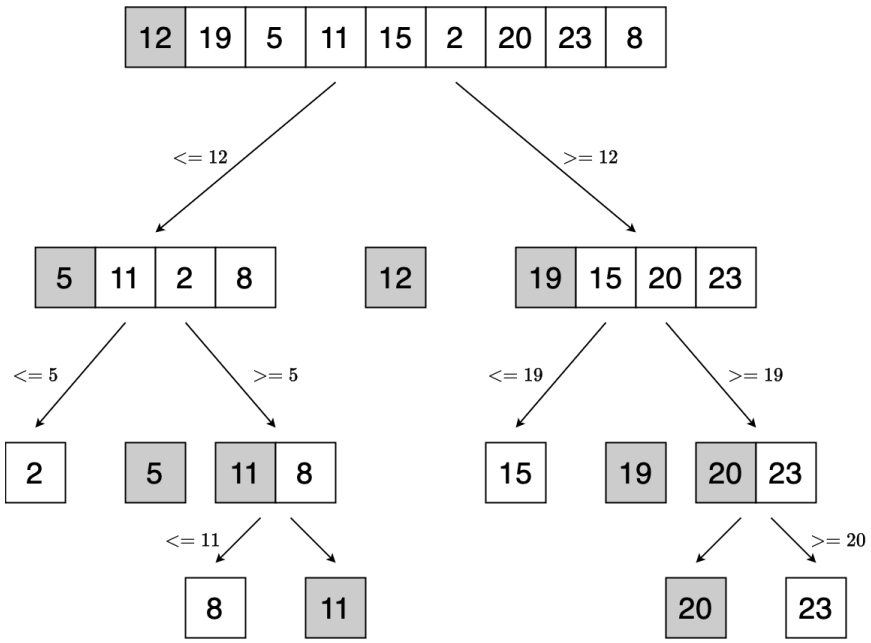



Figure 4.6: Streams - Example of the quicksort algorithm

#### Implementation

Assume the existence of the `forkWith2` abstraction, a version of `forkWith` that creates two channels and launches a thread.

To demonstrate the power and usefulness of our contribution, we present an implementation of quicksort using streams:

```
sqsort : IStream → OStream 1→ ()
sqsort (Done i) o = wait i; closeS o
sqsort (More i) o = let (x, i) = receive i in
                    let (i1, i2) = forkWith2 (splitSWith (<x) i) in
                    let i3 = forkWith (sqsort i1) in
                    let i4 = forkWith (sqsort i2) in
                    o ▷ forward i3 ▷ sendS x
                      ▷ forward i4 ▷ closeS

qsort : [Int] → [Int]
qsort xs = let i1 = forkWith (fromList xs) in
           let i2 = forkWith (sqsort i1) in
           toList i2
```

The `qsort` function comprises the essential building blocks of this implementation. It first uses `fromList` to concurrently feed the list of integers into a stream. It then calls `sqsort` to sort these numbers concurrently and finally converts the sorted stream integers back to a list using `toList`.

`sqsort` filter is the core component of this implementation. It takes an input stream endpoint to receive data from an existing stream and an output stream endpoint to feed data into another stream. Based on the previous definition of the quicksort algorithm, `sqsort` follows these steps:

1. **Choose a pivot**: Receives the head number from the input endpoint of the stream and declares it as the pivot.

2. **Partitioning**: Uses the `splitSWith` splitter to partition the stream into two new streams via the `forkWith2` abstraction. Numbers less than the pivot go into one stream, while the rest go into the other.

3. **Recursion**: Divides the problem into two smaller problems by recursively applying itself to the input endpoint of the split streams. Each call processes half of the data concurrently, repeating this process until each stream contains only one element.

4. **Merge**: Each call to `sqsort` also takes an output stream endpoint to communicate with the previous computation. Each computation receives the data from its two subproblems, uses its output endpoint to send the received data from the first subproblem, sends its selected pivot, and finally sends the received data of the second subproblem, respectively. This merge maintains the order, with lower elements on the left and higher ones on the right.

This divide-and-conquer approach creates a tree-like structure, similar to the one illustrated in Fig. 4.5.

### 4.3.3 Comparison to the StreamIt programming language

Attentive readers may notice that our approach to stream processing in this work diverges somewhat from the "storytelling" method throughout the document. The recurring discussion of Monte Carlo and Fibonacci culminated in their implementations using the Parallel and Futures modules,

respectively. For streams, we chose to delve into two distinct problems: the moving average, discussed in the StreamIt section (Section 2.4.3), and the quicksort algorithm in this section. These examples highlight the contrasting approaches to stream processing between our module for FreeST and StreamIt.

The moving average is a typical example in StreamIt, leveraging the `peek` operation to access stream values without removing them. In contrast, FreeST does not support such channel interaction features since accessing a channel value requires strict protocol adherence, consuming the linear endpoint entirely. Essentially, FreeST lacks the capability to "peek" into channel content without consuming it. Consequently, this limitation makes implementing a moving average in FreeST significantly different from StreamIt's approach and fails to demonstrate the potential of splitters.

To demonstrate the strengths of our module, we implemented the quicksort algorithm, known for its concurrency challenges. This algorithm aligns naturally with our module, contrasting with the complexities StreamIt faces due to quicksort's divide-and-conquer nature, requiring creative and intricate use of its constructs.

In conclusion, our module provides a lower-level approach to stream processing, suited to FreeST's functional characteristics (e.g., immutability and pure functions), features (e.g., linearity, session types), and overall use cases. Conversely, StreamIt provides high-level constructs optimized for parallel stream processing, incorporating imperative elements such as loops and state management. These differences shape distinct programming environments: our module abstracts common stream logic within FreeST's programming practices, while StreamIt is a fully-fledged language optimized specifically for stream programming. Thus, our module and the StreamIt programming language are designed to achieve distinct goals and adapt differently to various problems benefitting from asynchronous stream processing.

# Chapter 5

# Evaluation

FreeST is not simple. It is an embryonic language lacking a comprehensive suite of tools suitable for real-world applications. Its unique characteristics demand substantial domain knowledge and effort to develop parallel and concurrent systems effectively.

To address these challenges and adapt FreeST to the demands of concurrent programming, we have developed and integrated several modules that address well-known programming paradigms, abstracting their patterns. These modules are tailored to FreeST's specific needs, thereby expanding its capabilities and making it more accessible and practical for developers. Our contributions aim to ease the learning curve for developing parallel and concurrent programs, transforming FreeST into a more versatile tool.

To evaluate the effectiveness of these contributions, we conducted three surveys—each focused on one of the developed modules—and a Lines of Code (LoC) comparison between examples implemented in "pure" FreeST and those using our modules. These surveys were designed to gather feedback from FreeST programmers regarding their experiences and perceptions of the modules. The ultimate goal of this evaluation is to validate our contributions by assessing their relevance, usefulness, and accessibility in the context of parallel and concurrent programming in FreeST. Moreover, we did not conduct a performance analysis of these modules, as such an analysis is orthogonal to our primary objective of evaluating usability and practicality.

## 5.1 Surveys

The target audience for these surveys comprises the FreeST community, which includes students and professors highly familiar with the FreeST programming language. We chose such a restrictive target audience to ensure feedback from individuals with significant experience with the language, allowing for constructive feedback and deeper informed insights into our contributions.

The total number of participants remains uncertain due to an uneven number of responses across the surveys: five for the Parallel module, five for the Futures module, six for the Streams module. However, we estimate a total of six participants. This sample size, while limited, still provides valuable feedback from a knowledgeable and experienced group within the FreeST community.

### 5.1.1 Design

All our surveys follow a structured format to ensure consistency and comprehensiveness:

1. **Introduction**: Provides a brief overview of the module's purpose, the specific subjects it addresses, and the problems it aims to solve within the context of FreeST.

2. **Module design**: Describes the module's design decisions and implementation.

3. **Examples**: Presents practical examples demonstrating the application of the module to real-world parallel and concurrent programming problems.

4. **Exercise**: Tasks participants with implementing a specific exercise using the module. This hands-on activity evaluates the module's usability, effectiveness, and ease of integration into practical programming tasks.

5. **Feedback request**: Gathers feedback on the module's relevance, usability, functionality, and potential areas for improvement.

As emphasized throughout this work, each module links to a specific programming paradigm: the Parallel module focuses on data parallelism and embarrassingly parallel problems, the Futures module addresses futures and divide-and-conquer strategies, and the Streams module deals with stream programming.

After introducing participants to each module's design, implementation, and goals, we seek to analyze their reactions and adaptability through exercises that involve solving problems related to the respective programming paradigm using the corresponding module. Additionally, we incorporate open-ended questions for participants to articulate any difficulties or challenges encountered during these exercises. This approach is crucial for evaluating each module's effectiveness in achieving its intended goals and addressing the specific programming paradigms linked to them.

#### Feedback questions

To gather feedback, we request participants to evaluate their experience based on several parameters on a scale from 1 to 10:

1. **Accessibility**: How easy is the module to use and understand its workflow and programming environment?
   *Scale: 1 (Hard) to 10 (Easy)*

2. **Compatibility**: How well does the module align with the features (e.g., concurrency, linearity, and session types) and philosophy (e.g., simplicity and security) of the FreeST programming language?
   *Scale: 1 (Incompatible) to 10 (Compatible)*

3. **Relevance**: How relevant is the integration of this module into the FreeST programming language?
   *Scale: 1 (Irrelevant) to 10 (Relevant)*

However, the parameters evaluated vary from survey to survey:

- **Parallel module**: Evaluated according to parameters 1 and 3, as this module creates a complete working environment that addresses embarrassingly parallel problems with an implicit workflow.
- **Futures module**: Evaluated according to parameters 2 and 3 since it comprises simple abstractions that wrap existing FreeST primitives but does not invoke a specific workflow or form a distinct programming environment.
- **Streams module**: Evaluated according to parameters 1, 2 and 3, as it abstracts certain programming patterns in FreeST and encourages stream programming patterns.

Finally, we included an open-ended question to allow participants to provide constructive feedback and offer deeper insights by pointing out additional observations or questions.

See appendices C.1, C.2, and C.3 for the integral version of these surveys.

### 5.1.2 Results and analysis

We evaluate the effectiveness of our modules through two primary components: an analysis of participant performance in implementing each survey's exercise and an examination of the feedback gathered in each survey's concluding section.

#### Exercises analysis

**Parallel module's exercise** All participants completed this exercise successfully, demonstrating a proficient grasp of the module's functionalities and programming environment. While some implementations exhibited minor coding practice issues, these did not hinder the overall completion of the task. This consensus underscores the module's effectiveness in enabling participants to address embarrassingly parallel problems within FreeST.

**Futures module's exercise** Similarly, all participants exhibited a proficient understanding of leveraging FreeST's futures to implement divide-and-conquer strategies effectively. This outcome underscores the module's effectiveness in equipping participants with the necessary tools for approaching the divide-and-conquer paradigm within FreeST.

**Streams module's exercise** The exercise assigned to the Streams module presented more challenges compared to the others. While most participants understood how to use the module's features to implement the exercise, there were notable irregularities. One participant struggled to grasp why this scenario benefited from our abstractions and encountered difficulties dealing with

linearity. Another participant overlooked launching asynchronous computations, undermining the intended advantages of using FreeST. These observations suggest opportunities for enhancing the module's effectiveness in achieving its goals and optimizing the survey's efficacy in explaining and demonstrating its capabilities.

### Feedback analysis

By calculating each question's average rating, we can better understand whether each module achieved its goals. Using the evaluation parameter distribution from the previous subsection and the averages obtained from the ratings, we present the bar chart in Fig. 5.1.

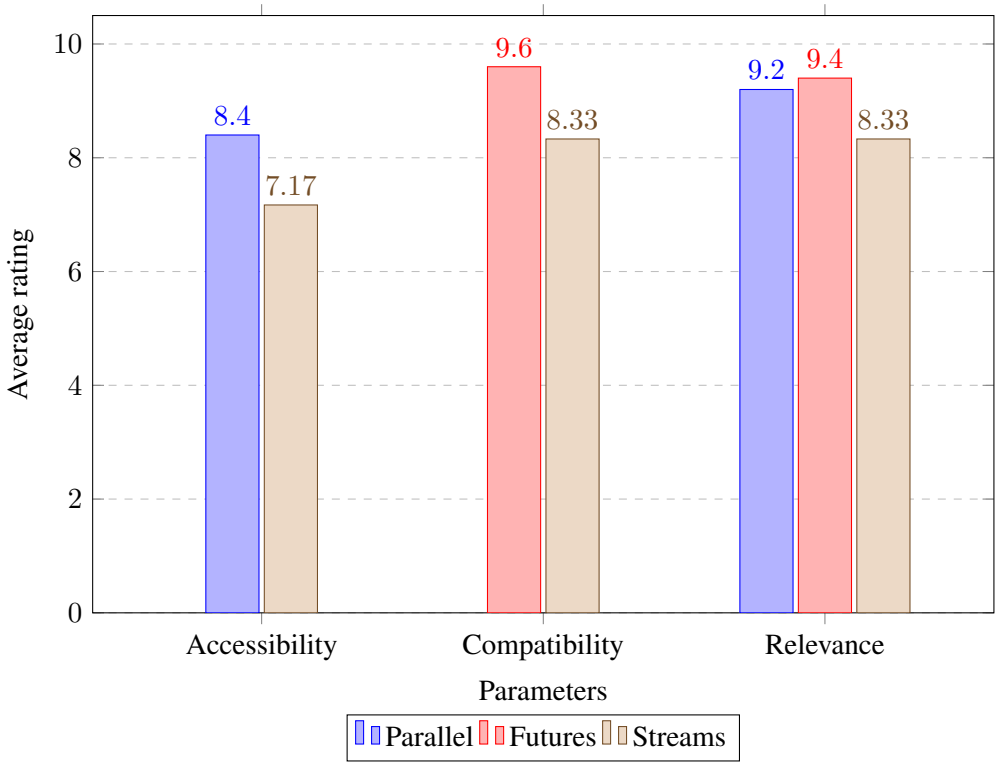



Figure 5.1: Comparison of survey results across all modules

We acknowledge that individual differences in background and expertise influence participants' reactions and adaptability. Despite this variability, the overall results depicted in Fig. 5.1 are very positive. We received overwhelmingly positive feedback overall, with only one major negative feedback per survey, preventing the results from achieving consistent averages of 9-10.

The Parallel and Futures modules received high ratings, confirming their effectiveness in addressing embarrassingly parallel problems and implementing divide-and-conquer strategies, respectively. The slightly lower ratings for the Streams module suggest some challenges in understanding and applying its features, as noted in the exercise analysis.

The positive feedback and constructive criticism provide valuable insights into how we can improve the modules and the surveys. The open-ended responses will further illuminate specific areas for enhancement.

**Parallel module's insights** The feedback on the Parallel module was very positive overall, as reflected by the high ratings in the evaluation. Participants acknowledged the module's promise and effectiveness in facilitating addressing embarrassingly parallel problems and praised its ability to abstract away boilerplate code, allowing them to focus on domain-specific logic, thus meeting its primary goals. Some feedback emphasized the need for comprehensive documentation to ease the learning curve, suggesting that clear explanations of each primitive's purpose would enhance understanding. While a few participants felt that session types were not essential and recommended more intuitive naming conventions for operations, these suggestions aimed at further improving an already effective module. Additionally, the desire for more higher-level functions indicates a recognition of the module's potential for even greater usability. Overall, the positive reception emphasizes the module's success in achieving its goals while providing valuable insights for continuous improvement.

**Futures module's insights** The feedback on the Futures module reflects a positive reception overall, correlating well with the high ratings evaluation. Participants found the module extremely useful for implementing solutions using the divide-and-conquer paradigm, appreciating its simplicity and compactness. They highlighted the module's effectiveness in modelling classic concurrent programming primitives through FreeST's session-typed channels, which aids new users in grasping session types. Suggestions for improvements, such as enhancing type inference to simplify lambda expressions and introducing syntactic sugar for thunks, primarily address limitations inherent in FreeST rather than shortcomings specific to the module. These insights underscore the module's value in bridging familiar programming paradigms with FreeST's capabilities, suggesting opportunities for future enhancements to streamline usability and improve user experience.

**Streams module's insights** The feedback on the Streams module reflects positive sentiment, consistent with the ratings in the evaluation. Participants appreciated its ability to simplify boilerplate code for concurrent programming while promoting stream programming patterns. However, some identified challenges for new users in manually creating and managing channel endpoints, which are essential in FreeST's design. This aligns with participants' comments such as "It took me a while to understand the behaviour of each function and the example provided since it uses a lot of forks" and "I do not want to create processes manually.". We argue that this low-level design decision fits FreeST better and introduces flexibility to accommodate various use cases, asserting that abstracting channel creation and forking would overly restrict user control. Some participants identified potential applications in data analysis and AI, such as neural networks and image processing. Features like `forkWith2` were highlighted for their utility, suggesting potential for future additions to further leverage the module's capabilities. Moreover, FreeST's limitations constrain the module's usability, such as the lack of type operators and support for lists with elements other than integers, which impact its widespread adoption.

The integral answers to the open-ended questions can be found in appendices D.1, D.2, and D.3.

## 5.2 Lines of Code comparison

To assess the efficiency of the Parallel module in simplifying the implementation of embarrassingly parallel problems, we compared the *Lines of Code* (LoC) between implementations using the pure FreeST and the Parallel module.

We implemented three examples using both approaches:

- *Parallel average*: Distributes a list across several processes (scatter), sums the sublists in parallel, then receives and sums the results (reduce), and calculates the average sequentially.
- *Scalar product*: Distributes two lists across several processes (two scatters), calculates the scalar product in parallel, then receives and sums the results (reduce) sequentially.
- *Monte Carlo*: A broadcast followed by a reduce (extensively discussed throughout this document; you should be familiar with it by now).

The bar chart below compares the LoC between the two implementations:

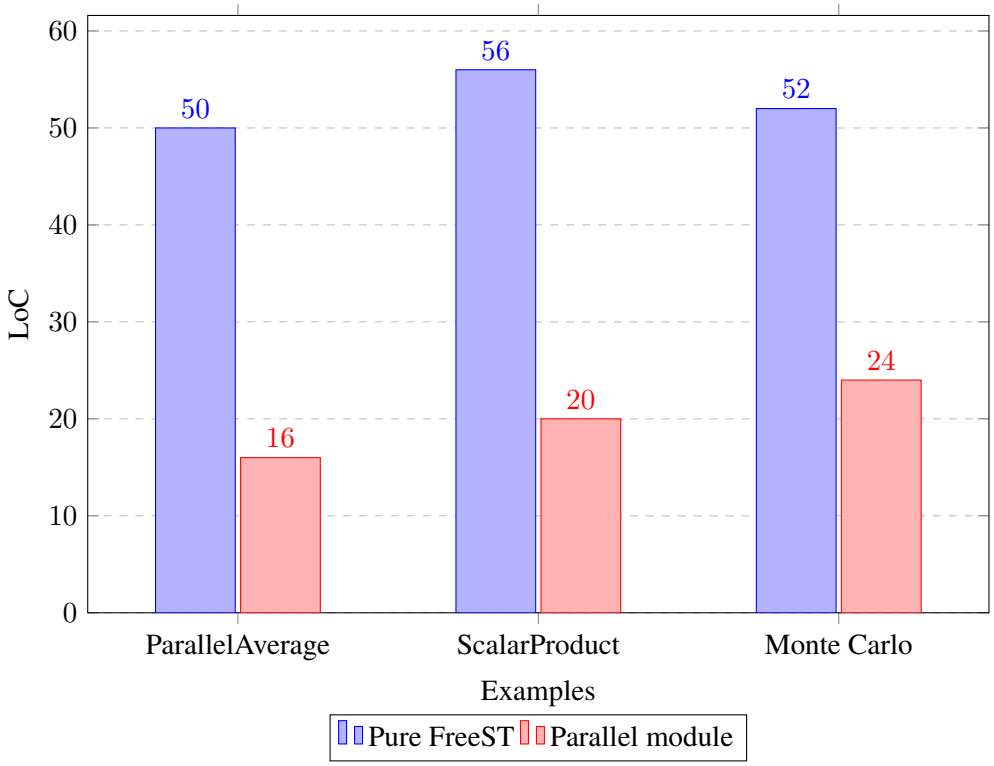



Figure 5.2: LoC comparison for pure FreeST and Parallel module implementations

These results highlight a significant LoC reduction when using the Parallel module compared to pure FreeST implementations. The module consistently reduces LoC by approximately 30 lines

per example, demonstrating its ability to abstract boilerplate code and streamline development. This reduction is an absolute value around 30 lines—varying slightly depending on the operations performed—rather than a scaling percentage. This abstraction allows developers to focus on the core logic of each problem, freeing them from the burden of the repetitive task of implementing infrastructure code to address embarrassingly problems. Moreover, this programming environment relieves developers from dealing with linearity, significantly reducing necessary effort.

In the case of the parallel average example, further optimization can be achieved by implementing the entire exercise using the `nfoldl` higher-order function, dramatically reducing the LoC from 50 to just 4.

We have determined that LoC is not a suitable metric for evaluating the Futures and Streams modules. The Futures module simply provides wraps for the `fork` primitive (or `forkWith`) and does not significantly affect LoC in its use-case implementations. Similarly, the Streams module abstracts the stream programming paradigm inherent to the FreeST language through definitions and abstractions. Unlike the Parallel module, it acts more like a helpful toolkit for a loosely defined set of problems rather than providing a programming environment with strict practices and patterns for a well-defined set of problems. Therefore, direct comparisons based on LoC are not meaningful.

# Chapter 6

# Conclusion and future work

FreeST is a concurrent programming language featuring context-free session types. However, its applicability to real-world scenarios remains limited, and its interactions with various computing concepts are still largely unexplored. Although some works have been done to extend the language and investigate its interactions with concepts such as shared channels [**?**] and subtyping [48], its potential in easing parallel and concurrent programming is not yet fully understood.

In this thesis, we explored the interaction of FreeST and session types with five prominent parallel and concurrent programming concepts and paradigms: data parallelism and embarrassingly parallel problems, futures and divide-and-conquer, and streams. To support these concepts, we proposed the integration of three new modules into the language, thereby facilitating parallel and concurrent programming in FreeST.

We verified that FreeST struggled with addressing embarrassingly parallel problems due to difficulties with linearity, state management, and overly complex, repetitive code. To mitigate these challenges, we developed the Parallel module, inspired by MPI's collective communication operations. This module provides a specialized programming environment designed to handle embarrassingly parallel problems effectively by reducing boilerplate code and allowing developers to concentrate on the problem itself.

Although FreeST includes the fork primitive for launching asynchronous computations, it lacks a straightforward way to retrieve their results. To resolve this, we introduced the Futures module, which simplifies the retrieval of asynchronous computation results and facilitates the divide-and-conquer paradigm in a manner similar to ForkJoin, which was previously challenging and unintuitive without futures.

Lastly, we identified the inherent concurrent stream programming nature in FreeST. We proposed the Streams module to promote more structured and well-defined programming practices. This module abstracts stream-related behaviour and incorporates features inspired by the StreamIt programming language.

Developing these modules required innovative and rigorous brainstorming to leverage FreeST's unique features effectively. To evaluate our contributions, we conducted three surveys——one per module——assessing various metrics on a scale of 1 to 10 and including open-ended questions for further feedback. Despite identifying several limitations within each module, mainly due to

inherent FreeST constraints, our surveys received very positive feedback, indicating a welcoming reception and validating the effectiveness of our contributions.

## Future work

While FreeST shows potential in enhancing parallel and concurrent programming through the proposed modules, further research is needed to refine these modules and explore additional programming paradigms, patterns, and applications.

One of the avenues to future work consists of enhancing the proposed modules, leveraging the limitations identified during their development (discussed in Sections 4.1.5, 4.2.4, and 4.3.3) as guidelines, and aiming to achieve more consistent and well-rounded modules. Addressing these limitations would create more flexible modules, allowing broader use cases. Additionally, we should consider the insights provided by the survey participants, who pointed out several opportunities for improvement, particularly concerning usability.

In addition to the paradigms explored in this work, we encourage further investigation into other programming paradigms and concepts to significantly extend FreeST's parallel and concurrent capabilities. Examples include *Functional Reactive Programming* (FRP) [44], the *actor model* [29, 28], and *pseudorandom number generation* (PRNG) [15]. Exploring these paradigms will not only deepen FreeST's functionality but also uncover new potential applications.

# Bibliography


[1] Scala's documentation on Futures and Promises. https://docs.scala-lang.org/overviews/core/futures.html. Last accessed: July 2024.

[2] The Erlang programming language. https://www.erlang.org/. Last accessed: July 2024.

[3] The FreeST programming language. https://freest-lang.github.io. Last accessed: July 2024.

[4] The Go programming language. https://go.dev/. Last accessed: July 2024.

[5] The Haskell programming language. https://www.haskell.org. Last accessed: July 2024.

[6] The Java programming language. https://docs.oracle.com/en/java/. Last accessed: July 2024.

[7] The Rust programming language. https://www.rust-lang.org. Last accessed: July 2024.

[8] The StreamIt programming language. http://groups.csail.mit.edu/cag/streamit/index.shtml. Last accessed: July 2024.

[9] Streamit cookbook. http://groups.csail.mit.edu/cag/streamit/papers/streamit-cookbook.pdf, 2006.

[10] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming Systems: The What, Where, When, and how of Large-scale Data Processing*. O'Reilly, 2018.

[11] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Freest: Context-free session types in a functional language. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, 2019.

[12] David R. Anderson, Dennis J. Sweeney, Thomas A. Williams, Jeffrey D. Camm, and James J. Cochran. *Statistics for Business & Economics*. Cengage Learning, 2016.

[13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Dallas, TX, 2 edition, 2013.

[14] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. In James Low, editor, *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, USA, August 15-17, 1977*. ACM, 1977.

[15] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – part 1: General (revision 3). *NIST Special Publication Revision*, 01 2005.

[16] Diogo Barros, Andreia Mordido, and Vasco T. Vasconcelos. Shared channels on context-free session types. 2023.

[17] Jonathan Beard. A short intro to stream processing. `https://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html`. Last accessed: July 2024.

[18] Jim Blandy, Jason Orendorff, and Leonora F S Tindall. *Programming Rust*. O'Reilly Media, Sebastopol, CA, 2 edition, June 2021.

[19] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. Evaluation of openmp task scheduling algorithms for large NUMA architectures. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. Springer, 2014.

[20] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, New York, NY, USA, 1963. Association for Computing Machinery.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[22] Katherine Cox-Buday. *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, Inc., 1st edition, 2017.

[23] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[24] Alcides Fonseca and Bruno Cabral. Evaluation of runtime cut-off approaches for parallel programs. In *High Performance Computing for Computational Science - VECPAR 2016 - 12th International Conference, Porto, Portugal, June 28-30, 2016, Revised Selected Papers*. Springer, 2016.

[25] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, (2-3), 2005.

[26] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[27] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[28] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, (3), 1977.

[29] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*. William Kaufmann, 1973.

[30] Charles A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, (7), 1961.

[31] Charles A. R. Hoare. Communicating sequential processes. *Commun. ACM*, (8), 1978.

[32] Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. Springer, 1993.

[33] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. Springer, 1998.

[34] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, (1), 2016.

[35] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. Concurrent haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. ACM Press, 1996.

[36] Wes Kendall, Dwaraka Nath, and Wesley Bland. MPI Tutorial. `https://mpitutorial.com`. Last accessed: July 2024.

[37] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[38] Sivaramakrishnan Lakshmivarahan, Sudarshan K. Dhall, and Leslie L. Miller. Parallel sorting algorithms. *Adv. Comput.*, 1984.

[39] Doug Lea. *Concurrent programming in Java - design principles and patterns*. Addison-Wesley-Longman, 1997.

[40] Doug Lea. A java fork/join framework. In Dennis Gannon and Piyush Mehrotra, editors, *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*. ACM, 2000.

[41] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. Springer, 2011.

[42] Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. ACM, 2009.

[43] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[44] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In Manuel M. T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002*. ACM, 2002.

[45] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[46] César Santos, Francisco Martins, and Vasco Thudichum Vasconcelos. Deductive verification of parallel programs using why3. In *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015*, 2015.

[47] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018.

[48] Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. In *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[49] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition*. Pearson Education, 2008.

[50] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. ACM, 2016.

[51] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Springer, 2002.

# Appendix A

# Modules' implementation code

## A.1 Parallel module

```
module Parallel where

import List

-----------------------------------------------------------------------

-- # ManagerStream Protocol & Comm

type ManagerStream = ⊕{ Broadcast: ![Int]
                      , Scatter  : ![Int]
                      , Gather   : ?[Int]
                      , Reduce   : ?Int
                      , Done     : Wait} ; ManagerStream

type WorkerStream = dualof ManagerStream

data Comm = WNil () | Worker ManagerStream Comm

-----------------------------------------------------------------------

-- # Matching the protocol for each operation from the workers' side

wbarrier : WorkerStream → WorkerStream
wbarrier c = let (_, c) = c ▷ wgather [] ▷ wbroadcast in c

wbroadcast : WorkerStream → ([Int], WorkerStream)
wbroadcast (Broadcast c) = receive c

wscatter : WorkerStream → ([Int], WorkerStream)
wscatter (Scatter c) = receive c

wgather : [Int] → WorkerStream → WorkerStream
wgather xs (Gather c) = send xs c

wreduce : Int → WorkerStream → WorkerStream
wreduce n (Reduce c) = send n c

wallgather : [Int] → WorkerStream
```

```
39                     → ([Int], WorkerStream)
40 wallgather xs c = c ▷ wgather xs ▷ wbroadcast
41
42 wallreduce : Int → WorkerStream
43                  → ([Int], WorkerStream)
44 wallreduce xs c = c ▷ wreduce xs ▷ wbroadcast
45
46 wdone : WorkerStream → ()
47 wdone (Done c) = close c
48
49 ----------------------------------------------------------------------
50
51 -- # Matching the protocol for each operation from the manager's side
52
53 mbarrier : Comm → Comm
54 mbarrier ws = let (_, ws) = mgather ws in
55               mbroadcast [] ws
56
57 mbroadcast : [Int] → Comm → Comm
58 mbroadcast _   (WNil _)       = WNil ()
59 mbroadcast xs (Worker w ws) = Worker (w ▷ select Broadcast ▷ send xs)
60                                 (mbroadcast xs ws)
61
62 mscatter : [Int] → Comm → Comm
63 mscatter xs ws = let (wsl, ws) = commLength ws in
64                  mscatter' wsl (length xs) xs ws
65 -- where
66 mscatter' : Int → Int → [Int] → Comm → Comm
67 mscatter' _ _   _   (WNil _)       = WNil ()
68 mscatter' n xsl xs (Worker w ws) =
69     let chunk = xsl / n + (if mod xsl n /= 0 then 1 else 0) in
70     let (xs, ys) = splitAt chunk xs in
71     Worker (w ▷ select Scatter ▷ send xs) $
72             mscatter' (n - 1) (xsl - chunk) ys ws
73
74 mgather : Comm → ([Int], Comm)
75 mgather (WNil _)       = ([], WNil ())
76 mgather (Worker w ws) = let (xs, w) = w ▷ select Gather ▷ receive in
77                         let (ys, ws) = mgather ws in
78                         (xs ++ ys, Worker w ws)
79
80 mreduce : (Int → Int → Int) → Int → Comm → (Int, Comm)
81 mreduce f z (WNil _)       = (z, WNil ())
82 mreduce f z (Worker w ws) =
83     let (x, ws) = mreduce f z ws in
84     let (y, w) = w ▷ select Reduce ▷ receive in
85     (f y x, Worker w ws)
86
87 mallgather : Comm → ([Int], Comm)
88 mallgather ws = let (xs, ws) = mgather ws in
89                 (xs, mbroadcast xs ws)
90
91 mallreduce : (Int → Int → Int) → Int → Comm → (Int, Comm)
92 mallreduce f z ws = let (xs, ws) = mreduce f z ws in
93                     (xs, mbroadcast [xs] ws)
94
```

```
95  mdone : Comm → ()
96  mdone (WNil _)       = ()
97  mdone (Worker w ws) = w ▷ select Done ▷ wait;
98                          mdone ws
99
100 ----------------------------------------------------------------------
101
102 -- # Initialization
103
104 initialize : (Comm → a) → (WorkerStream → ())
105                                 → Int → a
106 initialize m w n = m $ winitialize w n
107 -- where
108 winitialize : (WorkerStream → ()) → Int → Comm
109 winitialize w n
110     | n == 0    = WNil ()
111     | otherwise = Worker (forkWith @ManagerStream @() w)
112                          (winitialize w (n - 1))
113
114 ----------------------------------------------------------------------
115
116 -- # Parallelized high level operations on lists using collective
       communication
117
118 -- | Parallelized map
119 nmap : Int → (Int → Int) → [Int] → [Int]
120 nmap n f xs = initialize @[Int] (mapManager xs) (mapWorker f) n
121
122 mapManager : [Int] → Comm → [Int]
123 mapManager xs ws = let (xs, ws) = ws ▷ mscatter xs ▷ mgather in
124                    mdone ws; xs
125
126 mapWorker : (Int → Int) → WorkerStream → ()
127 mapWorker f c = let (xs, c) = wscatter c in
128                 c ▷ wgather (map f xs) ▷ wdone
129
130
131 -- | Parallelized foldl
132 nfoldl : Int → (Int → Int → Int) → Int → [Int] → Int
133 nfoldl n f z xs = initialize @Int (foldlManager f z xs)
134                                   (foldlWorker f z) n
135
136 foldlManager : (Int → Int → Int) → Int → [Int] → Comm → Int
137 foldlManager f z xs ws = let (x, ws) = ws ▷ mscatter xs
138                                           ▷ mreduce f z in
139                          mdone ws; x
140
141 foldlWorker : (Int → Int → Int) → Int → WorkerStream → ()
142 foldlWorker f z c = let (xs, c) = wscatter c in
143                     c ▷ wreduce (foldl @Int f z xs) ▷ wdone
144
145
146 -- | Parallelized foldr
147 nfoldr : Int → (Int → Int → Int) → Int → [Int] → Int
148 nfoldr n f z xs = initialize @Int (foldrManager f z xs)
149                                   (foldrWorker f z) n
```

```
150
151 foldrManager : (Int → Int → Int) → Int → [Int] → Comm → Int
152 foldrManager f z xs ws = let (x, ws) = ws ▷ mscatter xs
153                                         ▷ mreduce f z in
154                           mdone ws; x
155
156 foldrWorker : (Int → Int → Int) → Int → WorkerStream → ()
157 foldrWorker f z c = let (xs, c) = wscatter c in
158                     c ▷ wreduce (foldr @Int f z xs) ▷ wdone
159
160
161 -- | Parallelized filter
162 nfilter : Int → (Int → Bool) → [Int] → [Int]
163 nfilter n f xs = initialize @[Int] (filterManager xs)
164                                    (filterWorker f) n
165
166 filterManager : [Int] → Comm → [Int]
167 filterManager xs ws = let (xs, ws) = ws ▷ mscatter xs ▷ mgather in
168                       mdone ws; xs
169
170 filterWorker : (Int → Bool) → WorkerStream → ()
171 filterWorker f c = let (xs, c) = wscatter c in
172                    c ▷ wgather (filter f xs) ▷ wdone
173
174
175 -- | Parallelized zipWith
176 nzipWith : Int → (Int → Int → Int) → [Int] → [Int] → [Int]
177 nzipWith n f xs ys = initialize @[Int] (zipWithManager xs ys)
178                                        (zipWithWorker f) n
179
180 zipWithManager : [Int] → [Int] → Comm → [Int]
181 zipWithManager xs ys ws =
182     let (zs, ws) = ws ▷ mscatter xs ▷ mscatter ys ▷ mgather in
183     mdone ws; zs
184
185 zipWithWorker : (Int → Int → Int) → WorkerStream → ()
186 zipWithWorker f c = let (xs, c) = wscatter c in
187                     let (ys, c) = wscatter c in
188                     c ▷ wgather (zipWith f xs ys) ▷ wdone
189
190
191 -- | Parallelized zipWith3
192 nzipWith3 : Int → (Int → Int → Int → Int) → [Int] → [Int]
193                                               → [Int] → [Int]
194 nzipWith3 n f xs ys zs = initialize @[Int] (zipWith3Manager xs ys zs)
195                                            (zipWith3Worker f) n
196
197 zipWith3Manager : [Int] → [Int] → [Int] → Comm → [Int]
198 zipWith3Manager xs ys zs ws =
199     let (rs, ws) = ws ▷ mscatter xs ▷ mscatter ys
200                       ▷ mscatter zs ▷ mgather in
201     mdone ws; rs
202
203 zipWith3Worker : (Int → Int → Int → Int) → WorkerStream → ()
204 zipWith3Worker f c = let (xs, c) = wscatter c in
205                      let (ys, c) = wscatter c in
```

```
206                     let (zs, c) = wscatter c in
207                     c ▷ wgather (zipWith3 f xs ys zs) ▷ wdone
208
209 ----------------------------------------------------------------------
210
211 -- | A function that return the length of a list of
212 -- | ManagerStream endpoints.
213 commLength : Comm → (Int, Comm)
214 commLength (WNil _)       = (0, WNil ())
215 commLength (Worker w ws) = let (i, ws) = commLength ws in
216                                (1 + i, Worker w ws)
```

## A.2 Futures module

```
1  module Futures where
2
3  -- When possible, this type will represent a Future
4  -- type Future = ?a;Wait
5
6  future : (() → a) → ?a;Wait
7  future f = forkWith @(?a;Wait) @()
8                  (\c:!a;Close 1→ c ▷ send (f ()) ▷ close)
9
10 block : ?a;Wait → a
11 block = receiveAndWait
12
13 delay : (() → a) → (() → b) → ?b;Wait
14 delay f1 f2 = future @b (\_:() → f1 (); f2 ())
```

## A.3 Streams module

```
1  module Streams where
2
3  -- # Naming convention
4  -- | O - output (endpoint) of a stream
5  -- | I - input (endpoint) of a stream
6
7  type OStream = ⊕{More: !Int;OStream, Done: Wait}
8  type IStream = dualof OStream
9
10 ----------------------------------------------------------------------
11
12 -- # Basic functions
13
14 sendS : Int → OStream → OStream
15 sendS x o = o ▷ select More ▷ send x
16
17 forward : IStream → OStream 1→ OStream
18 forward (Done i) o = close i; o
19 forward (More i) o = let (x, i) = receive i in
20                      o ▷ sendS x ▷ forward i
21
```

```
waitS : OStream → ()
waitS o = o ▷ select Done ▷ wait

-----------------------------------------------------------------

-- # Splitter and joiner functions

splitSDup : IStream → OStream 1→ OStream 1→ ()
splitSDup (Done i) o1 o2 = waitS o1; waitS o2; close i
splitSDup (More i) o1 o2 = let (x, i) = receive i in
                           splitSDup i (sendS x o1) (sendS x o2)

splitSAlt : IStream → OStream 1→ OStream 1→ ()
splitSAlt (Done i) o1 o2 = waitS o1; waitS o2; close i
splitSAlt (More i) o1 o2 = let (x, i) = receive i in
                           splitSAlt i o2 (sendS x o1)

splitSWith : (Int → Bool) → IStream → OStream 1→ OStream 1→ ()
splitSWith p (Done i) o1 o2 = waitS o1; waitS o2; close i
splitSWith p (More i) o1 o2 =
    let (x, i) = receive i in
    if p x then splitSWith p i (sendS x o1) o2
    else        splitSWith p i o1 (sendS x o2)

joiner : IStream → IStream 1→ OStream 1→ ()
joiner i1 (Done i2) o = waitS i2; forward i1 o
joiner (Done i1) i2 o = waitS i1; forward i2 o
joiner (More i1) (More i2) o = let (x, i1) = receive i1 in
                               let (y, i2) = receive i2 in
                               joiner i1 i2 (o ▷ sendS x ▷ sendS y)

 -----------------------------------------------------------------

-- # Converting to and from a list

fromList : [Int] → OStream → ()
fromList []      o = o ▷ waitS
fromList (x::xs) o = o ▷ sendS x ▷ fromList xs

toList : IStream → [Int]
toList (Done i) = close i; []
toList (More i) = let (x, i) = receive i in x :: toList i

-----------------------------------------------------------------

-- # Stream transformations

mapS : (Int → Int) → IStream → OStream 1→ ()
mapS f (Done i) o = close i; waitS o
mapS f (More i) o = let (x, i) = receive i in
                    mapS f i (sendS (f x) o)

foldlS : (Int → Int → Int) → Int → IStream → Int
foldlS f z (Done i) = close i; z
foldlS f z (More i) = let (x, i) = receive i in
                      foldlS f (f z x) i
```

```
78
79  foldrS : (Int → Int → Int) → Int → IStream → Int
80  foldrS f z (Done i) = close i; z
81  foldrS f z (More i) = let (x, i) = receive i in
82                        f x (foldrS f z i)
83
84  filterS : (Int → Bool) → IStream → OStream 1→ ()
85  filterS p (Done i) o = close i; waitS o
86  filterS p (More i) o = let (x, i) = receive i in
87                         filterS p i (if p x then sendS x o else o)
88
89  ----------------------------------------------------------------------
90
91  -- | Variation of `forkWith` which creates two channels instead of one.
92  forkWith2 : ∀ a:1A b:1A c . (dualof a 1→ dualof b 1→ c) → (a, b)
93  forkWith2 f = let (x1, y1) = new @a () in
94                let (x2, y2) = new @b () in
95                fork (\_:() 1→ f y1 y2);
96                (x1, x2)
```

# Appendix B

# FEP-0009: Add new concurrency-related primitives to FreeST

## FEP-0009: Add new concurrency-related primitives to FreeST

### FEP-0009: Add new concurrency-related primitives to FreeST

#### Overview

| Header | Value |
|---|---|
| **FEP** | 0009 |
| **Title** | Add new concurrency-related primitives to FreeST |
| **Author** | @glopes |
| **Status** | Draft |
| **Type** | Enhancement |
| **Created** | 27-april-2022 |
| **FreeST-Version** | v3.0.0 |
| **Depends on** | - |

#### Abstract

As a concurrent programming language, having more control over threads and the overall concurrency system in FreeST would be helpful.

Haskell offers several abstractions so users can manipulate or be knowledgeable of the scheduler information in runtime. Most of their behaviour is currently inaccessible in FreeST, such as getting the ID of a thread or "blocking" a thread for a particular duration.

#### Motivation

Some languages are similar to FreeST in their approach and view towards concurrency. We frequently compare FreeST to the Go programming language to better explain and expose FreeST's advantages in what it is achieving. Both work with lightweight threads and channels as the (primary) source of communication between processes.

Other languages come to mind when considering constructing concurrent systems, such as Rust or even Java, as a more mainstream example. In the case of these languages, we (mostly) work with system threads, which are way more expensive and reflect a very different scheduler that manages their lifetime.

All these languages offer essential primitives that allow programmers to access information on their system (e.g. the number of cores available), ways of interacting with the runtime system or controlling concurrent units (threads). The primitives available vary accordingly with each language's needs and particular characteristics.

This behaviour is currently inaccessible in FreeST and is essential for implementing several concurrent patterns, complex structures and ambitious systems. Since we depend on Haskell's scheduler to manipulate and manage threads and are not allowed to interact with its runtime system, we can not implement the needed primitives for this behaviour.

## Specification

The following primitives I am proposing, borrowed from the Haskell `Control.Concurrent` library, are primitives I noticed to be meaningful and offer behaviour that FreeST does not currently allow:

- `getNumCores` - Return the number of system threads that can run truly simultaneously (on separate physical processors) at any given time.

  In concurrent programs, this primitive might be useful, for example, to control the number of processes that are "forked" programmatically or by offering the possibility of intuitively estimating it.

  It is a small addition, but nearly every popular programming language focusing on concurrency allows this (Go and Rust, for example).

- `setNumCores` - Set the number of system threads that can run truly simultaneously (on separate physical processor at any given time).

  Currently, FreeST programs are predefined to run on the maximum number of system threads possible in the project's `package.yaml` file in the `ghc-options: -main-is FreeST -threaded -rtsopts -with-rtsopts=-N` executable configuration line. Although, "It is strongly recommended that the number of capabilities is not set larger than the number of physical processor cores, and it may often be beneficial to leave one or more cores free to avoid contention with other processes in the machine.".

  Fortunately, it is possible to set the number of system threads to a different number by executing the program with the command `freest program.fst +RTS -Nx`, being *x* the number of system threads available to the program. However, a primitive that offers the possibility of altering this programmatically at any given moment might be an interesting addition.

  Once again, it is a small addition. However, this behaviour appears in programming languages that work with lightweight threads (Go, for example).

- `myThreadId` - Returns the Id of the calling thread.

- `threadCore` - Returns the number of the core on which the thread is currently running.

  This primitive would allow us to create structures that simulate the state of each thread system queue in run-time related to the distribution of lightweight threads in several system threads.

  For a personal application, I could use this to the ForkJoin granularity control technique called *Surplus*: Before creating a new task, the number of queued tasks (lightweight threads) in the current (system) thread that exceeds the number of tasks in other queues is compared to a parameterized threshold limit. If the surplus tasks count is higher than the threshold, the task will be executed sequentially. If the surplus tasks count is lower than the threshold, the task is created in parallel.

- `threadDelay` - Suspends the current thread for a given number of microseconds.

  This one is debatable. Most distributed systems frequently use this primitive (or an equivalent one) to synchronise part of asynchronous systems in specific scenarios. There are a lot of legitimate real-world applications, such as avoiding busy waiting by implementing delays between attempts at some execution. Issue #161 (closed) proposes another application using this primitive. However, this mechanism is a critical source of deadlocks, something we try distancing ourselves from.

  Moreover, nearly every popular programming language focusing on distributed systems allows this (C, Go, and Rust, for example).

These primitives can be consulted at Control.Concurrent for more details.

### Syntax and Implementation

Most of the implementation should be straightforward. Like other Haskell primitives built into FreeST's Prelude, we have to add and adjust the mentioned primitives in the `Interpreter.Builtin`.

In the Prelude, we would maybe have the following signatures:

```
type ThreadId = ??

getNumCores : Int
setNumCores : Int -> ()

myThreadId : ThreadId
threadCore : ThreadId -> Int

threadDelay : Int -> ()
```

I left ThreadId as a to-be defined type because the original `myThreadId` and `threadCore` (called `threadCapability` in Haskell) return and receive a particular data type `ThreadId`:

```
data ThreadId = ThreadId ThreadID#
```

"A `ThreadId` is an abstract type representing a handle to a thread." and "(...) if you have a `ThreadId`, you essentially have a pointer to the thread itself."

We would probably need to give this data type some special treatment since it appears impossible to transform an Int or String into a Haskell `ThreadId`.

A more basic version of the primitives I am proposing could be:

```
type ThreadId = String

myThreadId : ThreadId
threadCore : Int
```

`myThreadId` returns a conversion of the original Haskell `threadId` to a String ("The `Show` instance lets you convert an arbitrary-valued `ThreadId` to string form..."). As for `threadCore`, we could simplify its logic and behaviour by internally executing a `myThreadId`, returning the `ThreadId` of the current thread, and applying it to Haskell's `threadCapability` first parameter, returning the number of the capability on which the thread is currently running.

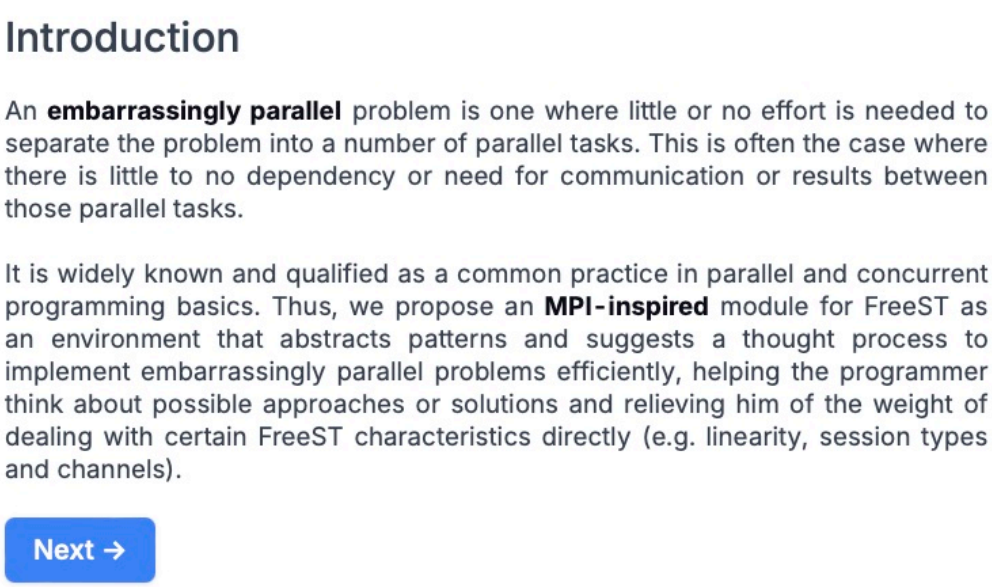
In short, the implementation of `getNumCores`, `setNumCores` and `threadDelay` is immediate. On the other hand, `myThreadId` and `threadCore` following my suggestion are also immediate; otherwise, it might require more time and effort for a more in-depth analysis for its implementation.

If FreeST happened to be written in another programming language, we would have to adjust the primitive association to a distinct language's concurrent API that interacts with a very particular scheduler. Then, FreeST would be significantly different, requiring a different approach to analyse its needs, which could impact this proposal.

# Appendix C

# Surveys

## C.1 Parallel module



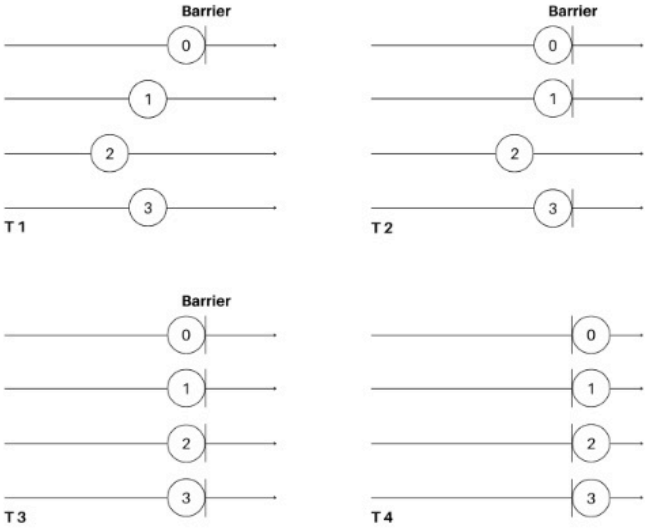
# The Message-passing Interface (MPI)

The **message-passing interface**, or **MPI**, is a message-passing library interface specification that primarily addresses the message-passing parallel programming model, in which data is moved from a process' address space to that of another through cooperative operations.

MPI defines a few relevant types of communication, particularly the collective, which involves a group or groups of processes.
Below, I've included an illustration of the core patterns this communication type provides.

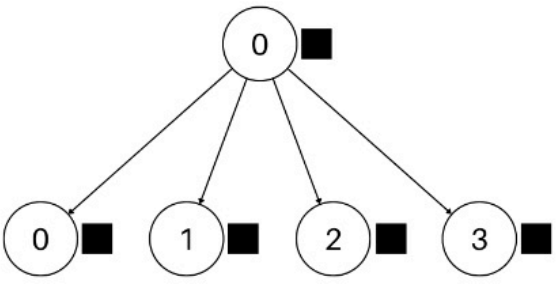
## Barrier

Forms a barrier, and no processes in the communicator can pass the barrier until all of them call the operations, i.e., synchronisation across all members of a group.



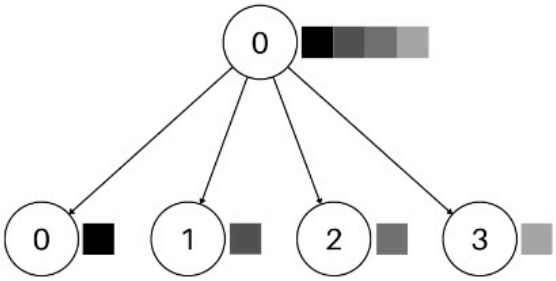

## Broadcast

Sends the same data from one member to all members.



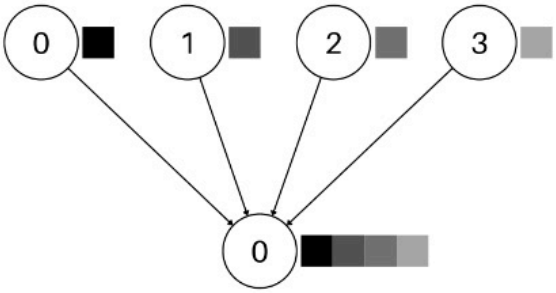
### Scatter

Sends data from one member to all members. In this case, it sends *chunks of an array* to different members.



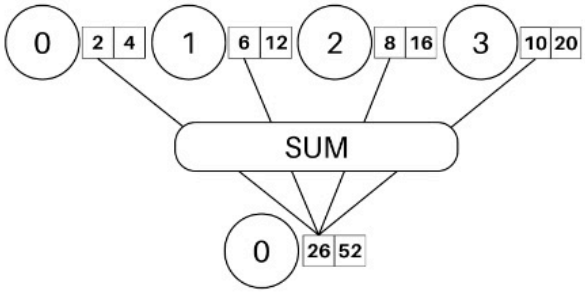

### Gather

Receives elements from all members and gathers them into a single member.



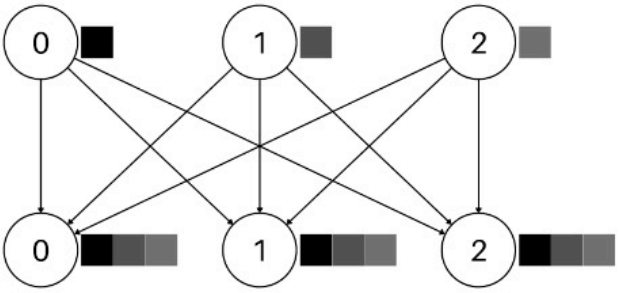

### Reduce

Takes an array of elements on each member and sends its reduced result to a single member. Examples of reduced results could be folding a list by applying a specific function, such as for summing numbers.



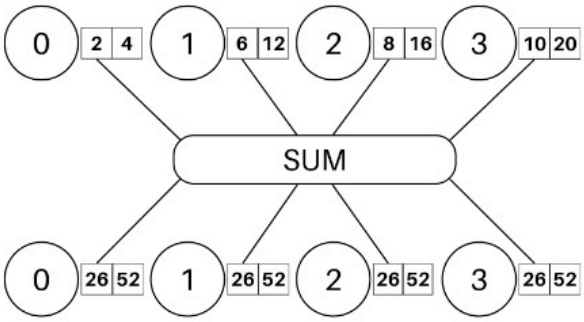
### Allgather

Variation of *gather* in which all members receive the result.




### Allreduce

Variation of *reduce* in which all members receive the result.




Next →

## Parallel module description

We found that an **MPI-inspired** module, identifying patterns and defining suitable abstractions for our purposes, would allow the programmer to focus on the essential logic of the problem by hiding all the complexity related to the standard and widely-known approaches for collective communication and nuances of the FreeST language.

In this module, called ***Parallel***, the MPI **collective communication** is simplified and adjusted to the FreeST particularities. So, it's necessary to maintain a communication channel between the root and each process.
In FreeST, each party in a communication is responsible for holding a channel endpoint. Therefore, the root has to keep a list of all the endpoints for all the communications established between it and each process.
Each operation demands communication between the root and all the processes, meaning each operation iterates (and reconstructs for their continuation) the list of endpoints from the root's side.

Accordingly, the module follows a protocol that defines the possibilities for specifying the collective communication:

```
type ParallelStream:1S = +{ Broadcast: ![Int]
                          , Scatter  : ![Int]
                          , Gather   : ?[Int]
                          , Reduce   : ?Int
                          , Done     : Wait} ; ParallelStream
```

Some operations do not appear in the protocol because they are a sequence of existing operations. These include the ***Barrier*** operation, a *Gather* followed by a *Broadcast*; the ***Allgather***, a *Gather* followed by a *Broadcast*; the ***Allreduce***, a *Reduce* followed by a *Broadcast*.

In our approach, we break from the root-processes terminology. Unlike in MPI's original specification, what could be called root does not participate in the work as a process but appears much more like a manager, i.e., as the single central originating or receiving point in the communication. Instead of the root-process terminology, we will use **manager-worker** and refer to their relationship.

Furthermore, we introduce the *initialize* abstraction that initializes the group of workers in the communication. Its signature is as follows:

```
initialize : forall a:*T . (WorkerList -> a) -> (dualof ParallelStream ->
()) -> Int -> a
```

It receives the function encapsulating the manager's logic, responsible for orderly calling/selecting the desired operations, the function encapsulating the workers' logic, and the number of workers to initialise, answering accordingly to the called operations and executing them concurrently.

Following up on the protocol above, we offer two abstractions for each operation, one for the manager side and one for the workers' side, i.e., the two parties responsible for two different channel endpoints. The signature for each abstraction is as follows:

```
mbarrier : WorkerList -> WorkerList
wbarrier : dualof ParallelStream -> dualof ParallelStream

mbroadcast : [Int] -> WorkerList -> WorkerList
wbroadcast : dualof ParallelStream -> ([Int], dualof ParallelStream)

mscatter : [Int] -> WorkerList -> WorkerList
wscatter : dualof ParallelStream -> ([Int], dualof ParallelStream)

mgather : WorkerList -> ([Int], WorkerList)
wgather : [Int] -> dualof ParallelStream -> dualof ParallelStream

mreduce : (Int -> Int -> Int) -> Int -> WorkerList -> (Int, WorkerList)
wreduce : Int -> dualof ParallelStream -> dualof ParallelStream

mallgather : WorkerList -> ([Int], WorkerList)
wallgather : [Int] -> dualof ParallelStream -> ([Int], dualof
ParallelStream)

mallreduce : (Int -> Int -> Int) -> Int -> WorkerList -> (Int, WorkerList)
wallreduce : Int -> dualof ParallelStream -> ([Int], dualof
ParallelStream)

mdone : WorkerList -> ()
wdone : dualof ParallelStream -> ()
```

Unfortunately, we decided that the *Broadcast* operation sends lists only because FreeST is not yet equipped with polymorphic data types (in this case, session types). We decided on this because of the *Allgather* operation since there is the need to broadcast the resulting list. To broadcast integers, we need to "wrap" them in a list of one element. Consider this temporary.

Our *reduce* operation differs from MPI's definition. In our case, each worker sends an Integer to the manager, which applies a function to the received values, reducing them into a final result.

Moreover, the scatter operation, in this case, the *mscatter* abstraction, distributes a list in **even** chunks.

Next →

## Example: Scalar product

The **scalar product** is an algebraic operation that takes two equal-length sequences of numbers and returns a single number.

In mathematics, the dot product or scalar product is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors) and returns a single number. Algebraically, the scalar product is the sum of the products of the corresponding entries of the two sequences of numbers.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

### Implementation with the Parallel module

```
import Parallel

worker : dualof ParallelStream -> ()
worker c = let (xs, c) = wscatter c in
           let (ys, c) = wscatter c in
           c |> wreduce (scalarProduct xs ys) |> wdone

scalarProduct : [Int] -> [Int] -> Int
scalarProduct [] []           = 0
scalarProduct (x::xs) (y::ys) = x * y + (scalarProduct xs ys)

manager : [Int] -> [Int] -> WorkerList -> Int
manager xs ys ws = let (zs, ws) = ws |> mscatter xs |> mscatter ys
                                     |> mreduce (+) 0 in
                   mdone ws; zs

main : Int
main = let xs = [1, 2, 4, 8, 16, 32, 64, 128] in
       let ys = [1, 2, 3, 4, 5, 6, 7, 8] in
       initialize @Int (manager xs ys) worker 3
```

In this implementation, the division into parts stands out.
The *main* function will be the starting point to tell the story of this program: it initialises two lists to be used in the scalar product computation and calls the *initialize* abstraction. Noticeably, it must receive two functions, one encapsulating the manager's logic and the other the workers' logic, establishing the communication for the *manager-workers* relation.

The *manager* function receives the two lists and distributes its contents between all workers and receives the results, reducing them into the final result. In other words, it translates in two *mscatter*, one for each list, and a *mreduce* call.

Through the *worker* function, the workers receive their chunk of each list, calculate the scalar product, and send the result back to the manager.
Thus, it translates into the calls that complete the communication accordingly, in this case, two *wscatter* and a *wreduce* call.

To end the communication, both sides (manager and workers) have to call *mdone* and *wdone*, respectively.

It is a fitting and simple way of implementing an *embarrassingly parallel* problem in which every worker processes and computes the distributed data concurrently.

Next →

## Exercise

Finally, using the Parallel.fst module, try implementing the following program: distribute the list **[1, 2, 4, 8, 16, 32, 64, 128, 256]**among **3** workers executing concurrently so that each one sums their respective received chunk, and then the manager sums each result into a final result.

**Upload your implementation**

Drag & drop a file or browse

**If you could not complete this exercise, describe what prevented you from completing it, your experience and difficulties.**

Next →

## Evaluation and feedback

Finally, for the evaluation, we would like to ask for feedback on the module's usefulness and accessibility.

Having the **scalar product** implementation and the given exercise as references, evaluate the following parameters on a scale from 1 to 10.

**How easy was it to implement embarrassingly parallel problems using the provided module and to understand its workflow?**
**In other words, how accessible is the module?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Hard Easy

**Is it relevant to integrate this module into the FreeST programming language?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Irrelevant Relevant

**How would you rate your overall experience concerning the module's development environment?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Bad Good

**At last, please feel free to point out any other observations or ask any questions.**

Submit

## C.2 Futures module

### Introduction

In the broadest sense, a **future** execution can be seen as a value that will eventually become available.

By definition, a **future** represents a value that may not be immediately available. It is a placeholder for a result that will be computed asynchronously. When a task has started, a future is returned immediately, and the program can continue its execution. The future will eventually hold the result of the task when the task is completed.

Futures might provide extra methods or features that offer better flexibility and control. Still, their essential advantage is allowing programmers to write code that can execute other tasks concurrently while waiting for the completion of a future.

Furthermore, futures are often used in programming languages to provide a way to write asynchronous, non-blocking code that can handle multiple tasks concurrently without blocking the program's execution. They help manage the complexity of asynchronous programming by providing a straightforward interface for handling asynchronous results and allowing developers to write more readable and maintainable code.

Some of its common applications are parallelism and concurrency, asynchronous I/O operations, web development, asynchronous event handling, data streaming, deferred execution and resource management.

Currently, FreeST offers the possibility of launching new threads to execute computations in parallel through the *fork* primitive and the *forkWith* abstraction. However, the computation's return is discarded when its execution finishes, so its behaviour, by default, is independent of other threads' computations.
The programmer has to manually work around this by exchanging data through channels. This means more code, implementation time, error chance, and other disadvantages that could be easily avoided with the correct tools.

With this in mind, futures sounds like a perfect fit for FreeST's built-in concurrency features.

Next →

## Implementation

The implementation was intuitive according to FreeST's use of channels for its message-passing communication approach.

The following abstraction is the core for everything in this module:

```
future : forall a:*T . (() -> a) -> ?a;Wait
```

As the name suggests, this abstraction **creates** and **returns a new future**. It receives a function to execute asynchronously in the newly created future and returns its representation, an input channel endpoint.

To implement this, we benefited from FreeST's channels and currently available primitives and abstractions.
It launches a new thread that executes a task in parallel and creates a new channel. Furthermore, it associates an output endpoint with the given function (task) and sends its result to the channel through the given endpoint.

The abstraction returns a future, which is an input channel endpoint, allowing the possibility of **waiting to receive its result** through the following abstraction:

```
block : forall a:1T . ?a;Wait -> a
```

We also offer some extra abstractions, such as *invoke* and *delay*, which apply to more specific use cases but are of no significant interest to this survey.

In the near future, we expect to be able to implement the type *Future* to abstract the session type *?a;Wait*.

Next →

## Examples

Some basic examples will demonstrate how this module and its abstractions can be used so the programmer can start developing parallel and concurrent programs.

Let's see how to **create** futures and **wait** for the result of their execution.

```
f : Int -> Int
f x = x

main : Int
main = future @Int (\_:() -> f 1) |> block @Int
```

In the *main* function, a future for a given *f* function is created through the *future* abstraction. Remember, this abstraction creates a new channel, holds an output endpoint and returns a future (the *dual* input endpoint). This new future executes *f* in parallel and, when finished, sends the result to the just-created channel.
With the execution running in parallel, the program blocks until it is complete. This is achieved through the *block* abstraction, in which the future is given (returned by the *future* abstraction) as an argument and proceeds to receive the result standing in the channel.

However, to benefit from what futures essentially desire to offer in terms of parallelism and concurrency, they are usually accompanied by other computations in the program.

```
g : Int -> Int
g x = ...

main : Int
main = let f1 = future @Int (\_:() -> g 3) in
       f 3; block @Int f1
```

To represent an arbitrary heavy computation, assume the *g* function to block the current thread for a given number of seconds and return that same number.
This is a simple example where, similar to common practices in FreeST with the *fork* and *forkWith* primitives, the programmer can create futures to execute computations in parallel while having the possibility of retrieving its result.
A future is created, a thread executes a computation that takes 3 seconds to finish and, simultaneously, *f* executes in the main thread, proceeding to wait for the future's computation to finish.

## Prime numbers

A **prime number** is a number greater than 1 that is not a product of two smaller numbers. For example, 5 is prime because the only way to write it as a product, 1 × 5 or 5 × 1, involves 5 itself.

Futures allow a possible approach for the parallel implementation of this problem through the **divide-and-conquer** paradigm in which an algorithm recursively breaks down a problem into two or more sub-problems of the same or related type until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Having the problem of counting the prime numbers within a range, see the following implementation:

```
primes : Int -> Int -> Int
primes low high
  | high - low < 10 = countPrimes low high
  | otherwise       = let mid = low + (high - low) / 2 in
                      let f1 = future @Int (\_:() -> primes low mid) in
                      let f2 = future @Int (\_:() -> primes (mid + 1)
high) in
                      (block @Int f1) + (block @Int f2)
```

In the first function call, the problem is divided into two sub-problems by dividing the given range into two halves and creating two futures which "recursively" execute the same function concurrently. The first branch runs '*primes low mid*' while the second runs '*primes (mid + 1) high*', representing the computations counting the prime numbers within the halves.
In each branch, the same behaviour repeats until some branch holds a range of length smaller than 10, proceeding to count the prime numbers within the given range sequentially.
Finally, the result for each future, starting from the "lowest branches", becomes available, and each one is gradually computed by summing the available result of the futures tasked with counting the prime numbers in a certain range.

Imagine it as a "tree".
The starting computation splits until some branches reach a range smaller than 10 in length; we could call them "leaves". Then, the results are merged (by addition, in this case) until they become one again.

Next →

## Exercise: Fibonacci sequence

In mathematics, the **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones. The sequence starts from 0 and 1, and the first few values are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.
The Fibonacci numbers may be defined by the recurrence relation (for n > 1):

$$F_0 = 0, \quad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}$$

In mathematics, a **recurrence relation** is an equation in which the nth term of a sequence of numbers is equal to some combination of the previous terms.

Finally, using the Futures.fst module, try implementing the concurrent divide-and-conquer version of the Fibonacci sequence.

Upload your implementation

Drag & drop a file or browse

If you could not complete this exercise, describe what prevented you from completing it, your experience and difficulties.

Next →

## Evaluation and feedback

Finally, for the evaluation, we would like to ask for feedback on the module.

Having the module's description, examples and the given exercise as references, evaluate the following parameters on a scale from 1 to 10.

**Is it adequate and coherent to integrate this module into the FreeST programming language? Is it compatible with its characteristics (e.g. session types, channels, threads, etc...)? ***

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Inadequate Adequate

**Is it relevant to integrate this module into the FreeST programming language? ***

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Irrelevant Relevant

**At last, please feel free to point out any other observations or ask any questions.**

Submit

## C.3 Streams module

# Introduction

In programming languages, **streams** are a sequence or flow of data elements that can be continuously and sequentially read from or written to. Streams are widely known for handling input and output operations, especially when dealing with large volumes of data or when data is produced or consumed over time.

Streams abstract the underlying details of **reading** and **writing** data, making it easier for programmers to work with different data sources and destinations without worrying about the specific implementation details. Streams provide a uniform interface to interact with various data sources, such as files, network sockets, standard I/O, and more.

Streams are commonly used in parallel and concurrent programming scenarios. They play a significant role in handling data processing tasks efficiently and effectively when dealing with multiple cores, threads, or processing units.

In FreeST, channels are the mechanism for the exchange of data between agents. To view it as a stream, it usually follows a recursive protocol that sends an element of a certain type until one of the agents decides to end the "transmission". An example of this could be the following:

```
type IntStream : 1S = +{More: !Int;IntStream, Done: Close}
```

This represents a stream of integers.

Since FreeST is a concurrent programming language, it can benefit from its concurrent nature to asynchronously exchange, handle and process data by launching threads tasked with particular behaviour in parallel.
In FreeST, it is a common practice to create and use stream-like sessions, so there is no reason not to encapsulate and abstract its behaviour in a module that provides a specialised tool to handle streams more efficiently.
Furthermore, concurrently handling data through streams allows new approaches and possibilities for parallel and concurrent problem implementations.

This led to the conclusion that a FreeST module dedicated to efficiently handling streams might open new possibilities for parallel and concurrent programming and real-world applications.

Next →

## Implementation

The following session types are the starting point for this module.

```
type OStream : 1S = +{More: !Int;OStream, Done: Close}
type IStream : 1S = dualof OStream
```

These protocols represent a stream of data and are dual to each other. This means that to exchange data between agents, one will hold an *OStream* output endpoint to "feed" (**write**) integers into a stream, whereas the other agent maintains the *IStream* input endpoint to **read** the integers.

In order to handle communications using these protocols, we offer the following abstractions:

```
sendS : Int -> OStream -> OStream
```

*sendS* is an abstraction that is given a value and sends it through a given *OStream* channel endpoint, returning its result, which is the channel continuation according to the protocol.

```
forward : IStream -> OStream 1-> OStream
```

*forward* is an abstraction that is given the input endpoint of a stream and the output endpoint of another and forwards the content of the first to the latter, returning its channel continuation.

```
closeS : OStream -> ()
```

*closeS* ends a channel communication, i.e., closes a stream and the possibility of using it.

```
fromList : [Int] -> OStream -> ()
```

*fromList* feeds the content of a list into a given stream.

```
toList : IStream -> [Int]
```

Inversely, *toList* uses the content of a stream to (re)construct a list.

These abstractions represent more straightforward communication patterns and are helpful for basic "serialisation" and "deserialisation" between agents.

However, to make use of FreeST's potential, we looked for ways to expand the number of features and abstractions this module offers, seeking to establish new flexible ways of using and thinking about streams and how to apply them to parallel and concurrent programming in general.

To satisfy this, **splitters** were introduced.

Splitters are an approach to distributing the data between two streams. It follows the logic of linking a stream output endpoint to two input endpoints without having the programmer decide how to distribute the data manually.

We offer three different types of splitter abstractions with similar signatures and behaviour.
Given an input endpoint of a stream and two output endpoints of streams the data is being distributed to, each abstraction feeds the data of the first one into the other two:

```
splitSDup : IStream -> OStream 1-> OStream 1-> ()
```

In this case, each element is **duplicated** and sent to each stream, resulting in each containing the same data.

```
splitSAlt : IStream -> OStream 1-> OStream 1-> ()
```

Here, the data distribution is divided by **alternating** which stream receives the next element.

```
splitSWith : (Int -> Bool) -> IStream -> OStream 1-> OStream 1-> ()
```

In this last case, the data distribution is divided depending on a **given predicate**, i.e., by applying each element to the predicate, and if it happens to be true, the element is sent to the first stream; otherwise, it is sent to the second.

Finally, the *mapS*, *foldlS*, *foldrS* and *filterS* abstractions for high-level operations for data manipulation using streams are also available.
*The mapS* and *filterS* manipulate an input stream's data and feed the resulting data into the output of another. The *foldlS* and *foldrS* do the same but return the result directly.

Even though this implementation operates only on integers, we intend, when it becomes possible, to adapt it to every type.

The exciting point of handling data through streams using this module is its dynamic approach of constantly processing and exchanging data in parallel.
For example, a stream can simultaneously receive and send data in parallel, making them independent computations. This is very efficient.

Next →

# Examples

The use of these abstractions is usually accompanied by parallelism, i.e., by forking them. This allows programs to benefit the most from FreeST's concurrent capabilities and the nature of streams, as I described previously.

Assume the existence of *forkWith2*, a version of *forkWith* which launches a new thread and **two** new channels.

## The Quicksort algorithm example

**Quicksort** is an efficient, widely used sorting algorithm that follows the **divide-and-conquer** approach to sort a list of elements in ascending or descending order.

The QuickSort algorithm works as follows:
1. **Choose a Pivot**: Select an element from the list as the "pivot". The choice of the pivot can significantly impact the algorithm's performance, but in the simplest implementation, the first or last element is often chosen as the pivot.
2. **Partitioning**: Rearrange the list such that all elements less than the pivot come before it, and all elements greater than or equal to the pivot come after it. The pivot is now in its final sorted position.
3. **Recursion**: Recursively apply QuickSort (steps 1 and 2) to the two sublists on either side of the pivot (the elements smaller than the pivot and the ones greater than the pivot). Continue this process until the base case is reached (e.g., a sublist contains zero or one element already sorted).
4. **Combine the Results**: As the recursive calls return, concatenate the sorted sublists around the pivot to form the fully sorted list.

To demonstrate the power and usefulness of our contribution, an implementation approach to the quicksort algorithm using streams follows.
The crucial part of this algorithm is the partition conditioned by the pivot. Intuitively, we can use a splitter to apply this part of the algorithm and distribute the data between two units.
Furthermore, let's see how we can adapt to its divide-and-conquer and recursive nature:

```
sqsort : IStream -> OStream 1-> ()
sqsort (Done i) o = wait i; closeS o
sqsort (More i) o =
  let (x, i) = receive i in
  let (i1, i2) = forkWith2 @IStream @IStream @() (splitSCond (<x) i) in
  let i3 = forkWith @IStream @() (sqsort i1) in
  let i4 = forkWith @IStream @() (sqsort i2) in
  o |> forward i3 |> sendS x |> forward i4 |> closeS

qsort : [Int] -> [Int]
qsort xs = let i1 = forkWith @IStream @() (fromList xs) in
           let i2 = forkWith @IStream @() (sqsort i1) in
           toList i2

main : [Int]
main = qsort [2, 4, 16, 8, 32, 64, 128, 1]
```

*qsort* is the function representing the building blocks. First, it calls *fromList* to feed its numbers into a stream. Then, *sqsort*, the function that orders those numbers, is called. Finally, *toList* converts the numbers in-stream to a list.

*sqsort* is the essential part of this implementation. It is given an input stream endpoint to receive data from an existing stream and an output stream endpoint to feed data into another stream.

1. The first iteration, holding the input endpoint for the stream holding the list of numbers, receives the head number and declares it as the **pivot**.
2. Accordingly, it proceeds to **partition** the data by using the conditional *splitSCond* splitter, which handles two new streams (created through the *forkWith2* abstraction), and if the remainder received numbers are less then the pivot, they are fed into a stream; otherwise, they are provided into the other.
3. Finally, in a divide-and-conquer manner, it divides the problem into two smaller problems by recursively calling *sqsort* with the input endpoint of the splitters; in other words, each deals with half the data in parallel. This process **recursively repeats** until the streams processing the data contain only one element. Until that point, each previous recursive computation is waiting for the program to reach that point where it can't divide the problem any longer, and the remaining task is to concatenate the data in the streams.
4. Each call to *sqsort* is also given an output stream endpoint to communicate with other agents, the previous computation nodes in the "recursion tree" in this case. In the last line of the function, each computation receives the data of the two computations it recursively calls. To **merge** the data, each call uses the output endpoint it holds to send the received data of the first subproblem, then sends the initially selected pivot and finally sends the received data of the second subproblem.

This creates a tree-like stream distribution, in which the "leaf" streams more to the left contain the lowest elements while the ones to the right have the higher ones, so it is possible to merge the ordered data without problems.

## Exercise

Finally, using the **Streams.fst** module, try implementing the following program: feed the data of a list into a stream, split the data between even and odd by feeding them into other two streams, and then return a pair with the count of even and odd numbers.

Upload your implementation

Drag & drop a file or browse

If you could not complete this exercise, describe what prevented you from completing it, your experience and difficulties.

Next →

## Evaluation and feedback

Finally, for the evaluation, we would like to ask for feedback on the module's usefulness and accessibility according to the goals and language design of the FreeST programming language.

Having the given explanations and examples as references, evaluate the following parameters on a scale from 1 to 10.

**Is it adequate and coherent to integrate this module into the FreeST programming language? Is it compatible with its characteristics (e.g. session types, channels, threads, etc...)?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Inadequate Adequate

**How easy is using the provided module and understanding its workflow? In other words, how accessible is the module?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Hard Easy

**Concerning its use cases for stream-related programming, is integrating this module into the FreeST programming language relevant?** *

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

Irrelevant Relevant

**At last, please feel free to point out any other observations or ask any questions.**

Submit

# Appendix D

# Survey insights

## D.1 Parallel module

1. This module made it easy to abstract away all the boilerplate required to build a parallel application and leave only the logic relevant to the domain. In other words, I thing your goals have been met.

   I have two complaints, both about names. First: I think, 'dualof ParallelStream' is used too many times not to be given a synonym. It would make reading the type signatures much easier. Second: the names of the operations on the worker's side can be misleading. For example, at first glance, 'wscatter' seems to imply that the worker is 'scattering' a list, when it is being 'scattered with' it. While this was explained in the introduction, and it may be easy to keep in mind when *writing* code, I think it can still be misleading when *reading* code. Perhaps you could find more suggestive names for these operations – perhaps including propositions, or different verb conjugations. I have no concrete suggestions for now though, sorry :^)

2. Again, creating the number of workers and reducing should be high-level functions, without the need to close. That should be wrapped in a higher-level API without Session Types.

3. Very well documented

4. It is not straightforward to work with this module. It seems very promising and in the spirit of channels and parallel/concurrent programming, but it will need very good documentation to flatten the learning curve as much as possible. In my opinion, starting by giving purpose to each primitive is better than presenting a primitive and expect the programmer to have 'hopes of using it'. Documentation and learning aside, it seems very easy once you acquire the right mindset. My only 'beef' with it is that session types are not essential to this implementation, it's a "FreeST meets parallel programming" rather than a "FreeST has essential tools and primitives for parallel programming" (at least in the case of MPI-style programming). This is not inherently bad, but it might undermine the motivation of "Why FreeST and not another language"

## D.2 Futures module

1. This simple abstraction faithfully models a classic concurrent programming primitive using FreeST's session-typed channels. By providing this familiar construct, this module may help new users adapt to the language, and by exposing simple session types in the type signature of future and block, this module may also help new users understand how session types work. For this reason I think it would be valuable to have a guide in the official website that introduces this abstraction in familiar terms and gradually exposes the session-typed 'machinery' behind it.

   It is unfortunate that the use of this module is currently impaired by the need to include type applications and type signatures in lambda expressions, but I believe it will be much more usable in the presence of type inference. The need to create a thunk to prevent evaluation before the desired expression is passed to the future function seems to be unavoidable, but since this is also a common idiom when forking new threads, it might be worth it to think whether some syntactic sugar such as '(thunk e)', or even '(\thunk e)' would be a valuable addition to the language.

2. Although not related to this module in specific, it may be weird for the new users to use the future function like "future @Int (_:() -> fibonacci (n - 1))" instead of "future @Int (fibonacci (n - 1))"

3. Abstractions like these are extremely useful. They provide a simple and compact way to implement solutions using the divide-and-conquer paradigm, which I find to be a natural approach for handling concurrent programs.

   Currently, implementing this in FreeST requires writing cumbersome boilerplate code to fork new threads and manage data exchange through communication channels.

4. Without type inference, the syntax for the lambda that takes the Unit type becomes quite distracting. The name Unit (à lá Scala) makes it more readable. And avoids the () literal vs () type confusion (as in Haskell).

5. No documentation ( or seems missing). Some descriptions on the first pages were a bit confusing, rephrasing some lines should be good.

## D.3 Streams module

1. The wide-scale use of this module seems to be limited by the language itself, which does not support type operators or lists with elements other than integers. It's use is also made verbose by the need to include type applications, but seems very clean without it. The forkWith2 function is quite useful, and I suspect other such helper functions will appear in due time. It will be interesting to see how it will be used (and expanded) when these limitations disappear.

2. It took me a while to understand the behaviour of each function and the example provided since it uses a lot of forks

3. I wanted a non-session types API that would use the Session Type Checking to point errors in my code. But I do not want to create processes manually.

4. I think this module is easier to understand and follow than the parallel module

5. Module seems to unify and simplify boiler-plate code for these types of problems so it's definitely a plus. I think it is very easy to use, however complete newbies might still struggle with creating and managing channel endpoints (every endpoint is user managed). In terms of use cases, for now they seem very slim, but maybe some problems in the realm of data analysis or data processing will yield interesting cases (perhaps even AI?). My intuition points to neural networks, or image processing (use a bitmap image for simplicity) as very interesting opportunities.