Simple, Safe and Efficient Abstractions for Communication and Streaming in Parallel Computing

Amaury Maillé

▶ To cite this version:

Amaury Maillé. Simple, Safe and Efficient Abstractions for Communication and Streaming in Parallel Computing. Distributed, Parallel, and Cluster Computing [cs.DC]. Ecole normale supérieure de lyon - ENS LYON, 2023. English. NNT : 2023ENSL0041 . tel-04238803

HAL Id: tel-04238803
https://theses.hal.science/tel-04238803v1

Submitted on 12 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse
en vue de l'obtention du grade de Docteur, délivré par
l'École Normale Supérieure de Lyon

École Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 07/07/2023, par:

Amaury MAILLÉ

Simple, Safe and Efficient Abstractions for Communication and Streaming in Parallel Computing

Des Abstractions Simples, Sûres et Efficaces pour la Communication et le Streaming dans le Calcul Parallèle

---

Devant le jury composé de:

Baude Françoise, Professeure des Universités, Université Côte d'Azur	Rapporteuse
Thibault Samuel, Professeur des Universités, Université de Bordeaux	Rapporteur
Moy Matthieu, Maître de Conférences, Ucbl	Examinateur
Pouzet Marc, Professeur des Universités, Ens - Psl	Examinateur
Henrio Ludovic, Chargé de Recherche, Cnrs	Directeur de thèse

Résumé

Depuis la fin du scaling de Dennard, il s'est avéré que de futurs gains de performance viendront du calcul parallèle, et plus particulièrement de l'utilisation d'architectures multicœur. Aussi bien programmer qu'obtenir les performances optimales sur de telles architectures s'est historiquement révélé difficile.

De nombreux outils ont été créés afin d'aider les programmeurs à écrire des programmes corrects et performants. Les futurs en sont un exemple : une abstraction qui permet à une tâche de se synchroniser avec la fin d'une autre tâche et de récupérer sa valeur calculée. Les files sont un autre de ces outils : des canaux de communication qui permettent à des tâches, appelées producteurs, de stocker des valeurs dans un espace mémoire partagé, dans lequel d'autres tâches, appelées consommateurs, pourront récupérer ces valeurs, le tout de façon sûre.

Dans ce manuscrit nous étudions et améliorons aussi bien les futurs que les files sous trois aspects : sûreté, efficacité et simplicité. Nous fournissons les dataflow explicit futures en tant que bibliothèque. Il s'agit d'une évolution des futurs qui offre aux programmeurs une plus grande expressivité et leur permet d'écrire plus facilement des programmes qui manipulent les futurs. Une seconde contribution, FifoPlus, améliore les files. FifoPlus est une file qui embarque un modèle analytique afin de reconfigurer dynamiquement sa granularité de synchronisation. Cette reconfiguration se base sur le calcul d'une approximation du temps nécessaire pour effectuer un transfert de données selon une granularité de synchronisation. Contrairement à plusieurs outils pré-existants, l'analyse est simple et effectuée à l'exécution, ce qui rend l'outil simple d'utilisation.

Abstract

Since the end of Dennard scaling, it has become evident that performance gain will come from parallel computing and more particularly, multicore architectures. Programming and drawing maximum efficiency on such architectures has historically proven to be notoriously difficult.

Countless tools have been provided to help programmers achieve both program correctness and performance. Futures are such a tool, an abstraction that allows a task to synchronize with the completion of another and retrieve its computed value. Another tool is FIFO channels that allow tasks, known as producers, to store values in a shared buffer from which other tasks, known as consumers, can retrieve and use them, in a safe way.

In this thesis, we study and improve on both futures and FIFOs under three different aspects: safety, efficiency, and simplicity. We provide an implementation of dataflow explicit futures, an evolution of futures that grants programmers more expressivity and allows them to write simpler programs that use futures. We also show that dataflow explicit futures make it possible to optimize their transmission, which improves performance. A second contribution is an improvement on FIFOs, called FifoPlus. It is a FIFO that integrates an analytical model that allows dynamic reconfiguration of the granularity of synchronization of the FIFO. This reconfiguration is based on the computation of an approximation of the time it would take to run the entire program based on a given granularity. Contrarily to many existing related approaches, FifoPlus only requires an online simple analysis which makes it easy to use.

Remerciements

Après quatre années de thèse, il est compliqué de remercier exhaustivement toutes les personnes qui ont permis à ce manuscrit de voir le jour. Je souhaite cependant présenter mes remerciements à celles et ceux qui m'ont apporté leur plus grand soutien.

En premier lieu je souhaite remercier Yves Caniou pour m'avoir accompagné depuis ma licence jusqu'à la fin de ma thèse. Sans lui, je ne me serais pas engagé sur le chemin de la thèse. Son enthousiasme et sa passion ont été sources d'inspirations. Par extension, je souhaite remercier Matthieu Moy et Ludovic Henrio, mes encadrants, pour m'avoir offert l'opportunité d'effectuer une thèse, ainsi que pour leur patience et leurs conseils, même lors des nombreux moments difficiles. Je remercie également tous les membres de l'équipe Cash : Christophe Alias et ses fameux polyhèdres, Laure Gonnord et ses traquenards (et ses gâteaux), Yannick Zakowski et ses incohérences d'univers, et Gabriel Radanne et ses traquenards (et ses modules). Ne vous inquiétez pas, même loin de Lyon je trouverai le moyen de revenir vous hanter avec Isaac encore quelques fois.

Je ne peux remercier l'équipe Cash sans remercier tous les doctorants que j'ai pu rencontrer. Merci à Julien Emmanuel qui m'a fait suivre ses aventures sur les *fameuses* cartes BXI, à Julien Braine pour ses fascinantes discussions sur les langages de programmation, à Paul Iannetta que je ne saurais assez remercier pour tout ce qu'il m'a fait découvrir en termes de langages et les opportunités qu'il m'a offertes, à Thaïs Baudon, Nicolas Chappe et Hugo Thiévenaz le trio incroyable qui, chaque jour, trouvait le moyen de me faire rire. Prend bien soin d'Europe et d'Octojaune pour moi Thaïs.

Merci mille fois à Samuel Thibault et à Françoise Baude d'avoir accepté de relire ce manuscrit et d'avoir effectué des retours détaillés afin de l'achever dans le meilleur état possible. Merci également à Emmanuelle Saillard et à Marc Pouzet d'avoir accepté le rôle d'examinateur, et à Laurent Lefèvre et Arnaud Legrand pour leurs précieux conseils et encouragements lors des comités de suivi.

Naturellement, je tiens à remercier mes parents qui m'ont accompagné et soutenu tout au long de voyage. A mes amis Rémy, Grégory, Auriane, et Julian. Plus que quiconque au monde, je remercie Pierre qui m'a aidé à me relever maintes et maintes fois, et sans qui ce manuscrit n'aurait jamais vu le jour. Je ne saurais jamais lui en être assez reconnaissant.

And now for something different...

I do not know how common it is to thank people you have never met in person, and who maybe do not even know you exist. But, to quote The Machine "But if you meant

something to someone, if you helped someone, you loved someone, if even a single person remembers you... Then maybe you never really die. Then maybe this isn't the end at all". So here is my way of preventing the end.

To Hidetaka MIYAZAKI and Yui TANIMURA. A crown is warranted with strength, and strength you have given me, times and times again. Please, keep that sun shining upon us.

To Maggie THOMPSON who taught me the wonderful pain of healing. Climbing a mountain is no easy task, yet here I am, finally at the summit. As Grandma said, funny how we get attached to the struggle. Please take care.

To James Clinton HOWELL, of the Deltahead Translation Group, for his masterpiece *A Memory of Fire*. Old habits are hard to get rid of indeed, yet like Sebastian here I stand, having finally made my way to the ordinary world.

To all the wonderful people that trusted me to help them ride dragons: JSG, jerb, fly, DeadInfinity, tem, kittenchilly, ProjectTZ, Guwahavel, ratrat and all the others, I cannot write all your names here otherwise this would fill pages and pages, but I thank all of you equally. You gave me one last hope to hang on to as the end drew near, and you are in no small part responsible for this manuscript to exist now. Together, we will birth these thirty dragons and ride them into the future.

Pour Pierre

Contents

1 Introduction - Performance and Safety **1**

2 Bugs and Safety in Parallel Programs **5**

2.1 Ordering - Origin of Bugs in Parallel Programs and Crux of Efficiency . . 5
2.1.1 An Introduction to Ordering 6
2.1.2 Constraining Order: Synchronization Primitives 9
2.1.3 The Tension Between Ordering and Efficiency 18
2.1.4 Summary 20
2.2 Some High-Level Languages and Libraries for Writing Safe and Efficient Parallel Programs 20
2.2.1 How to Choose Between Languages and Libraries 20
2.2.2 Distributed Programming With MPI 22
2.2.3 OpenMP 25
2.2.4 The Actor Model & Active Objects 28
2.2.5 StarPU: A Framework for Task Scheduling on Heterogeneous Architectures 30
2.2.6 Skeleton Programming with SkePU 32
2.2.7 Cilk: A Multithreaded Runtime System 33
2.2.8 XKaapi: Dataflow Task Programming on Heterogeneous Architectures 36
2.3 A Study of Futures 38
2.3.1 The Encore Programming Language 39
2.3.2 Limitations of Futures 39
2.3.3 Typing and Synchronizing Futures - Explicit and Implicit. Control-Flow and Dataflow 43
2.3.4 The `forward` Construct - Delegating Resolution 45
2.3.5 The Godot System — Dataflow Explicit Futures 48
2.4 Data Streaming 50
2.4.1 A Dataflow Streaming Extension for OpenMP: OpenStream 51
2.4.2 StreamIt: A Language for Streaming Applications 53
2.4.3 FastFlow: High-Level and Efficient Streaming on Multi-Cores . . . 56
2.4.4 Futures for Streaming Data in ABS 57
2.5 Miscellaneous Tools 60

2.5.1 Distributed Futures 60
2.5.2 Message Sets 61
2.5.3 Joins 63
2.6 Summary 65

3 Dataflow Explicit Futures **66**
3.1 The `DeF` and `DeF+F` languages 67
3.1.1 Syntax of `DeF` 67
3.1.2 Semantics of `DeF` 69
3.1.3 Syntax and Semantics of `DeF+F` 71
3.1.4 The Type Systems of `DeF` and `DeF+F` 75
3.2 The forward-return equivalence 77
3.2.1 Translation from `DeF+F` to `DeF` and Program Equivalence 78
3.2.2 Branching Bisimulation between `DeF+F` and `DeF` 79
3.3 Implementation in Encore and Benchmarks 82
3.3.1 Prerequisites — The Encore Programming Language 82
3.3.2 Implementation of `Flow` 84
3.3.3 `forward*` : Typing and consequences in implementation 90
3.3.4 Encoding `Fut`[] from `Flow`[] 94
3.3.5 Benchmarks 95
3.4 Summary 99

4 Flexible Synchronization **100**
4.1 PromisePlus 100
4.1.1 Motivation 101
4.1.2 Flexible Synchronization for Arrays 104
4.1.3 Benchmarks 110
4.1.4 Summary 117
4.2 FifoPlus – Automatic Deduction of the Granularity of Synchronization . 117
4.2.1 Case Study – The PARSEC-Dedup Algorithm 118
4.2.2 A First Glance at FifoPlus 120
4.2.3 An Analytical Performance Model 122
4.2.4 The FifoPlus Libraries 128
4.2.5 Benchmarks 132
4.2.6 Summary 140
4.3 Conclusion – Granularity of Synchronization 140

5 Positioning **141**
5.1 OpenStream 141
5.2 SkePU 142
5.3 StreamIt 142
5.4 Futures for Streaming Data in ABS 143
5.5 Distributed futures 144

6 Future Work and Conclusion **145**
6.1 Perspectives 145
6.2 Conclusion 147

Chapter 1

Introduction – From Sequential to Parallel: The Quest for Performance and Safety

In order for the performance of applications not to be limited by the end of Dennard's scaling, programming has shifted towards multicore programming. Parallel programming is a paradigm in which performance, *i.e.* writing fast applications, and safety, *i.e.* writing correct applications, are difficult to bring together. This difficulty arises from the concept of races: multiple threads have conflicting accesses to the same shared resource. One solution to this problem is thread synchronization: constraining the possible orderings between some operations of two different threads.

Synchronization resolves the problem of safety. However, this usually comes at the price of performance: synchronization operations are costly. Achieving performance then becomes a matter of slowly peeling away synchronization layers until only the absolutely necessary core to ensure safety remains. This process is error-prone: removing the wrong layer of synchronization will result in an incorrect program; keeping an unnecessary one may drain performance.

Many synchronization tools have been created as time went on: atomic variables, semaphores, mutexes, condition variables, the list goes on and on. These synchronization primitives are "general" in a way, and are fitted to solve most, if not all synchronization problems. Experience shows that they are not always suited to efficiently or elegantly solve all synchronization problems, so synchronization abstractions were created. A programming abstraction is a construct made with the intent of hiding technical details so programmers can have a simple tool to achieve the desired result. More specifically, synchronization abstractions are structures that hide away the details of the synchronization and provide higher-level APIs that make writing synchronizations easier and less error-prone.

Synchronizing on the end of a task and retrieving its result is a classic synchronization problem. It can be solved using low-level synchronization primitives and a shared variable to store the result, but this solution is tedious in addition to being error-prone. This

solution needlessly adds complexity to the code. Futures are an abstraction that resolves this specific problem in a very elegant way. When programmers launch a task and need to access its result, they can bind a future to the task. The future acts as a transparent memory location in which the result of the task will be written once available. The transparency comes from the fact that the task itself is not aware of the existence of the future: it is the runtime of the application that takes the responsibility of writing the result to the future. In order to access the result of the task, the programmer can use an operation called `get` on the future. This operation waits until a value has been written in the future, and then returns it.

Another classic synchronization abstraction is the producer-consumer interaction: a group of tasks, known as producers, create data that other tasks, called consumers, will process. Thread-safe FIFO queues are an abstraction that provide this specific interaction. They offer operations called `push` and `pop`. `push` stores a data in the FIFO queue, and `pop` removes a data from the FIFO queue. If no data is available, `pop` blocks until one is available. Both these operations are thread-safe, *i.e.* they can be called concurrently.

While abstractions can be used to successfully hide low-level details and push forward a certain programming style, they do so at the expense of flexibility. Consider this example: a task that returns an array of integers and we want to access its result in order to perform some computation on it. The standard tool is to use a future holding the whole array. However, depending on what the computation does with the array, we may run into difficulties: if the consumer wants to perform an operation without dependencies on all values of the array, then it will spend a lot of time waiting for the entire array to be produced by the function. This is time lost: if the consumer had a partial result, it could have worked in parallel. Another naive approach would be to change the result type of the task to an array of futures: the task creates many different subtasks that will each produce one of the elements of array. This allows us to work in parallel with the creation of the array, but it also creates a lot of synchronization, which are costly. In this scenario, a more flexible, though lower-level, abstraction would have allowed us to synchronize on slices of the array, rather than on the entire array or on each element.

One of the foundations of our work is Godot[1]. This paper provides the theoretical foundations for a new kind of futures, called dataflow explicit futures. These futures are equipped with a synchronization operator called `forward*`, which is an optimized resolution operator.

Objectives In this thesis we explore middle-level abstractions. They offer more flexibility than high-level synchronization abstractions, yet provide more guarantees than lower-level abstractions.

Contributions In this thesis, we focused on the creation of constructs and libraries that offer easy-to-use, yet intuitive, efficient and safe APIs. Our contributions are as follows:

- We provide an implementation of dataflow explicit futures. Their theoretical foundations were provided in [1], with a partial implementation in Scala. Our

implementation is done in the Encore language. [1] conjectured that using the `forward*` operator on dataflow explicit futures was equivalent to using `return`. We provide a proof of this equivalence. Furthermore, we provide a set of benchmarks that showcase the efficiency of dataflow explicit futures.

- We provide two synchronization tools to work on arrays and streams. These synchronization tools have a configurable granularity of synchronization. Their objective is to allow a compromise between the cost of synchronization and the time lost waiting for a process to synchronize.
  - The first tool is PromisePlus. This tool is based on the existing promise [2] construct. We specialize promises to work efficiently on arrays, by specifying a granularity of synchronization. Our promises are able to perform synchronization on chunks of an array, rather than on single elements or on the whole array only. We provide a benchmark that shows this improves performance, and that performance varies with the granularity of synchronization.
  - The second tool is FifoPlus. This tool takes the idea of configurable granularity of synchronization we developed in PromisePlus and applies it to FIFO queues. Furthermore, we created an analytical performance model that can be used to predict the time it would take to produce, transfer and consume a given quantity of items using FifoPlus when configured with a given granularity of synchronization. By measuring the time taken to perform various operations at execution, this model can be used to automatically reconfigure a running FifoPlus with the theoretical optimal granularity of synchronization. We provide a microbenchmark that shows the accuracy of our model and the performance gain brought by FifoPlus, and a benchmark inspired by a real-world compression algorithm that shows our model still performs well in an unfavorable scenario.

  Both of these tools come with easy-to-use APIs, based on the APIs of the tools they improve. Furthermore, the presentation of each of these tools is accompanied by a discussion on performance and safety, and how they were designed with these two concepts in mind.

Organisation of the Manuscript This thesis is structured as follows:

- Chapter 2 presents the concept of ordering, and how its misuse is the origin of many bugs in parallel programming. The chapter then reviews some synchronization primitives, libraries and frameworks that have been created in order to ease the process of creating safe and efficient parallel programs. Finally, the chapter explores futures and data streaming in depth.

- Chapter 3 presents our work on dataflow explicit futures, initialy published in [3]. We start with a formal presentation, the proof of equivalence between `forward*` and `return`, before presenting the work required to implement these futures in a

language. We finally provide benchmarks to experimentally show the efficiency of the construct.

- Chapter 4 presents our work on the granularity of synchronization. It first presents PromisePlus, its API and its implementation choices to achieve efficiency, initially published in [4]. We then present FifoPlus and the analytical performance model that comes with it, and, again, the API and implementation choices to achieve efficiency. We provide benchmarks to experimentally show the efficiency of both PromisePlus and FifoPlus.
- Chapter 5 compares our contributions with their closest competitors in the literature, and Chapter 6 presents future works and concludes.

Chapter 2

Preventing Bugs and Being Efficient in Parallel Programs

As we have discussed in the introduction, there is a need for abstractions that are safe, efficient and intuitive to use in parallel programming. In this thesis, "safety" will mean the absence of bugs specific to parallel programming. Section 2.1 presents the concept of ordering, which is the source of bugs specific to parallel programming, as well as primitive tools that can be used to avoid these bugs. Section 2.2 details multiple languages and libraries that can be used to avoid these bugs. Section 2.3 details futures, the synchronization tool we introduced in the previous chapter. Section 2.4 reviews the concept of data streaming. Section 2.5 presents miscellaneous tools that can be used to prevent bugs in parallel programs, that are too specific to fit into any of the previous sections. Section 2.6 concludes this chapter, and contextualizes our contributions with regards to the tools we presented.

2.1 Ordering - Origin of Bugs in Parallel Programs and Crux of Efficiency

In sequential programming, programmers are used to a variety of bugs: accessing unmapped memory, going out-of-bounds in an array, using uninitialized memory and so on. Some bugs however are exclusive to parallel programming, for instance two thread writing at the same memory location at the same time, or a thread reading from a memory location at the same time another thread writes to it. This is the kind of bugs we will be focusing on in this section. As a result, the usage of *bug* here, unless noted otherwise, will implicitly refer to a bug exclusive to parallel programming. Since the objective of this thesis is not to provide formal proofs of safety in programs, we will not provide a formal specification of bugs, nor will we focus on verifying an absence of bugs.

2.1.1 An Introduction to Ordering

Bugs in parallel programming originate from a lack of ordering, which is the source of race conditions and data races. We define each of these terms below.

Ordering Informally, ordering refers to the order in which the accesses to multiple variables in a multithreaded program are observed by the different threads. The classic example in this situation can be found in the program written in the two listings of Figure 2.1.

Figure 2.1: Motivating example for the concept of ordering

```
1 // Thread 1
2 /* 1.a) */ x = 32;
3 /* 1.b) */ y = 17;
```

```
1 // Thread 2
2 /* 2.a */ print(y);
3 /* 2.b */ print(x);
```

Assuming both `x` and `y` are initialized to zero before both threads start to run, there are four possible outputs for this program: `0 0`, `0 32`, `17 32` and `17 0`.

While the first four cases are rather intuitive, the last one can be trickier to understand. What is happening here is a consequence of either the compiler or the processor reordering operations. This is a kind of optimization that can happen at compile time (the compiler reorders instructions) or at runtime (the CPU reorders instructions). The objective of these optimizations is to reduce pipeline stall, which happens whenever an instruction is waiting the result of another, under process, instruction. These optimizations break the assumption of *sequential consistency*: reads and writes at execution appear in the order in which they were specified in the source program [5]. Table 2.1 presents the different outputs depending on the ordering of operation. Red text indicates that a reordering occured.

In a sequential program, a reordering of operations is allowed as long as the order in which they are executed produces the same observable result as if no reordering had happened; determining if a reordering is correct is “easy” to do, as long as the dependencies between operations are respected. In a parallel program, determining if a reordering is correct is much more complicated. The compiler or the CPU may not be aware that some data can be accessed by multiple threads at once, and will by default perform reordering as if the program was sequential. This may result in an ordering that is not what the programmer *planned*. A *memory model* is used to restrict which operations are allowed to be reordered before or after other operations. The *strength* of a memory model can informally be seen as how much it allows memory instructions to be reordered. *Weak memory models*, like the ARM memory model, allow the kind of reordering seen in the example above, where Thread 1 reverses the order of assignments to `x` and `y` at runtime. Weak memory models can be arbitrarily complex and allow all sorts of optimization [5]. This thesis is not focused on weak memory models, even though their existence will motivate some of our choices.

Table 2.1: Possible orders of execution and their results for the listings of Figure 2.1

1	2	3	4	Result	
`1.a`	`1.b`	`2.a`	`2.b`	17	32
`1.a`	`2.a`	`1.b`	`2.b`	0	32
`1.a`	`2.a`	`2.b`	`1.b`	0	32
`2.a`	`1.a`	`1.b`	`2.b`	0	32
`2.a`	`1.a`	`2.b`	`1.b`	0	32
`2.a`	`2.b`	`1.a`	`1.b`	0	0
`1.b`	`1.a`	`2.a`	`2.b`	17	32
`1.b`	`2.a`	`1.a`	`2.b`	17	32
`1.b`	`2.a`	`2.b`	`1.a`	17	0
`2.a`	`1.b`	`1.a`	`2.b`	0	32
`2.a`	`1.b`	`2.b`	`1.a`	0	0
`2.a`	`2.b`	`1.b`	`1.a`	0	0

Listing 2.1 above is a good example of *race condition*.

Definition 1 (Race condition)**.** *A race condition occurs when different orderings of the same set of events result in different behaviors.*

Multiple executions of such a program would yield different results, since both threads may interleave their instructions in multiple different ways. This results in different execution order of the events, therefore race condition.

Listing 2.1 also showcases possible data races.

Definition 2 (Data race)**.** *A data race occurs at runtime in a program when one thread reads or writes from a memory location at the same time as another thread writes to this same memory location, and there is no constraint enforcing either operation to wait for the other to be finished before starting.*

Here, it is entirely possible for Thread 1 to start writing to `x` while Thread 2 fetches its value in order to display it. While there are some architectures where this is not a problem, for instance x86[1], this is not the case on all architectures. This is an example of a data race: Thread 2 may read `0`, `32` or complete garbage.

The necessity of ordering When many-core systems were designed, it was decided that it would be easier to have all cores behave independently from each other. In practice it means that cores do not naturally synchronize when they manipulate shared data. As a result, all cores are allowed to perform aggressive optimization by behaving as-if they were still alone, for instance by reordering instructions, at the expense of requiring the

[1] Reads and writes to scalars are naturally atomic in x86, although there are no natural ordering constraints between them

programmer to properly perform synchronizations between cores. Synchronizing means constraining ordering. Consider the following example, which is very common. Assume `f` is initialized to `false`:

Listing 2.1: Flag problem in multithread programming (Thread 1)

```
1 // Thread 1
2 x = 42;
3 f = true;
```

Listing 2.2: Flag problem in multithread programming (Thread 2)

```
1 // Thread 2
2 while (!f)
3   ;
4 assert (x == 42);
```

What the programmer expects from this program is that the second thread will block until the value of `x` is `42`, by virtue of using `f` as some kind of flag. In practice, there are many reasons why this may not yield the proper result.

The assertion may fail, if, for some reason, the core on which Thread 1 is executing decides to reorder `f = true` before `x = 42`. Or the assertion may fail if it gets reordered before `while (!f)`. These reordering are legal in a sequential program, as there are no dependencies between the different instructions, therefore their order does not matter at all[2]. However, in the context of multithreaded programs, as discussed before, the order of the different instructions is actually critical, and the programmer may want to prevent some orderings.

However, if compilers or processors ran under the assumption that every access to a variable may actually be critical, then a whole set of optimizations would be discarded. Therefore, in many languages like C or Java, it is up to the programmer to restrict what can be done when it becomes necessary. In order to know when an ordering is legal and how to constrain possible reorderings, the programmer needs to be aware of the memory model and to have some synchronization tools at their disposal.

Oversimplification warning The above paragraph contains an oversimplification, where one could understand that read and write events are instantaneous, that there is a very specific moment where an event happens. This simplification allows for an easy, albeit flawed, understanding of ordering, where it is possible to precisely define that an event A *happens before* an event B. In reality, this is not the case, a write may be protracted over several dozens CPU cycles, either because it first went into a cache before being written to memory, or because it went into a write buffer before being written to memory. As a result, a read that *started* after a write started may complete before the write becomes observable in main memory. However, we prefer to use a simplified model of ordering as an expert-level understanding of it is not necessary nor relevant to the content of this manuscript.

In this thesis our focus point is the communication of data between multiple threads, either in the context of compute kernels composition or streaming application. There will be a need for safety, which will come from ordering constraints in order to avoid

[2]A smart compiler may even detect that the code in Thread 2 performs an infinite loop and consider this undefined behavior. The programmer may declare `f` as `volatile` to prevent this optimization.

data races. This leads us to the presentation of some tools that can be used to constrain ordering.

In a single-threaded context, all events are ordered as per the natural ordering of the CPU instructions in memory, since there cannot be an interleaving of instructions, and therefore no interleaving of events.

2.1.2 Constraining Order: Synchronization Primitives

Synchronization primitives are basic building blocks that can be used to avoid data races and prevent race conditions by constraining order across threads. We first present them briefly before discussing their use.

Definition 3 (Atomic operations)**.** *An operation O on a memory location is said to be atomic if no other operation that operates on the same memory location can have its execution interleaved with the execution of O.*

For instance, in the C++ language, the prefix increment of an integer is not atomic: `++i` is syntactic sugar for three successive operations 1) Fetch the value of `i`, 2) Increment said value and 3) Write this new value back in memory. Any other operation that involves `i` may have its execution interleaved with these three steps.

Atomic variables Variables that are "atomic" expose atomic operations. A `load` operation fetches the value of a variable in an atomic way, a `store` operation writes a new value inside the variable in an atomic way. Other operations such as *compare-exchange* or *fetch-modify* exist. The first one can be used to atomically compare the value of the variable with a given value, and store another value inside the variable if the comparison yields "equal". The second can be used to atomically modify the value of the variable, either by adding, subtracting, multiplying... another value to the value inside the variable.

For instance, `atomic_compare_exchange(x, false, true)` will compare the value of `x` with `false`, and if `x` is indeed `false`, then it will write `true` inside. All of this will be done atomically: no other read or write to `x` can be interleaved with the compare nor the write operation. The function returns a value that indicates whether the update was successful or not.

Similarly, `atomic_fetch_add(x, 1)` will atomically add 1 to the value stored inside `x`, with no other read or write to `x` possibly interleaved during the fetch-update-write sequence.

The actual semantics of atomic operations depend on the language. Since we will be working with C++, we detail a bit more the semantics in this language.

C++ atomics In C++, atomic operations can receive *ordering constraints*. An ordering constraint defines how reads and writes that happen around an atomic operation can be reordered with regard to this atomic operation. This allows the programmer to specify the behavior of an operation at a very fine level.

For instance, an atomic operation may be *relaxed*: the atomicity of the operation is guaranteed, but the ordering of the surrounding operations is not constrained. Reads and writes around a relaxed atomic operation may be reordered before or after that operation. This is illustrated in Listings 2.3 and 2.4.

Listing 2.3: Relaxed memory ordering for the flag problem (Thread 1)

```
1 x = 42
2 f.store(true, relaxed);
```

Listing 2.4: Relaxed memory ordering for the flag problem (Thread 2)

```
1 while (!f.load(relaxed))
2   ;
3 assert (x == 42);
```

Because the load and store operations do not actually specify any ordering constraint, this is almost the same as what we did in Listings 2.1 and 2.2. The only difference is that now the accesses to `f` are atomic.

Other ordering constraints exist. When one thread reads from an atomic variable and another thread writes to this same atomic variable, the threads may perform an *acquire-release* synchronization. The read specifies an *acquire* constraint, and the write specifies a *release* constraint. Reads and writes that appear (in source code) before the write to the atomic are not allowed to be reordered after this write. Reads and writes that appear (in source code) after the read to the atomic are not allowed to be reordered before this read. In addition, all writes that happen before the write to the atomic become visible to the thread that performs the load on the atomic variable as soon as the load is completed. This is illustrated in Listings 2.5 and 2.6.

Listing 2.5: Acquire-release memory ordering for the flag problem (Thread 1)

```
1 x = 42
2 f.store(true, release);
```

Listing 2.6: Acquire-release memory ordering for the flag problem (Thread 2)

```
1 while (!f.load(acquire))
2   ;
3 assert (x == 42);
```

This time, the assertion cannot fail under any circumstances. By virtue of using a *release* ordering in Thread 1, the write to `x` cannot be reordered after the write to `f`. Similarly, since we are using an *acquire* ordering in Thread 2, the read to `x` in the assertion cannot be reordered before the read to `f`.

The *relaxed* ordering does not impose ordering constraints, yet it can still be used in a parallel program. A common usage in the C++ standard library is to increment reference counters of objects that use automatic memory management, as such an operation does not require ordering between threads, merely atomicity of the operation.

The *acquire-release* ordering is not sufficient to solve all ordering problems. There are stronger ordering constraints, such as *sequential consistency* (which we will not detail here) that are sometimes necessary. Also not all race-conditions can be prevented easily only through the use of atomic variables. Worse, adding synchronizations might introduce dead-locks in parallel programs, leading to the introduction of different bugs specific to parallel programs.

Atomic variables are a rather low-level tool, more suited to be used by experts. Since

Listing 2.7: Building a mutex from an atomic

```
1  class mutex {
2    atomic<bool> _locked;
3
4  public:
5    mutex() { _locked.store(false, memory_order_relaxed); }
6
7    void lock() {
8      while (!_locked.compare_exchange(false, true, memory_order_acq
         ↪ ))
9        ;
10   }
11
12   void unlock() {
13     _locked.store(false, memory_order_release);
14   }
15 };
```

they give programmers access to very low-level mechanisms, like the underlying memory model, they can be used to improve performance in an extremely precise way. Misuse however can greatly degrade performance, or, in C++ at least, induce bugs by inducing ordering constraints that do not avoid race conditions or data races. Most programmers will use higher-level constructs, like mutexes and condition variables.

Mutex A mutex, short for mutual exclusion, is one of the earliest synchronization primitives, and a higher-level construct in parallel programming akin to a lock, still widely used today. It begins in an unlocked state, and can be locked, albeit by a single thread at a time, and then unlocked. In practice, mutexes are used to protect critical sections. A critical section is an area of code such that there are never two threads inside it at a time. This is quite common when implementing a queue from which multiple threads can pull work: one thread should not be reading from the queue at the same time as another thread.

A mutex is very similar to an atomic boolean, and can even be implemented using one. In practice, a mutex acts like a wrapper, an abstraction above an atomic boolean, that relieves the programmer from thinking about ordering constraints and allows them to think in terms of sections of code. This comes at the cost of making them more opaque and less flexible than atomic booleans themselves. Here “flexibility” refers to the control the programmer has over how the synchronization between threads is performed: an atomic variable can receive a memory order specification, a mutex only has the lock and unlock operations it comes with.

Listing 2.7 shows how to build a mutex using an atomic boolean in C++.

When constructing the mutex on line 5, the `_locked` boolean can be initialized to `false` with a *relaxed* ordering since there is no need to order operations here. In `lock`,

Listing 2.8: Motivational example for the use of condition variables: unbounded communication channel between threads

```
mutex m;

char* read(channel_t* channel) {
  char* result = nullptr
  m.lock();
  if (channel->messages.size() == 0) {
    DO SOMETHING
  }

  result = channel->messages.pop();
  m.unlock();

  return result;
}

void write(channel_t* channel, char* msg) {
  m.lock();
  channel->messages.push(msg);
  m.unlock();
}
```

the compare-exchange operation atomically checks that the value of the boolean is `false`, switches its value to `true` if so and returns `true`, otherwise the operation fails, changes nothing, and returns `false`. `lock` performs this compare-exchange until it succeeds. The `unlock` method atomically stores `false` inside the boolean, using a *release* ordering to synchronize with the threads that may be looping inside `lock`.

Mutual exclusion in operating systems In modern operating systems, mutual exclusion uses futexes rather than atomic booleans. A futex has the same property as a mutex: the futex cannot be locked by a thread if it is already locked. The key difference is in how the futex behaves when a thread attempts to lock but fails: the naive version with atomics repeats the locking attempt until it succeeds; a futex will instead attempt to lock a few times, and if no attempt succeeds, then the thread is put to sleep and the OS scheduler is allowed to schedule a different thread. This is more efficient energy-wise, and makes better use of CPU time.

Condition variable In most programming languages, mutexes are paired with condition variables that are used to perform wait operations. Consider Listing 2.8 where we try to implement an unbounded communication channel.

The `read` function reads a message from the channel, the `write` function writes a message to the channel. Since the channel is unbounded, we can add as many messages as

we want without limit. However, there is a corner case: what happens when the channel is empty and we try to read from it?

There are several solutions: return a dummy value that indicates "No message" and have the programmer try again at a later time; have the `read` method looping until there is a message available; wait for a finite amount of time before trying again. However, none of these solutions is both efficient and easy-to-use: all three revolve around the thread monpolizing CPU time until a data arrives. This is not efficient energy-wise, nor performance-wise: while a CPU core is monopolized by a thread, no other thread may be scheduled on it, even if the currently-scheduled thread is doing nothing but waiting. A more interesting approach would be to have the thread in `read` sleep until a message is available, which would allow other threads to be scheduled in the meantime. However, since `read` and `write` are critical sections, a thread cannot enter `write` while there is one in `read`.

A condition variable is the appropriate tool to solve this problem. It acts as a signalization point: a thread A in a critical section can decide to `wait` on a condition variable, giving up its ownership of the mutex protecting the critical section, and the thread will then go to sleep. The condition variable unlocks the mutex, allowing other threads to enter the critical section. Once a thread B has resolved the problem that was preventing A from progressing, it `notifies` the condition variable. Thread A wakes up, and attempts to acquire the mutex until it succeeds[3]. Listing 2.9 shows how to use a condition variable to solve our problem with the communication channel.

As we see on lines 6 and 18 through the locking of a shared mutex both functions form a critical section, only one thread can be in either at any given time. This is necessary, as they both manipulate the array `messages` of the channel through concurrent operations: insertion (`push`) and retrieval-deletion (`pop`).

The `read` function cannot proceed if the channel is empty. As such, `read` relies on `write` to help it out of this situation. The call to `wait` on line 8 in `read` has a matching notification on line 20 in `write`.

A real scenario could be as follows: thread T_r reads from the channel, locking mutex `m`, except the channel is empty. T_r must wait until thread T_w has added something in the channel. This is done by `wait`ing on condition variable `cv_empty`, which unlocks mutex `m`. At some point in time, T_w sends a message in the channel. Mutex `m` is unlocked, T_w locks it. T_w adds the message and then notifies all threads waiting on `cv_empty` that there is data in the queue. T_w unlocks `m` as it leaves the critical section. T_r is awoken, reacquires `m`, extracts its message from the queue, and unlocks `m`. All of this is safe, as well as rather intuitive and far away from low-level considerations like ordering or atomicity.

When confronted with the problem of waiting in a critical section, a condition variable is the right tool. Depending on the context, programmers will not use the same abstractions. Someone writing a HPC application will not use atomics, they would rather use a higher-level abstraction that may use atomics for efficiency. Someone writing a

[3]Reacquiring the mutex is mandatory to ensure the correctness of the program, as the woken up thread is going to return in a critical section

Listing 2.9: Mutex and condition variable at work to implement a safe unbounded communication channel between threads

```
mutex m;
condition_variable cv_empty;

char* read(channel_t* channel) {
  char* result = nullptr
  m.lock();
  while (channel->messages.size() == 0) {
    cv_empty.wait(m);
  }

  result = channel->messages.pop();
  m.unlock();

  return result;
}

void write(channel_t* channel, char* msg) {
  m.lock();
  channel->messages.push(msg);
  cv_empty.notify_all();
  m.unlock();
}
```

kernel-level scheduler will want to use low-level tools to ensure maximum efficiency and retain full control over everything that happens.

Mutex and condition variable are abstractions over atomic variables, yet there are some problems they are not the best suited to solve as-is. For instance, consider the following problem: one thread must produce a value to be consumed by another thread. A possible solution is illustrated in Listing 2.10.

While this solution is safe, it is a bit tedious to use repeatedly. The mutex, condition variable, boolean and value are globals to keep the example simple; in a real-life example they would need to be passed as parameters to the `producer` and `consumer` functions, which would be impractical. An abstraction where the programmer could simply use a `set` method to store a value in a shared-memory space and a `get` method to retrieve the value of this shared-memory space once it has been set would be much better.

Promises A promise is an abstraction that does exactly that: it acts as a shared-memory space that can be safely accessed by a writer thread and multiple reader threads, with the guarantee that the readers will never read an undefined value. The interface of a promise is rather intuitive: the `set` method is used by the writer thread to place a value inside the promise, *fulfilling* the promise, the `get` method is used by the readers to access the value inside the promise, waiting until there is one if necessary.

Listing 2.10: Producer-consumer with a mutex and a condition variable

```
mutex m;
condition_variable cv;
bool ready;
int value;

void producer() {
  int result;
  // ... Code that computes result
  m.lock();
  ready = true;
  value = result;
  cv.notify_all();
  m.unlock();
}

void consumer() {
  int result;
  m.lock();
  while (!ready)
    cv.wait(m);
  m.unlock();
  result = value;
  // ... Code that uses result
}
```

Listing 2.11 shows how a promise can be implemented in C++, and Listing 2.12 illustrates promises at work to compute the N-th Fibonnaci number in C++.

Note how the implementation proposed here uses a mutex and a condition variable to protect the shared variables `_ready` and `_value`, and how the condition variable is used to prevent a thread from reading `_value` until it has been properly set.

A rather common use for promises is to have two threads synchronize: one thread produces a value, and the other thread wants to get this value and use it to perform some computation.

Promises use safe and efficient synchronization primitives, their interface is elegant. However they do not have a guarantee of fulfillment. A promise, as its name implies, establishes a contract between two threads. One of these threads *promises* it will send a value to the other. However, promises do not necessarily enforce this contract. Some languages, for instance C++, have a runtime that is able to detect whether a promise was fulfilled or not, and raise an exception if it was never fulfilled before being destroyed. However it would be more interesting for the programmer if it could be statically possible to detect whether a promise will get resolved or not.

Listing 2.11: Typical implementation of a promise

```
template<typename T>
class Promise { // Generic
public:
  void set(T const& value) {
    _m.lock();
    _value = value;
    _ready = ready;
    _cv.notify_all();
    _m.unlock();
  }

  T get() {
    _m.lock();
    while (!_ready)
      _cv.wait(_m);
    _m.unlock();
    return _value;
  }
private:
  bool _ready = false;
  condition_variable _cv;
  mutex _m;
  T _value;
};
```

Listing 2.12: Using promises to compute the N-th Fibonnaci number

```
void fibo(int n, promise<int>& result) {
  if (n < 2) {
    result.set(n);
    return;
  }

  promise<int> f1, f2;
  thread t1(fibo, n - 1, ref(f1));
  thread t2(fibo, n - 2, ref(f2));

  result.set(f1.get() + f2.get());
}
```

Futures Futures were conceptualized by Baker and Hewitt in [6], and later named and implemented by Halstead in the Multilisp language [7]. A future is a handle to a value that is being computed. The standard use-case is as follows: a programmer asynchronously launches a task that computes some kind of value (generally through a

Listing 2.13: Fibonacci with Futures

```
int fibonacci(int n) {
  if (n < 2) {
    return n;
  }

  future<int> f1 = async(fibonacci, n - 1);
  future<int> f2 = async(fibonacci, n - 2);

  return f1.get() + f2.get();
}
```

keyword/function called `async`), and later needs that value. Here asynchronous means that the task will be executed at some point in the future, maybe by another thread or by another process. This asynchronous launch produces a future that will hold the value computed by the task, once the task is completed.

A future exposes a `get` operation that causes the calling thread to block until a value is available inside the future, at which point the call to `get` returns this value. If the value is already available when `get` is called, it is immediately returned without blocking.

A future is therefore a more "automated" promise: the programmer does not create the future explicitly, but rather its creation is associated with the creation of a task. The programmer does not resolve the future explicitly, it is automatically resolved upon encountering the return statement of the task.

Listing 2.13 provides an example on how futures are used, here in the C++ language, to compute the n-th Fibonacci.

The comparison with promises can be interesting. In Listing 2.12 we used promises to compute the N-th fibonnaci number. There are a number of differences between the two versions:

- Futures are created by the runtime, whereas promises are created manually. Futures are more automated with more guarantees, but they are more rigid in their usage.
- Promises have to be passed as parameters to the function, meaning there was no way to use `fibonnaci` without promises. Futures are more transparent: they are used inside the function, but this is an implementation detail: *how* a function does something is *generally*[4] less important than *what* it does (programming to interfaces);
- The promise version of `fibonnaci` explicitly used threads, whereas `async` can be translated to anything, from a sequential call to the creation of another process.

[4]There are of course cases where internal knowledge can be useful, in particular to perform extremely specific optimization

- The return type of `fibonnaci` is explicit (`int`) in the future version, whereas it was less informative (`void`) in the promise version.

We will discuss the differences between futures and promises in depth in Section 2.3.

The implementation of a future is similar to that of a promise, except the interface does not expose a `set` method, and there is an external tool (a function, a keyword. . .) that is used to asynchronously launch a function, which produces a future that will hold the result of the function's execution.

Futures are safe by design, and solve a problem that is quite common. As we will see later, futures form the basis, among other building blocks, of a programming model known as the active-object model.

Summary: futures and promises Futures may seem to be as expressive as promises with strictly more guarantees. However, they are also more rigid in their usage: a promise may not have a guarantee of fulfillment, but it can be resolved at any time; a future has a guarantee of resolution, but at the cost of forcing the programmer to write their code in a certain way, which we will discuss more in depth in Chapter 4. Moreover, there are some problems tied to type systems that prevent futures from being used in contexts such as terminal recursive functions. Promises are a good alternative in these situations, and we will explore this topic more in-depth in Chapter 3.

We have presented basic synchronization tools, that we will use in this thesis to constrain ordering in our tools, in order to make communication between threads safe. However, communications should not only be safe, they should also be efficient, a combination that has historically been difficult to achieve. We now discuss why there is a tension between constraining ordering and reaching peak performance while preserving correctness.

2.1.3 The Tension Between Ordering and Efficiency

While ordering is a problem that arises only when writing parallel programs, the question of efficiency arises in both sequential and parallel programs. Sequential programs have no need for explicit ordering, therefore the programmer can focus only on the algorithmic aspects of the program. Most parallel programs need some ordering constraints in order to be correct (i.e. no data races); however, operations that constrain order have a high performance cost. On multicore machines, this cost comes from the interaction between ordering constraints and the cache.

The cache and inconsistent view of memory CPU caches are really fast to read from / write to, since they are extremely close to cores themselves, compared to the main memory. To further improve performance, CPU cores usually have multiple levels of cache. As the level rises, the time required to read or write to the cache increases. However, higher-level caches are also cheaper to manufacture and have a higher memory capacity. Level 1 (the fastest and smallest) is usually individual and each core has its

own. Higher-levels are sometimes shared by multiple cores. While using caches allows for efficient data access, it also gives each core a different view of memory. Two cores may read the same memory location and get different values, for example because one may fetch data from main memory and the other from a cache. If such behavior is not desired, the programmer must enforce a consistent view of memory by imposing an order by using one of the aforementioned synchronization tools.

The cost of synchronization We have stated synchronization is a cost-heavy operation. There are two reasons for this. The first is that synchronization usually involves one thread waiting for another thread. The second is that synchronization operations constrain ordering, which is detrimental to performance by nature.

Waiting Synchronization generally relies on one thread waiting for another thread before doing something, *e.g.* because of a mutex. A wait can be necessary to ensure the correctness of the algorithm, however if it waits longer than necessary, *e.g.* because the synchronization is not well designed, then it incurs a loss of performance.

Cost of ordering Ordering constraints correspond to CPU instructions called *fences* that force a core A to commit some changes to memory. In addition, executing the fence may invalidate some buffers and some parts of the cache. The exact behavior is architecture-dependant and as such is not detailed in this manuscript. A result of cache invalidation is that the next access to data that was flushed will trigger a cache miss, and force this data to be reloaded from main memory. This reduces performance as memory accesses are slower than cache accesses.

In general, as a synchronization tool gets higher-level, it becomes safer yet more costly to use. For instance, a mutex is easier to use correctly than an atomic: no ordering constraint needs to be specified. On the other hand, a well-used atomic may be faster.

On wait-free and lock-free structures and algorithms Designing and using synchronization tools is a difficult task, where the optimal solution is difficult to find. Sometimes it is possible to write algorithms and structures that require ordering, without using locking mechanisms, like mutexes, which may result in better performance. However writing such algorithms is quite difficult. These algorithms that do not perform any kind of locking operation are called *lock-free*. Related to that are *wait-free* algorithms, where not only no thread performs a locking operation, but also every operation executed is guaranteed to execute in finite (bounded) time. We will look at an example of wait-free structure in Section 2.4.3.

In summary, because synchronization and ordering are complex topics, it is best if the programmer naturally has access to abstractions that are both efficient and safe. In this thesis, in particular in Chapter 4 we focus on low-level details that are relevant to the problems we want to solve, while providing high-level abstractions. Low-level details will bring performance; the higher levels of each abstraction will create an interface that

hides these details from the programmer, and ensures they can only use the tool in a way that preserves safety. Furthermore, we will make sure the low-level details benefit from information provided either by the programmer or deduced at runtime in order to improve performance even more.

2.1.4 Summary

We have seen multiple tools to perform synchronization between threads, and constrain ordering to avoid bugs in parallel programs. There is a tension between the ease-of-use of these tools and the performance and safety they bring. Atomic variables give access to extremely fine-grained optimization, but misuse can have consequences ranging from degraded performance to outright bugs. Moreover, they are ill-suited to solving some problems. Higher-level tools like mutexes and futures are more suited to solve some other problems. While higher-level tools usually have safety guarantees, performance is not always assured. And, as seen with the futures, there is a trade-off between ease-of-use and flexibility as well.

However, these tools are not the only ones that exist. There are frameworks and programming languages dedicated to solving problems in parallel programming. Similarly to synchronization primitives, they all have to walk the thin line between ease-of-use, flexibility, performance and safety. We will present those that are the closest to what we do, and study some of their design choices in order to inform our own.

2.2 Some High-Level Languages and Libraries for Writing Safe and Efficient Parallel Programs

In the previous section we presented different synchronization constructs to synchronize threads and constrain ordering inside parallel applications, and how bugs in parallel programs often originate from a lack of synchronization between threads. We also presented some tools that can be used to constrain ordering between threads and potentially remove these bugs. While there are some low-level tools, it is also common for programmers to use dedicated libraries or dedicated programming languages to write safe and efficient parallel programs. In fact, one of our contribution is a language construct (Chapter 3), while the other two are libraries (Chapter 4).

In this section we will first present what libraries and languages are and the pros and cons of using a language or a library. We will then move on to present multiple languages and libraries that offer programmers tools for parallel programming and what they have to offer with regard to performance and safety.

2.2.1 How to Choose Between Languages and Libraries

When trying to solve a problem, programmers are exposed to languages (C++, Java, Encore [8]), libraries (pthread, StarPU [9]), or intermediates (the GOMP implementation of the OpenMP standard). We study below the different approaches to implement a new

programming model or a new programming abstraction like the ones we will propose in this thesis. We also study the advantages and drawbacks of library-based approaches relatively to full-fledged languages.

The choice between a language, a library, or an intermediate solution to solve a given problem is not exclusive to parallel programming. The examples in this section will mostly focus on parallel programming, but the arguments made are not specific to a paradigm.

Libraries A library is a set of functions and language entities[5] that provide a set of tools to the programmer. For instance, the pthread library allows programmers to write multithreaded applications in C and C++. Libraries are interesting because they are written in an existing language, meaning a programmer fluent in said language will not have to learn new syntax in order to use the library. Additionally, maintenance of a library is not too difficult, even when the host language evolves. On the flip side, since libraries exist within a given language, they are inherently constrained by what the language and its ecosystem offer. For example, a data-streaming library in C++ will not benefit from aggressive optimizations targeting stream computing because such optimizations are too specific for a general-purpose language.

Languages Languages are created in order to provide a way to write programs. Some languages are general purpose, others target a specific application domain or a specific way to program. Each language has its own abstraction level and inherent properties. For example, C++ is a rather low-level language which gives a lot of freedom to the programmer, at the cost of letting them write unsafe programs. Rust is also a low-level language, but it inherently constrains how programmers code in order to prevent the appearance of some bugs, such as out-of-bounds accesses. Different languages may also be more or less expressive, with dedicated constructs or keywords that act as syntactic sugar to make writing programs easier.

Just like there are general-purpose languages, there are also languages dedicated to solving a specific category of problems: StreamIt [10] for instance is dedicated to data streaming.

Most languages come with a compiler or a runtime, or both, and in the case of a language dedicated to solving a specific category of problems, compilers and runtimes can be tailored to perform aggressive optimizations on the code, optimizations that would be seen as too specific to fit in a more general language.

While libraries can let programmers stay in their comfort zone if they are already familiar with the language in which the library is written, switching to a new programming language can be more demanding: a possibly new syntax as well as a new programming model may need to be learned before being able to write "good", *i.e.* safe and efficient, code with ease.

This possibility to have dedicated syntax and optimizations comes at a price. When providing a new language, the language developer has to implement a new compiler

[5] For instance, classes in an object-oriented language, monads in a functional language. . .

(or several of them), possibly generating code for an existing language, which is costly. Maintaining such compilers is in general more difficult than maintaining a library.

Intermediates There are also intermediate approaches placed between the pure library approach and the design of a new programming language, like OpenMP. It is both a C library with C functions and an extension of the C language through the use of `#pragma` directives. Such intermediate tools offer some kind of middle ground. They can keep programmers close to their comfort zone, like libraries, while still requiring them to learn a possibly new syntax, like languages.

Summary Libraries keep programmers in their comfort zone at the expense of performance since they cannot perform as aggressive optimization as dedicated languages. Their expressiveness is also limited by what the language offers: a library will usually not introduce new keywords or language constructs. The safety guarantees in a library are also more oriented towards runtime guarantees rather than static guarantees compared to language approaches.

Languages can be more or less expressive with keywords or static constructs acting as syntactic sugar, can offer aggressive optimizations, but this comes at the cost of a steeper learning curve.

Intermediate approaches stand between both: they keep programmers close to their comfort zone, but still require them to learn syntax extensions, and the possible optimizations, while generally more aggressive, are still constrained.

The choice boils down to the amount of time needed by a team to get familiar with a given tool and the kind of guarantees they are looking for. The cost of development of each solution also depends on the approach, libraries being in general easier to maintain.

We now move on to present some libraries, languages and intermediates that target parallel programming, through the lenses of efficiency, safety and ease-of-use.

2.2.2 Distributed Programming With MPI

MPI (Message Passing Interface) [11] is a specification that is implemented as libraries in different languages, such as C, dedicated to performing distributed computing; it is, alongside the OpenMP language-library, one of the historical tools in parallel computing. All the examples in this section are written in C.

MPI exposes functions that allow programmers to run multiple instances of a same program on multiple different machines, and have all these instances communicate with each other. Communication is performed through the exchange of messages. Unlike in a protocol like HTTP where a message (a request in HTTP) expects an answer (a response in HTTP) and no further messages can be sent before receiving a response, an MPI program may send a message without expecting a response. If a response is expected, the program is still free to keep running until it actually needs to wait for the response, which may be at any point in the future.

Listing 2.14: Short MPI example

```
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int value = rand();
    MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("Sent %d to process 1\n", value);
  } else {
    int value;
    MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, NULL);
    printf("Received %d from process 0\n", value);
  }

  MPI_Finalize();
}
```

In this section, we will focus on the OpenMPI [12] implementation of the MPI standard in the C language.

Let us go into more details. In order to run an MPI program, programmers use the `mpirun` command-line tool. It takes the program name and arguments as parameters, as well as the number of instances of the program that should be ran, *e.g.* `mpirun -np 3 ./a.out` will run 3 instances of the `a.out` program. These instances may run on the current machine, or on multiple different machines.

Communication between instances of an MPI program are done through *communicators*. All instances that share a common communicator can exchange message with each other through this communicator. When an MPI program is launched through `mpirun`, all instances share a global communicator, called `MPI_COMM_WORLD`; the programmer may then proceed to create additional communicators in order to represent groups of processes and refine which communications are allowed. In order to identify themselves, instances are assigned a unique *rank*. This rank is used in the MPI messaging functions, `MPI_Send` which is used to send a message, and `MPI_Recv` which is used to receive a message. `MPI_Send` takes as parameter the rank of the receiver, `MPI_Recv` takes as parameter the rank of the emitter.

Listing 2.14 gives a small example where one process sends a number to another.

Since the main objective of MPI is to exchange messages, let us see how the programmer expresses sending a message and receiving a message. Both `MPI_Send` and `MPI_Recv` have verbose signatures. The first parameter of `MPI_Send` is the data to send, passed by address. The programmer then specifies the number of elements in the data buffer, as well as their type, and, finally, the identifier of the target process (its *rank*) as well as which communicator to use.

There are two categories of observations here: some that are tied to MPI itself, and some that are tied to the choice of the C language. On the MPI-specific side, we can see that processes are abstracted by a number and a communicator. Such an abstraction is less informative and more error-prone than abstracting them as instances of a full-fledged datatype that would expose dedicated operations for sending and receiving messages. On the C specific side, there are some observations tied to the type system of the language. `MPI_Send` and `MPI_Recv` need type information passed as parameters because the type system of C does not offer genericity that is type-safe. This is an illustration of a limitation of libraries: they are limited by what the host language offers.

However, MPI still offers great abstractions. For instance, the `MPI_Send` function might be type unsafe but it is a really good abstraction over the standard POSIX `send` function in the sockets library. To elaborate, we did not need to check the endianness of the computer before sending an integer, possibly through the network, to another computer. If we were using the socket library, we would need to convert the machine representation of the integer to the network representation, and then convert the received integer back to the representation of the receiver machine. Both `MPI_Send` and `MPI_Recv` abstract this away. Additionally, even if the programmer needs to specify the amount and the type of the data to send/receive when working with MPI, it is an improvement over the socket library where the programmer would have to specify the number of bytes to read/write. MPI can abstract away details like getting the exact size of an `int`, which is machine dependent.

MPI is also massively used for its collective operations that allow the synchronization of many processes.

Summary MPI is an historical tool, that serves as the main building block of many abstractions that have been built over the years in an effort to make distributed computing and message passing easier and more efficient. This paragraph does not intend to work as positioning, but merely as a summary.

MPI is focused on distributed computing, which is not our focus point: we target parallel computing on a single machine. However, our results may be extended to fit in the distributed paradigm. Nonetheless, interesting observations can be made on some of the design decisions in MPI and what they entail in terms of safety, ease of programming and efficiency.

Since MPI is focused on distributed computing, it limits data races: memory is not shared, each process has its own memory, similar to multiprocessus programming on a local machine through `forks`. Nevertheless, it is still possible to encounter data races when working with the asynchronous versions of `MPI_Send` and `MPI_Recv` that may work in parallel with the main program, and read / write to their buffer in parallel with the main program. Regardless of all of the above, MPI does not eliminate the problem of race conditions.

Ease-of-use is not ideal, however libraries have been made to improve it. DSPARLIB [13] is a C++ wrapper above MPI that exploits the tools of the language, namely *templates*, in order to provide an easier to use API. The active-object model that we

mentioned earlier completely abstracts away the idea of message sending by instead using asynchronous method calls between objects.

If we consider the efficiency of MPI, the *implementation* of the functions in the MPI standard are usually efficient. They do not specifically target data streaming or kernel compositions, however having efficient implementations of the standard MPI functions allows higher-level, or even lower-level tools, to achieve efficiency when it comes to communication between machines.

Again, we do not target MPI nor distributed systems in this thesis, but we share some objectives with MPI. MPI has brought some ease of use to programmers and high-level abstractions.

2.2.3 OpenMP

OpenMP (Open Multi-Processing) [14] is a standard for multithreading that has been implemented in many languages, such as the C language. It is, alongside the MPI library, one of the historical tools in parallel computing. We will focus on C implementations of OpenMP in this section.

An OpenMP program is a C program extended with OpenMP pragmas directives, *e.g.* `#pragma omp parallel`. An OpenMP pragma defines the behavior of a part of the application, and can be annotated to alter its behavior. Since these pragmas are not part of the C standard, an OpenMP program needs to be compiled with a compiler that has support for it built-in.

The most common use of OpenMP is to have multiple threads execute a block of code in parallel, on the same machine, which can be achieved through the `parallel` pragma. The second most common use of OpenMP is to share the execution of a `for` loop between multiple threads of execution, through the `for` pragma. We now present some common pragmas.

The `parallel` pragma When the flow of execution reaches the `parallel` pragma, a number of threads is spawned and they all begin executing the content between braces, and then wait for each other at closing brace, waiting on an implicit barrier[6]. The number of threads can be left to the runtime, or explicitly stated through the `num_threads` annotation. Listing 2.15 presents a simple use of the `parallel` directive. The two calls to `printf` have been willingly split in order to show the interleaving of instructions during execution.

[6] This barrier can be disabled using the `nowait` annotation

Listing 2.15: Example use of the `parallel` pragma in OpenMP

```
int main() {
  #pragma omp parallel num_threads(4)
  {
    printf("Thread ");
    printf("%d executing\n", omp_get_thread_num());
  }

  return 0;
}
```

There are multiple possible outputs. Some can be of the following form if all threads manage to execute both calls to `printf` before any other can interleave a call:

```
  Thread A executing
  Thread B executing
  Thread C executing
  Thread D executing
```

Some can be of the following form if multiple threads interleave their calls to `printf`:

```
  Thread Thread B executing
  A executing
  Thread Thread C executing
  D executing
```

The set of all possible outputs if not shown for brevity.
These outputs are with A, B, C and D taking the values of all possible permuations of the set { 0, 1, 2, 3 }.

The `for` pragma The `for` directive is used to distribute the iterations of a loop between multiple threads, in order to speed up its execution. The programmer can specify how the iterations of the loop will be scheduled between the different threads through the `schedule` annotation. Listing 2.16 presents a simple use of the `for` directive. It should be noted that the `for` directive is only allowed to appear inside a `parallel` region[7], since it is a worksharing construct: it inherently requires multiple threads to exist in parallel in the executing context.

OpenMP tasks A common use of parallel programming is the scheduling of work between multiple cores. As a result, OpenMP provides multiple pragmas that can help the programmer distribute work between multiple threads; these are called work-sharing constructs. We have already seen the `for` pragma in the previous section; here we will focus on the `task` pragma, as it will come back later when we talk about the OpenStream library.

An OpenMP task is a block of code that begins with the `task` pragma, and is delimited by braces. Like with the `for` pragma, tasks are only allowed to appear inside a `parallel`

[7]The pragma is sometimes abbreviated as `#pragma omp parallel for` for brevity sake

Listing 2.16: Example use of the `for` pragma in OpenMP

```
int main() {
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for schedule(static)
    {
      for (int i = 0; i < 16; ++i) {
        printf("Thread %d executing iteration %d\n",
               omp_get_thread_num(), i);
      }
    }
  }
}
```

region. At runtime, whenever a thread reaches an OpenMP task, the task is added to a pool of tasks to be executed (note that if multiple threads reach the same OpenMP task, then the task is put in the pool one time per thread). Then, when the runtime decides that it is best, it selects one of the threads that were created, and assigns the task to this thread for execution.

Synchronization Since OpenMP uses a shared-memory model, data races will happen when multiple threads read and write to the same variable without performing synchronization. Therefore, there are two relevant questions to ask: 1) What are the constructs offered by OpenMP in order to synchronize threads? and 2) How can they safely and efficiently interact with each other?

There are three different synchronization tools inherently offered by OpenMP: barriers, critical sections, and atomic operations with specification of an ordering constraint. Barriers synchronize all threads with each other. Critical sections prevent two threads from executing the same section of code at the same time, similarly to mutexes. And atomic operations can be used to constrain ordering, as we have already discussed in Section 2.1.2.

Barriers and critical sections are rather easy to use. A barrier is a `#pragma omp barrier` at the point in code where the programmer wants to put a barrier. A critical section is defined by `#pragma omp critical` blocks of code. The efficiency of the implementation of either is up to a given implementation of the OpenMP standard. Atomic operations in OpenMP suffer from the save caveats as atomic operations in modern C++, where specifying the wrong ordering can create incorrect code.

Positioning As we can see, OpenMP offers its own version of most of the primitive synchronization tools that we have discussed in the previous section. However, it lacks more evolved synchronization tools. For instance, we will talk about the LU Gauss-Seidel algorithm in Chapter 4, and its implementation in a set of benchmarks. In this

implementation, there was a need for a synchronization pattern based on an alternate bit protocol, which had to be implemented manually by the programmers, using only OpenMP basic building blocks. Rather than being abstracted away in a class or structure or module, the pattern was implemented with an array and atomic operations, which made it quite difficult to understand. Standard structures for communication between threads like FIFO queues do not exist in OpenMP, and it is up to the programmer to create them. This also means that efficiency of communication is irrelevant here: OpenMP does not offer tools for communication, so their efficiency cannot be judged. In Chapter 4 we will present PromisePlus, an abstraction to exchange arrays efficiently in an OpenMP program.

2.2.4 The Actor Model & Active Objects

During our presentation of futures in Section 2.1.2, we mentioned that they form one of the basic synchronization construct of the programming model known as "active objects". In this section we first present the actor model, which is another building block of the active-objects model, before moving on to the active-objects model itself.

The actor model The actor model was developed by Agha [15] in the 1980's, with the intent of making distributed and parallel programming both easier and safer than what existed then, *i.e.* synchronous communication models based on either process calculi like CSP [16] and CCS [17], or on shared-variable models like C.

The actor model is a programming model that manipulates actors: single-threaded entities that each have an associated message queue and data. Communication between actors is not performed through shared data, but rather through *asynchronous* messages, *i.e.* upon sending a message an actor does not wait for the answer and proceeds immediately. Internally, an actor performs pattern matching over the different messages in its message queue, and works depending on which messages it has received.

This approach, where each actor is a self-contained mono-threaded entity and all communications are reduced to message passing, removes data races[8]. In the case of parallel programming, it abstracts away the need for synchronization between threads which allows programmers to focus on algorithmic aspects, unlike in OpenMP where synchronization for communication is explicit. It also makes static analysis and model checking much easier than in the shared-variable model of parallel programming, as it ties together data and control: an actor only manipulates its own data, and never touches data from another actor. However, it still leaves the possibility for race conditions: the behaviour of the program depends of the order in which messages are processed, and a race occurs if this order is not enforced. Additionally, actors still suffer from a kind of starvation, where some messages may be ignored if the actor does not switch to a state in which it can process these messages.

[8]The actor model does not specify whether the parallel programming model is based on shared memory or on distributed memory. If the model is a shared-memory model, then communication between actors is thread safe as long as the communication channels are thread safe

In Agha's model, an actor is a kind of black box: how a message is processed depends entirely on the actor. You can see it as a class with a method `work` that loops over the messages in its queue. The exact behavior of this method and what it does depending on the messages it proceeds is opaque, and the model does not allow to specify the behavior of an actor, *e.g.* by defining an interface. A consequence of this design is an absence of separation of concerns: what a message does is not tied to a contract, but to implementation. Similarly to how the object-oriented paradigm was developed over the last decades and introduced modules and classes in order to structure programs and loosen coupling, the active-objects model of computation was introduced to address some of the flaws in the original actor model.

The active-objects model The active-object (AO) model [18] takes inspiration from the object model: actors, now called active objects, are instances of classes, that may support object features such as inheritance, encapsulation and methods; message calls are now asynchronous method invocations[9]. A key feature of active-objects languages is the use of futures (Section 2.3): every asynchronous method invocation produces a future that will hold the result of the execution of this method. This future allows an object to synchronize with the end of the execution of the corresponding method.[10]

The inspiration from the object model solves the lack of separation of concerns: each method offers its own contract. An asynchronous call to a method has inherently more meaning than sending a message to an actor. Another consequence is the removal of the starvation that could occur in an actor if it never switched to the appropriate state to process a message. In the AO model, each object has a FIFO queue of pending method invocation, and processes them one after another. Provided every method call finishes, the object will eventually process all method invocations in a bounded time.

Another interesting feature of the active-objects model is the ease of synchronization. In the actor model, a programmer could not easily synchronize with the answer to a message. However, in the active-objects paradigm, thanks to the use of futures, synchronization is performed through a simple `get` statement on a future produced by an asynchronous call.

By nature, active-object languages lend themselves to all kind of optimizations.

- *Cooperative scheduling* Some of the active-object models allow the programmer to use cooperative scheduling, in general through the keyword `await` that can be applied to a future. `await` checks whether the future is resolved. If it is, execution continues; if it is not, then the active-object suspends the execution of the current asynchronous call and tries to process another. A specific example can be found in the ABS language [19], that offers a concept of Concurrent Objects Groups (COGs), that partitions active objects into COGs, having only one active object in each COG running; in this case, yielding control with `await` can transfer control to another object in the COG.

[9] In this context, asynchronous means that the thread does not wait for the method to run before resuming execution.

[10] Whenever an object calls an internal method, the call is synchronous, naturally

- *Speculative execution* The Encore [8] language offers monadic combinators [20] that can be used to perform speculative computations: multiple computations are launched in parallel, each of them yielding a future, and the programmer can wait on this group of futures until the first one is resolved, and then cancel all other tasks in the group. This can be used, for example, to try multiple strategies to solve a given problem in parallel and keep the result of the fastest one.

Thread safety is a common point of interest when working within the active-objects model, since it is a model oriented towards working with thousands of parallel entities. Many languages, like the Encore language [8] have a "no shared variables" policy, which is enforced by the type system of the language. In Encore, the type system offers "capabilities" [21], which are similar to *ownership* [22] in Rust: a capability restricts the operations allowed on a value of a given type. The most prominent use of capabilities in Encore can be found with the active-objects themselves: the internal state of an object cannot be shared with any other object. This greatly limits data races.

As active-objects languages target HPC, they also focus on efficiency [23]. A more detailed view of how active-objects languages achieve efficiency can be found in [18].

Summary The active object and actor models of computation are an extremely active field of academic research, as well as the inspiration for some industrial success, like the Akka [24] framework, which adds actors in the Scala and Java languages. Considering the extremely wide scope of the models, we will not position ourselves relative to them. However, since parts of our work are done in the active-object language Encore and target efficiency and ease-of-use (see Chapter 3), and others are inspired by the usage of futures for data streaming in the active-objects language ABS, we will position ourselves relative to these two languages in Chapter 3 and Section 2.4.4 respectively.

2.2.5 StarPU: A Framework for Task Scheduling on Heterogeneous Architectures

StarPU [9] is a library that focuses on scheduling tasks on heterogeneous multicore architectures, with support for distributed computing. A heterogeneous architecture is one that contains multiple kinds of processing units: CPU, GPU, MIC.... Working on heterogeneous architectures is one way to improve the performance of applications: nowadays, GPUs can be used for general-purpose computing, and are particularly well suited for massively parallel ones since GPUs excel at parallelism[11]. Programming on heterogeneous architectures with multiple different kinds of processing units is quite recent, and StarPU was motivated by the need for efficient scheduling runtimes for these kinds of architecture.

In StarPU, the basic unit of work is a *task*. A task is a set of functions that all implement the same algorithm albeit on different architectures. For instance, a matrix multiplication task may be implemented as a function that runs on a CPU using standard

[11]Although this comes with a steep learning curve as programming on a GPU is wildly different than programming on a CPU

BLAS, and as a function written in CUDA to run on an NVIDIA GPU. A programmer will conceive their program as a graph of tasks, and will either specify the dependencies between task or let the runtime determine them automatically, and then gives this graph of tasks to the scheduler. At runtime, the tasks will be scheduled on different computing units, while respecting dependencies. The choice of the computing unit is up to the runtime, without the need for an intervention of the programmer.

StarPU supports multiple scheduling algorithms. They can be guided by scheduling hints provided by programmers, *e.g.* giving a priority to a task. StarPU also focuses on performance models (for instance the ATLAS [25] model for the BLAS kernel) which are used to get an estimate of how long a given task will take to complete on a given computing unit, based on the current workload of this computing unit. Using such a model can drive the scheduling by dynamically finding the best computing unit on which to run a task. In addition to hints and performance models, StarPU can also drive its scheduling by exploiting the results of previous executions of a task: the runtime records the time it takes to run a given task on a given computing unit in order to have a baseline with which it can inform future scheduling decisions.

This allows the runtime to perform smart scheduling decisions: if the best computing unit for a task is available, then the task can be scheduled immediately. If the best computing unit is not available, but the second best is, then the runtime can decide whether it is worth immediately scheduling the task on the second best, or if it is more interesting to wait until the best computing unit is available.

StarPU does not support communication between running tasks, except upon task creation and termination. If A and B are two tasks and B specifies that it depends upon A then the runtime will only schedule B once A has completed. StarPU aims first and foremost to offer a unified execution model and a scheduling framework on a heterogeneous architecture, communication between tasks is therefore not a priority w.r.t. StarPU objectives. This can also be explained by the difficulty in communicating between multiple different computing units.

Positioning StarPU strays quite far away from our objective, that is improving the synchronization between computation kernels in order to improve efficiency, however what it does with the recording of the execution time of tasks is interesting for us. One of the idea we will explore in Chapter 4 is to slice the data into chunks, and finding the best size for these chunks. Recording the chosen size for these chunks and the execution time associated could give us heuristics to find the best granularity.

However, there is one pitfall with the StarPU approach for exploiting execution time: the runtime cannot deduce on the fly on which computing unit it should schedule a task when it receives this task for the first time, and because of the cost of moving data between computing units the scheduling decision cannot be reconsidered once the task has been scheduled. As a consequence the runtime needs multiple runs of a given task in order to find out, experimentally, which computing unit is the best for this specific task. In turn, this means that the task should lend itself to repeat executions whose execution time is not highly dependent on the input. Algorithms that are dataflow or

have an execution time highly dependent on the input are not the best target for StarPU. In Chapter 4 we will present a tool that takes inspiration from StarPU's measurements and uses them to improve the efficiency of communication between multiple threads.

2.2.6 Skeleton Programming with SkePU

SkePU [26] is a library for algorithmic skeletons [27] programming. An algorithmic skeleton is a higher-order function whose body is inherently/easily parallelizable, such as many constructs inspired by functional programming, like *map*, *reduce* or *scan*. A SkePU program can then be seen as a sequence of calls to skeletons with user-supplied functions, that performs a series of transformations on some input.

Similarly to StarPU, SkePU works on heterogeneous architectures. In SkePU, processing units other than "traditional" CPUs are called *accelerators* (*e.g.* GPUs). In order to work with these accelerators, SkePU supports multiple *backend*s. A backend is a library used to communicate with an accelerator and work with it. For instance, if one wants to work on an NVIDIA GPU outside of SkePU, they may use NVIDIA's proprietary library *CUDA*, the high-level library *Boost.Compute* part of the Boost project, or the low-level library *OpenCL*. When working with NVIDIA GPUs, SkePU supports both the OpenCL and CUDA backends. What this means in practice is that the programmer may write a function to be run in a skeleton using either OpenCL or CUDA, and SkePU will seamlessly work with either, which gives some freedom to the programmers as to which tool they use.

The use of skeletons allows programmers to focus on the *structure* of the application they are writing, the sequence of transformations they want to perform on a given input, rather than focus on questions like "I have a GPU here, and a CPU here, would it be better to execute on the GPU? If so, how do I perform the transfer? How do I adapt it if I change the library I use for communication with the GPU?". There are also multiple internal features of SkePU that reduce the concerns of programmers:

- The containers in SkePU are "smart containers" [28, 29]. A smart container is akin to an array / a matrix that is able to seamlessly offload parts of its content on different computing units, while also keeping track of where each part is located. One of the advantages of this technique is that it allows the SkePU runtime to avoid needless transfer of data between multiple computing units: if two skeleton calls that operate on the same smart container are executed on the same computing unit with no other skeleton reading from this smart container, then there is no need to transfer the container's content back to main memory;

- Every time a skeleton is invoked, the runtime records the call site as well as the arguments, but does not immediately run the skeleton. Internally, the runtime will build a graph, called a lineage [29], which allows the runtime to perform lazy evaluation, deferring the actual evaluation of a skeleton to the moment it is actually needed. This allows the runtime to perform some optimization like loop fusion or loop tiling by exploiting knowledge of the sequence of calls.

- The two optimizations above improve locality of reference, which usually increases performance.
- Lastly, SkePU is able to perform an auto tuning [30, 31] of the skeletons, where the runtime will automatically deduce which backend is the best to run a given skeleton / a given set of skeletons. This process exploits the lineage of skeleton calls in order to follow the sequences of transformations performed on each smart container. With this additional information, the runtime is able to better compute the cost of executing a skeleton on a given computing unit.

All of these features alleviate the work of the programmers, who can focus entirely on the sequence of transformations they want to perform. Writing a SkePU program is therefore really close to sequential programming.

Positioning SkePU's scope is extremely wide: it supports hybrid CPU-GPU programming, with high-level abstractions that successfully hide away most of the underlying technical details so that programmers may focus entirely on the structure of their program while getting strong performance results. Our scope is much more narrow: we focus first and foremost on providing safe and efficient communication tools between threads that, for now, run on CPU, rather than on a hybrid architecture.

Smart containers in SkePU take the form of arrays or matrices which are one of our focus point. Our primary objective is to improve the composition of kernels that work on such structures. One of our contributions that we will elaborate upon in Chapter 4 is a tuning algorithm that finds a good granularity of synchronization when exchanging arrays and matrices between two kernels. Such a tool could possibly be used inside SkePU in order to improve the efficiency of the communication between skeleton calls by finding a good granularity. In particular, this could be applied to the algorithm in SkePU that performs loop tiling: finding a good size of tiles will improve performance. It could be possible to exploit the capabilities of the SkePU runtime, such as the lineage of skeleton calls, in order to defer some of the configuration of our tools to the runtime itself, which would further improve ease-of-use. We will explain this in more details in Section 4.2.

The choice of the backend in SkePU is done through an offline machine-learning algorithm [31]. In one of our contribution we propose an algorithm that finds a good granularity of communication between kernels. While the approaches attempt to find two different kind of values (SkePU attempts to find where to run a function, our tool attempts to find a size), they both have in common a desire to increase performance and improve ease-of-use by auto tuning a tool given to the programmer.

2.2.7 Cilk: A Multithreaded Runtime System

Cilk is an extension of the C language as well as a runtime system. It was first proposed by Blumofe et al. in [32] and evolved until the Cilk 5 release in [33] by Frigo et al. A Cilk program is a C program augmented with extra keywords, therefore requiring a dedicated compiler in order to translate Cilk to C before invoking the C compiler.

In Cilk, the basic unit of work is a *procedure*. A Cilk *procedure* is a C function annotated with the keyword `cilk`. Unlike C functions, a Cilk procedure cannot be called as-is, and instead requires the programmer to prefix the call to the procedure with the keyword `spawn`. When a procedure is `spawn`ed, the runtime schedules its execution on one of the processors of the machine. This is done through a system of double-ended queues (deque): Cilk assigns each processor a deque that will hold the procedures this processor will execute, one after the other. Note that the procedures spawned inside a procedure may not necessarily be scheduled on the same processor as the one executing the procedure that spawned others.

Listing 2.17 presents Cilk code to compute the nth Fibonacci number in parallel. It is taken almost verbatim from the example provided by Frigo et al. [33].

Listing 2.17: Example of code in Cilk

```
1  cilk int fib(int n) {
2    if (n < 2) {
3      return n;
4    } else {
5      int x, y;
6      x = spawn fib(n - 1);
7      y = spawn fib(n - 2);
8      sync;
9      return x + y;
10   }
11 }
```

`fib` is a Cilk procedure as indicated by the presence of the `cilk` keyword before the return type. The Fibonnaci algorithm is implemented in its naive form (without memoization or any optimization) and computes the *n-1*th and *n-2*th numbers before summing them. The computation of these numbers is done by `spawn`ing additional procedures. Execution then reaches the `sync` keyword. This creates a barrier that waits for the completion of the children procedures of the current procedure. This is the only possible way of performing a synchronization in Cilk. This is in contrast to futures where programs can only wait for the completion of one procedure.

Work-stealing The key feature of Cilk is its use of *work-stealing* [34]. Work-stealing is a mechanism whereby computing resources are allowed to steal work from other computing resources in order to automatically balance the workload. This is achieved in Cilk by the dequeues mentioned earlier. If a processor runs out of procedures to run in its dequeue, it attempts to steal a procedure from another processor as follows:

- Every time a processor spawns a new procedure, it begins executing this procedure, suspending the current one and placing it at the bottom of its queue;
- Whenever a processor runs out of procedures, it attempts to steal one from another processor;
- Whenever a processor stalls, because a synchronization is necessary, it suspends the current procedure, and attempts to execute the next one available, stealing from

Listing 2.18: Computing Fibonacci in the original release of Cilk

```
thread fib(cont ink k, int n) {
  if (n < 2) {
    send_argument(k, n);
  } else {
    cont int x, y;
    spawn_next sum(k, ?x, ?y);
    spawn fib(x, n - 1);
    spawn fib(y, n - 2);
  }
}

thread sum(cont ink k, int x, int y) {
  send_argument(k, x + y);
}
```

other processors if necessary;

- Whenever a processor resolves a synchronization for a procedure ran by another processor, it steals this procedure and places it at the bottom of its queue.

In [34], Blumofe et al. prove that this scheduling strategy and the implementation of the Cilk work-stealer, is efficient and its performance can be predicted. Work-stealing has since been used in other runtimes, like StarPU (Section 2.2.5) or XKaapi (Section 2.2.8).

The original release of Cilk

This section presents the original release of Cilk [32]. In [33], the authors point out that one of their objectives since the initial release of Cilk had been to make the language more user-friendly. The objective of this section is to provide an example of how languages can evolve in a way that makes them easier to use.

Listing 2.18 presents the Cilk code used to compute the nth Fibonacci number in parallel, in the original release of Cilk. If one compares this code to Listing 2.17, there is a number of differences.

- The `fib` procedure does not have a return type and is instead tagged `thread`.
- There is an additional keyword called `cont` that can be applied to variables, and designates them as *continuation variables*.
- The `return` statements are replaced with the `send_argument` in the terminal case, and completely removed in the general case.
- A new unary operator `?` is introduced and can be used on continuation variables.
- The `sync` keyword does not appear and all assignments have been removed.

These are due to two major changes: absence of return types and use of *continuation variables*.

Return type In the original release of Cilk, Cilk procedures did not have a return type. In fact they could not even contain a `return` statement. Instead, the result of a Cilk procedure had to be communicated through a *continuation variable*.

Continuation variables A continuation variable is akin to a promise. At its creation, it does not hold a value. The `send_argument` function takes a continuation variable and a value as parameters, and resolve the continuation variable with this value, similar to a `set` on a promise. The `?` operator behaves similarly to a `get` operation; however, unlike `get` which can be called at any point, the `?` can only be used when the value of a continuation is required as a parameter to a Cilk procedure (in other words, one cannot write `int a = ?x;`).

Observations The usage of continuation variables allowed Cilk to behave like a dataflow language: procedures could be spawned but not necessarily scheduled immediately if one of their effective parameters was an unresolved continuation variable. The scheduler would keep track of the missing continuation values and only schedule the procedure once all its parameters were known. Such a programming style is quite common in synchronous languages like Lustre [35]. However, as Cilk aims to extend the C language, both these choices made Cilk harder to use than necessary. The lack of return values and the shift to a more dataflow oriented paradigm are both at odds with the existing programming paradigm of C.

Evolution towards Cilk 5 Cilk 5 added return values in Cilk procedures, and replaced continuation variables with barriers. It made the programming model of Cilk closer to the imperative nature of C. On the other hand, it also made synchronization less refined: it went from being able to synchronize on a single value to only being able to synchronize on the completion of every single child task.

Positioning Cilk, like StarPU, is a runtime system targetting efficiency, although with different approaches: in StarPU the program is described as a graph of tasks with explicit dependencies, whereas Cilk schedules procedures as they are created. Both frameworks perform dynamic scheduling, and both approaches use a form of barrier for synchronization. Barriers provide simple and safe synchronization, but sometimes a finer-grain synchronization is more efficient. We believe these frameworks would benefit from advanced synchronization tools with variable granularity of synchronization, like the ones we design in this thesis.

2.2.8 XKaapi: Dataflow Task Programming on Heterogeneous Architectures

XKaapi [36] is both a runtime system and a library that allows dataflow task programming on heterogeneous architectures. An XKaapi task is similar to OpenMP tasks, X10 [37] asynchronous activities, or even Encore asynchronous functions: it is a block of code that

is to be executed at a certain point in time, be it by a thread, by another process, or by another machine.

An XKaapi task can be written for several different computing units, such as a CPU or a GPU. It is possible to provide both a CPU and a GPU implementation of the same task; in this case, the programmer may chose to spawn the task on a specific computing unit or simply let the runtime decide which computing unit is the best through heuristics. This is similar to what happens in StarPU, although both runtimes use different heuristics to determine where to run the task.

XKaapi draws inspiration from Cilk by using a work-stealing scheduler: once a thread becomes idle, it attempts to get work by stealing some from another thread, called the *victim*. The work-stealing scheduler in XKaapi computes dataflow dependencies between tasks in order to know which task to steal. Unlike some other approaches like StarPU or OmpSS [38] that compute dependencies as tasks are created and build a graph of dependent tasks, XKaapi computes dependencies when it attempts to steal a task. This is done in order to reduce the overhead that happens if dependencies are computed at the same time as task creation. This is inspired by the "work-first principle" of Cilk [33], which states that the design of a runtime should minimize overheads during work and transfer them onto the critical path of the application. The *critical path* [39] of a parallel program is the longest chain of dependencies between tasks, which is the main bottleneck of a parallel application.

As a result of this scheduling, non stolen tasks are executed in a FIFO order, with the oldest created task being executed first. This FIFO order is a valid sequential order of execution as was shown in XKaapi's predecessors Athapascan [40] and Kaapi [41].

The library part of XKaapi exposes types that are useful when dealing with arrays, called *ranges*. A range is basically a slice of an array, and this allows the programmer to effectively slice an array into chunks that can be processed in parallel by multiple different tasks, or multiple instances of the same task. The ranges can be tagged with *read*, *write* or *read-write* annotations to help the runtime determine whether a task read from, write to or read-from-and-write-to a specific range, which guides scheduling.

Positioning XKaapi appears as a mix of Cilk and of StarPU: it gives tools for dataflow programming (original Cilk), uses a work-stealing scheduler (all versions of Cilk) and targets heterogeneous architectures (StarPU). The focus on dataflow is interesting as it removes the need for explicit synchronizations using synchronization tools. Instead, the programmer can simply specify which parts of the memory are read / written / read-and-written by a task and the runtime schedules the tasks with respect to these specifications. This is, again, similar to what is possible in StarPU.

We believe that there are improvements that can be made when it comes to tasks that manipulate arrays. The concept of *ranges* is interesting as slicing an array into chunks can result in finer-grained synchronizations, which may improve parallelism compared to a synchronization over the whole array, an idea we explore in Chapter 4 when applied to the combination of arrays and promises. However, the slicing is static and explicitly made by the programmer: a task is created for each slice, and there is no way of knowing

the best slice size. Since we also explore this idea of finding a good slice size in Chapter 4, we believe our tools could be used in combination with XKaapi to further improve performance.

We now move on to two sections that present some background as well some state of the art on two concepts we use in this thesis: futures in Section 2.3 and data streaming in Section 2.4.

2.3 A Study of Futures

In Section 2.1.2 we briefly introduced *futures*. A future is a synchronization tool that can be used to synchronize a task A with the completion of a task B. In addition to performing the synchronization, the future will contain the value returned by task B upon completion. Listing 2.19 shows a toy example in C++.

Listing 2.19: Toy example of a future used to retrieve the value computed by a thread

```
1  int f() {
2    return 2;
3  }
4
5  int main() {
6    future<int> fut = async(f);
7    // ...
8    printf("%d\n", fut.get());
9    return 0;
10 }
```

The `async` keyword[12] is used to launch function `f` *asynchronously*: the function may start running immediately, or at some point in the future. The function may run in another thread, in another process or even on a different machine. In addition to launching the function, `async` produces a future that will hold the result of the execution of `f`. Synchronization on the future is performed by using the `get` method on this future. The call to `get` blocks until a value is available, at which point this value is returned.

Upon creation, a future does not hold any value: it is said to be *unresolved*. It is said to be *resolved* once a value is stored inside it.

Futures are used in the active-objects programming paradigm, a presentation of which can be found in Section 2.2.4. A quick recap is presented here. The active-object paradigm is inspired by the object paradigm: programmers work with single-threaded entities called actors, instantiated from classes equipped with methods and attributes, that communicate with each other through *asynchronous method calls*. An asynchronous method call produces a future that will hold the result of the call.

In this thesis, when dealing with futures we work in the active-object language Encore [8]. From now on, examples using futures are written in this language. The incoming section, Section 2.3.1 presents the Encore language. If the reader is already familiar

[12] In C++ `async` is a function, but we do not make this distinction here

with its syntax and semantics, they may skip to Section 2.3.2 in which we present the limitations of the future construct.

2.3.1 The Encore Programming Language

Listing 2.20 presents the Hello World of Encore.

Listing 2.20: Encore's Hello World

```
1 active class Main
2   def main(): unit
3     println("Hello World!")
4   end
5 end
```

In Encore the `Main` class acts as the entry point in a program. The class is qualified as *active*: instanciating an active class creates an active object. Its `main` method behaves like the `main` method of the main class of a Java application.

In Encore, a future that will be resolved with a value of type `T` is typed `Fut[T]`. Futures may be created through the `async` keyword, or through the `!` operator. `async` asynchronously runs a function call, and produces a future that will hold the result of the execution of this function. The `!` operator is the asynchronous variant of the dot (`.`) operator in object-oriented languages and designates an asynchronous method call.

Futures expose two operations. The first, `get`, retrieves the value inside the future, blocking the calling thread if no value is available. The second, `await`, checks if a value is stored inside the future, and suspends the currently-executing thread if no value is available. `await` implements a form of cooperative scheduling; it can be used for example to synchronize on a future but release the current thread if the future is not resolved.

Listing 2.21 illustrates both the creation and manipulation of futures.

2.3.2 Limitations of Futures

As we have seen before, futures are a good abstraction when it comes to synchronizing one thread with the completion of a task. However, they suffer from limitations, due to how they are typed. Consider Listing 2.22.

In the `main` function, at line 15, an instance of `Foo` is created. Through the `!` operator on line 16, an asynchronous call is made to the method `foo` of this object, sending `12` as its parameter. Since the method has return type `Fut[int]`, the result of the asynchronous call is typed `Fut[Fut[int]]`. This means the programmer is exposed to a future whose resolving value is itself a future. This is called *nesting*. As we can see on line 17, this requires the programmer to perform two calls to `get` in order to extract the integer computed by the `bar` method of the `C` object instantiated by `foo`. Finally, on line 18, a call is made to method `foo_fut` of the `Foo` object previously instanciated, sending a future of integer as parameter.

There are two important observations here: 1) Futures can be *nested*, and such nesting *requires* multiple explicit synchronizations to be performed in order to access a non future

Listing 2.21: Active objects in the Encore language

```
-- Instanciating Foo will create an active object
active class Foo
  def f(): int
    return 2
  end
end

-- Global function
fun factorial(n: int): int
  if n < 1 then
    return 1
  else
    return n * factorial(n - 1)
  end
end

active class Main
  -- Entry point
  def main(): unit
    -- Create an instance of Foo
    -- Store the reference in `foo`
    var foo: Foo = new Foo()
    -- Async call to method `f` through the `!` operator
    var fut: Fut[int] = foo!f()
    println(get(fut))

    -- Asynchronous call to function `factorial`
    -- through the `async` keyword
    var fut2: Fut[int] = async(factorial(10))
    println(get(fut2))
  end
end
```

value, and 2) Non future values cannot be promoted to their future selves: one cannot substitute an integer for a future of integer.

The requirement for multiple explicit synchronizations when dealing with nested futures makes sense in a context where the programmer is interested in whether a task is completed or not; in a context where the programmer cares more about the computation of values, a single explicit synchronization that yields a non future value would be preferable. If the programmer is only interested in the computation of values, not even knowing the number of nested futures involved in the computation would be preferable: the programmer should know the value will be available in the future, but the exact number of synchronizations needed to get this value is irrelevant. Giachino et al. in [42], as well as Henrio and Rochas in [43] highlighted these two different forms

Listing 2.22: Some problems tied to the typing of futures

```
active class Foo
  def foo(x: int): Fut[int]
    val c: C = new C(x)
    return c!bar()
  end

  def foo_fut(x: Fut[int]): Fut[int]
    val c: C = new C(get(x))
    return c!bar()
  end
end

active class Main
  def main(): unit
    val foo: Foo = new Foo()
    val fut: Fut[Fut[int]] = foo!foo(12)
    val result: int = get(get(fut))
    val result2: Fut[Fut[int]] = foo!foo_fut(get(fut))

    -- ...
  end
end
```

of synchronization. In [44], Henrio names the first synchronization pattern *control-flow* synchronization: synchronization is driven by the execution of a `return` statement, by the flow of control of the application. The second synchronization is called both *wait-by-necessity* and *dataflow* synchronization: synchronization is driven by the necessity of *data*, of a non future value. We discuss the pros and cons of these two patterns in Section 2.3.3.

The second observation, the inability to promote non future values to their future selves, originates from a lack of expressivity in the type system of the language. This induces a need to write extra code if a function should work with both future and non future parameters. Additionally, it makes writing functions that *may* return a future value complicated. Consider the following example: a programmer is writing a Broker whose purpose is to dispatch tasks to Workers. Each time a task is submitted to the Broker, the Broker returns a future that will contain the result of the execution of the task. This is shown in Listing 2.23.

Listing 2.23: Excerpt of a task Broker

```
1 active class Broker
2   def submit(task: Task[T]): Fut[T]
3     return select_worker()!submit(task)
4   end
5 end
```

An optimization on Brokers is to have the Broker cache results as they are produced, so submitting the same task a second time may yield its cached result. Listing 2.24 shows an attempt at implementing this pattern.

Listing 2.24: Task Broker using a cache

```
1  active class Broker
2    val cache: Map
3
4    def submit(task: Task[T]): Fut[T]
5      var cached: Maybe[T] = cache.lookup(task)
6      match cached with
7        case Just(value) => return value -- Typed T
8        case _ => return select_worker()!submit(task) -- Typed Fut[T
           ↪ ]
9      end
10   end
11 end
```

We can observe a typing problem. If the value was already computed, then the `submit` method needs to return a value of type `T`. If the value was not computed, then it will be by a Worker, and the method returns a value of type `Fut[T]`. There is an inconsistency in the return type, therefore the method cannot be written this way. A simple solution would be to return the result of an asynchronous call to the identity function. Replace `return value` with `return async(id(value))`. This is less than ideal: it creates an additional asynchronous call, which means a future needs to be created, and an extra synchronization needs to be performed. If a value of type `T` could be promoted to a `Fut[T]`, the problem would solve itself in an elegant way.

Multiple explicit synchronizations can be avoided by using implicit futures, which we present in Section 2.3.3. They can also be avoided using the `forward` construct which we present in Section 2.3.4. Lifting non future values to futures is somewhat mitigated when using implicit futures as well. We will see that both solutions (implicit futures and `forward`) suffer from some drawbacks. In Chapter 3 we build upon both in order to solve these two problems (multiple synchronization and promotion of non future values). Moreover, we achieve better expressivity when using futures, and allow further optimizations of operations on futures.

Let us begin with a presentation of implicit futures and their synchronization.

2.3.3 Typing and Synchronizing Futures - Explicit and Implicit. Control-Flow and Dataflow

The futures we have discussed until now in Section 2.1.2 and Section 2.3 are called *explicit futures*, in opposition to *implicit futures*. The classification of futures between implicit and explicit comes from Flanagan and Felleisen in their work on formalizing futures [45, 46]. Futures are said to be implicit if there is no need for a dedicated operation to access their content, and explicit if such an operation is needed (*i.e.* `get`).

For instance, in the MultiLisp language [7], futures are implicit. Listing 2.25 illustrates this property. As a point of comparison, Listing 2.26 presents the same code in the Encore language that uses explicit futures.

Listing 2.25: Usage of futures in the MultiLisp language

```
1 (def foo
2   (lambda (x y z)
3       (+ (future
4              (bar y z))
5          x)
6   )
7 )
```

Listing 2.26: Listing 2.25 rewritten in Encore

```
1 fun foo(x: float,
2         y: float,
3         z: float): float
4   val f: Fut[float] = async(
      ↪ bar(y, z))
5   return get(f) + x
6 end
```

Both listings define a function called `foo` that takes three parameters. `foo` computes the sum of the first parameter `x` with the result of asynchronously calling a `bar` function and passing it the remaining two parameters, `y` and `z`.

In MultiLisp, futures are created using the `future` construct. This constructs takes an expression e as parameter, launches its evaluation in parallel to the current execution context and returns a future that will hold a result of the evaluation of the expression e. In the MultiLisp version, the `+` function is able to seamlessly use the future as if it was an integer. Internally, this induces an implicit synchronization on the future in order to get its value (this mechanism is called *wait-by-necessity*). This is in contrast to the Encore version in which a call to `get` is necessary to extract the value from the future before performing the addition. [13]

As we can see from the MultiLisp example, both problems encountered with explicit futures are resolved with implicit futures. The lack of a syntactic distinction between a future and a non future value means the same piece code can work seamlessly with future and non future values. Additionally, although the example does not highlight it, if the future passed as parameter to the `+` function is nested, the runtime *automatically* performs as many synchronizations as needed in order to get a non future value: synchronization is driven by the availability of data rather than the end of a computation. The synchronization is said to be *dataflow*.

Although implicit futures solve both problems, one can wonder whether an implicit

[13] Implicit futures are not exclusive to programming languages without a static type system. See the ProActive library [47] where a value with static type `T` may be a handle to a future of `T` or value of type `T`.

synchronization is inherently preferable when compared to an explicit one. Consider Listing 2.27 in MultiLisp.

Listing 2.27: Implicit future synchronization inducing a deadlock in MultiLisp

```
1 (define f1 #f) ; Define a global called f1 initialized to false
2 (define cycle
3   (lambda (x)
4     (set! f1 (future (+ x f1))) ; f1 refers to the resulting
        ↪ future
5   )
6 )
7
8 (cycle 12)
9 (display f1)
```

The `cycle` function evaluates the expression `(+ x f1)` in parallel. The parallel evaluation is initiated through the `future` function, that returns a future, stored in `f1`, that will be resolved with the result of the evaluation of `(+ x f1)`. This piece of code deadlocks because future `f1` is resolved with itself on Line 4. While this problem is by no means exclusive to implicit futures, it would be much easier to debug when the synchronization is explicit. Here, the synchronization occurs during the execution of the `+` function. However, since we are in MultiLisp, future values are indistinguishable from non future values. As a consequence, the interface of the `+` function, here simplified as $number \rightarrow number \rightarrow number$, does not indicate that calling this function may induce a synchronization. This gives an argument in favor of explicit futures.

Summary There are several features one may want when working with futures: explicit synchronization for easy debugging, no nesting at the type level for code simplicity, lifting of non future values to future values for code reusability, dataflow synchronization for code simplicity, control-flow synchronization for control-flow awareness. However, these features are split between explicit and implicit futures and there are incompatibilities between these two ways of typing. Implicit futures cannot have an explicit synchronization by definition: an implicit future cannot be distinguished from a non future value. Furthermore implicit futures cannot have a control-flow synchronization, again by definition. All of this points towards using explicit futures, but as we have seen they lack many desirable features. Throughout litterature, explicit futures have been consistently paired with a control-flow synchronization which requires nesting in order to properly type the synchronization operation. However, in recent years there have been many efforts to bring features from the implicit world into explicit futures:

- In Section 2.3.4 we present the `forward` construct, proposed by Fernandez-Reyes et al. This construct offers a "delegation" operation on explicit futures which mimics a dataflow synchronization with limitations.
- In [44], Henrio proposed a type system and operational semantics for explicit futures with a dataflow synchronization.

- These two works formed the basis of the Godot system [1], which we present in Section 2.3.5. The Godot system proposes an implementation of the dataflow explicit futures of Henrio.

We now present the `forward` construct that adds a form of dataflow synchronization on explicit futures.

2.3.4 The `forward` Construct - Delegating Resolution

Let us consider the problem of the task broker we presented above in Listing 2.23. Listing 2.28 presents an extended version of the original excerpt with an effective call to the broker in order to dispatch some work.

Listing 2.28: Broker in Encore, active objects edition

```
1  active class Worker
2    def work(task: Task[T]): T
3      -- ...
4    end
5  end
6
7  active class Broker
8    def dispatch(task: Task[T]) : Fut[T]
9      return select_worker()!work(task)
10   end
11 end
12
13 active class Main
14   def main(): unit
15     var broker: Broker = new Broker()
16     var result: Fut[Fut[int]] = broker!dispatch(
17       new Task(fun(x: int, y: int) return x + y end, 12, 13)
18     )
19     println(get(get(result)))
20   end
21 end
```

As we can see, there is a nesting of futures that occurs on Line 16. This is due to the fact that the `Broker.dispatch` method has a return type of `Fut[T]`. As such, asynchronously calling it produces a `Fut[Fut[T]]`.

One may wonder if `dispatch` needs to be typed `Fut[T]`, which is the source of nesting. There are options to change this typing, although they suffer from drawbacks:

- Perform a `get` in `dispatch` to produce a value of type `T`. This prevents parallelism because now `dispatch` must wait until the task has been executed.
- Use the `await` statement to yield the currently-executing thread until the task has been executed. This is elegant, but it may lead to a lot of memory being consumed

as more and more futures may be awaited. Moreover, the language used may not offer this operation.

- Use a promise to store the result of `work`, which completely removes nesting and allows parallelism. This is an interesting solution, although it suffers from the drawbacks we presented in Section 2.1.2, namely a lack of guarantee of resolution of the promise.

In [48], Fernandez-Reyes et al. present the `forward` construct. Listing 2.29 presents the (shortened) resulting code when using `forward` in the Broker.

Listing 2.29: Asynchronous Broker in Encore using the `forward` construct

```
1 active class Broker
2   def dispatch(task: Task[T]): T
3     var fut: Fut[T] = select_worker()!work(task)
4     forward(fut)
5   end
6 end
```

The `forward` construct works by delegating the resolution of the *current future* to another task. The *current future* is a concept in the execution model of the Encore language: every time an asynchronous call is made, be it through the `async` keyword or the `!` operator, the runtime creates a task. To this task is associated a *current future*, which is the future the task will resolve upon completion of its execution.

The `forward` construct takes a single parameter: a future. `forward(f)` ties the resolution of the current future with the resolution of `f`. In addition, it terminates the calling task in a similar way to a return statement, without returning a value. The task produces no value, and its future remains in an unresolved state until future `f` gets resolved, at which point the task's future gets resolved with the same value. Listing 2.30 shows a toy use of `forward`.

We can now look back at Listing 2.29. Function `dispatch` is now typed `T`, so an asynchronous call is simply typed `Fut[T]`: no nesting appears. This version does not limit parallelism (unlike the version in which `dispatch` immediately performs a `get` on the resulting future), and memory is not saturated if multiple calls to `work` run in parallel (unlike the version in which `dispatch` immediately `await`s the resulting future). Finally, since this version still uses futures rather than promises, there is a guarantee of resolution.

Optimizing `forward`

There exists a compile-time optimization used to reduce the amount of futures created at runtime. Consider Listing 2.31.

Listing 2.30: `forward` example

```
fun f(): int
  return 12
end

fun g(): int
  var res: Fut[int] = async(f)
  -- Once res is resolved (here with 12), the current future of
  -- this call to g will be resolved with 12 as well.
  forward(res)
end

active class Main
  def main(): unit
    var res: Fut[int] = async(g)
    println(get(res)) -- 12
  end
end
```

Listing 2.31: Optimization of the `forward` statement in Encore

```
active class Broker
  def dispatch(task: Task[T]): T
    forward(select_worker()!work(task))
  end
end
```

The single difference with Listing 2.29 is that the future produced by the asynchronous call to `work` is no longer stored in an intermediate variable. Instead, the entire asynchronous call is given as parameter to `forward`. In this case, the compiler changes code generation. Rather than chaining the resolution of the current future with the resolution of the future given to `forward`, the runtime outright replaces the current future of the call to `dispatch` with the future created by the asynchronous method call. This effectively removes one creation of a future at runtime and removes a chaining.

In short, `forward` seems to be a good solution to the lack of dataflow synchronization on explicit futures: it can be used to remove nesting, which leads to synchronizations that yield non future values. However, it does not solve all problems: non future values still cannot be promoted to future values, which means code that work indifferently on future and non futures values is still not possible. It also does not solve the problem of a function that may return a future or non future value depending on the context. Programmers must also explicitly write `forward` every time they need to perform this “collapsing” of type to prevent nesting: collapsing would be more interesting at the type system level, as it would be automatic.

2.3.5 The Godot System — Dataflow Explicit Futures

In [1], Fernandez-Reyes et al. build upon the work of Henrio in [44], as well as the `forward` construct by Fernandez-Reyes et al. in [48] in order to create *dataflow explicit futures*. Dataflow explicit futures, as their name suggests, are explicit futures that offer a dataflow synchronization.

[1] observes three problems on traditional futures, be they explicit or implicit.

- The Type Proliferation Problem, which refers to the nesting of futures that appears as the type system level. It also encompasses the lack of promotion from non future values to their future selves. This is exclusive to explicit futures.
- The Future Proliferation Problem, which refers to the long chains of futures that may need to be followed in order to reach a value. This problem appears on both implicit and explicit futures. As we have seen before in Section 2.3.4, the `forward` construct can help mitigate this problem when using explicit futures.
- The Fulfillment Observation Problem, which refers to the fact that control-flow and dataflow futures do not observe the same events during synchronization. A control-flow future observes the completion of a task, a dataflow future observes the availability of a non future value.

The dataflow explicit futures proposed by the Godot system can be used to solve all three problems.

Godot introduces a new type, called `Flow`, which denotes a dataflow explicit future. `Flow[T]` therefore is a dataflow explicit future. Intuitively, it corresponds to a chain of futures of an arbitrary length, eventually holding a value of type `T`. Synchronizing on such a future produces a value of type `T`.

The Godot system proposes a type system that offers a *collapsing* rule as well as a *lifting* rule. Collapsing, denoted by the ↓ operator, prevents the appearance of nested `Flow`s. Informally, `Flow[Flow[T]]` is recursively collapsed into `Flow[T]`. For instance, `Flow[Flow[int]]` collapses into `Flow[int]`, and `Array[Flow[Flow[int]]]` collapses into `Array[Flow[int]]`. Lifting, denoted by the subtyping rule `T` <: `Flow[T]` allows any non-flow value of type `T` to be promoted to its dataflow explicit future self, `Flow[T]`.

The syntactic and semantic parts of the Godot system introduce operations on dataflow explicit futures that mirror the ones existing on classical control-flow explicit futures:

- The `get*` keyword performs a dataflow synchronization on a dataflow explicit future: if a value is available it returns this value, otherwise it blocks the calling thread until a value is available, at which point that value is returned. Importantly, this means that the result of `get*` will always be a non-flow value. In other words, `get*` ≡ `get(get(get(...)))`.
- The `await*` keyword checks if a dataflow explicit future has been resolved with a non-flow value. If it has been, execution continues. Otherwise, `await*` suspends the currently-executing thread.

- The `forward*` operator is the dataflow equivalent of the `forward` operator. The authors of [1] assert that forwarding a dataflow explicit future is equivalent to returning it, however no proof of this equivalence is provided.

Finally, Godot exposes how to encode control-flow futures from dataflow futures by using a boxing operator that stops collapsing. It also exposes how to encode dataflow futures from control-flow ones.

One can see that all three problems mentioned earlier are solved:

- The collapsing and lifting rules together solve the Type Proliferation Problem: collapsing prevent nested futures from appearing at the type system level, and lifting allows non future values to be lifted to their future selves.

- The Future Proliferation Problem is solved by the usage of `forward` and `forward*`. In the case of dataflow explicit futures, since `return` is (stated to be) equivalent to `forward*`, there is no longer a need for the programmer to write `forward*`: the more natural `return` is enough.

- The Fulfillment Observation Problem is solved by the possibility to encode control-flow futures from dataflow ones and vice-versa. Language designers can now offer both forms of synchronization on explicit futures, and programmers can freely choose the kind of synchronization they need.

Finally, an artifact associated with [1] implements dataflow futures as a Scala library on top of Scala's existing control-flow futures by following the rules established in the paper.

Positioning Godot is the culmination of several works that study the relationship between the explicit / implicit nature of futures and their synchronization. The dataflow explicit futures of Godot are interesting when it comes to ease-of-use, as they allow greater expressivity than traditional explicit futures. However, there are some aspects of the formalism that could be improved. Notably, the calculus proposed in Godot does not allow side-effects. Many languages that offer futures allow side-effects, and a calculus that allows them could be used to better model programs in such languages. The authors also claim that, with dataflow explicit futures, using `forward*` is observably equivalent to using `return`, although no formal proof is provided. This claim is interesting as it could allow interesting optimizations, for instance by having a compiler decide when it is best to use one or the other. However, the first step would be to prove this statement. The artifact in Scala also suffers from a limitation: the implementation of the collapsing rule prevents the appearance of types like `Flow[Flow[T]]`, however the limitations of the language did not allow the authors to prevent nesting within parametric types. Listing 2.32 illustrates the problem.

Listing 2.32: Limitation of the Godot artifact in Scala

```
class Foo[T]:
  private var fut: Flow[T] = Nil
```

If the generic class `Foo` is instanciated with `T = Flow[T']`, then the type of the attribute `fut` will be `Flow[Flow[T']]`: the collapsing will not occur. Finally, the question of efficiency is not touched upon, and the paper lacks benchmarks.

In Chapter 3 we improve on all these points: we propose a full implementation of dataflow explicit futures in the Encore language with complete support for collapsing in parametric types; we offer a calculus that allows side-effects; we propose a proof of the equivalence between `forward*` and `return` on dataflow explicit futures; and we produce benchmarks that present the performance of dataflow explicit futures compared to the native control-flow explicit ones of Encore, as well as control-flow futures encoded from our dataflow ones.

2.4 Data Streaming

Introduction In [49, 50], data streaming is defined as "a programming model in which a set of *modules* working in parallel communicate with each other through *channels*". In practice, one can see this approach as generating a directed graph of objects who are allowed to communicate only with their neighbors. Modules are classified as either *sources*, if they receive data from an external source; *filters* if they process data they receive from another module and send it to another module; and *sink* if they receive data from a module and send data to an external consumer. An important aspect of stream processing is that the amount of data that circulates through the graph may be (and generally is) infinite.

An early example of stream processing would be Kahn's Process Networks (KPN) [51], in which multiple deterministic processes communicate with each other through first-in first-out channels. Reading from a channel is blocking, *i.e.* if no data is available then the process waits until data arrives, and writing is non-blocking. An interesting property of KPNs is that their behavior is entirely deterministic, and is not influenced by either communication delays or when a computation occurs.

Nowadays stream processing is ubiquitous in programming, used in video or audio processing, rendering, compression and so forth. The theoretical model of KPNs is now used as the inspiration for modern streaming tools, which may or may not keep all the original properties of KPNs. The very nature of stream processing lends itself quite well to parallelism: all the modules in the system can safely run in parallel, as communication between modules is done through channels, which can avoid data races[14]. This also makes stream processing a programming model in which communications can create a bottleneck if they are not optimized. A example of bad communication would be having filters repeatedly request a single unit of work, rather than waiting until multiple units

[14] Assuming the same data does not appear in multiple modules at once, and assuming channels are safe

of work are available before performing a single synchronization. We will explore this in more details in later chapters, mainly Chapter 4.

Stream processing can be done in one of two flavors: push-based or pull-based. In *pull-based* stream processing, a module requests data and waits until this request completes without doing anything else. In *push-based* stream processing, a module registers a continuation to be called when data is available, and can keep working while awaiting data. The pull-based approach offers a linear control-flow that is easy to follow, at the cost of forcing some threads to wait when there is nothing to do which induces costly context switches. The push-based approach induces a non linear control-flow, which may be harder to debug if an error occurs, but reduces the amount of context switches which may improve performance.

In this section we present four different approaches to streaming: an extension of OpenMP called OpenStream, a dedicated language StreamIt, a framework called FastFlow and a future-based approach in ABS. We position ourselves on a surface level in this section. Chapter 5 will present a more detailed positioning.

2.4.1 A Dataflow Streaming Extension for OpenMP: OpenStream

In Section 2.2.3 we presented the OpenMP framework. In particular we presented the concept of OpenMP tasks. As a reminder, a task is a section of code inside an OpenMP parallel region whose execution is assigned to one of the threads inside the region. OpenStream [52, 53] is an extension of OpenMP that adds streaming capabilities to OpenMP tasks.

In OpenStream, the OpenMP `task` directive can receive up to two new additional clauses, named `input` and `output`. Each of these clauses takes a list of parameters, known as *streams*. Intuitively, streams passed as parameters to the `input` clause denote streams from which the task will read data, and streams passed as parameters to the `output` clause denote streams to which the task will write data.

A stream is a C object (scalar, array or structure). When using a stream in an `input` or `output` clause, the programmer may connect a window to the stream. A window is a simple C array of the same type as the stream and of a given size. Connecting a window to a stream allows the retrieval (from `input` streams) or insertion (in `output` streams) of multiple data at once. If no window is connected then elements are consumed/produced one at a time. There are two concepts tied to the window: the horizon and the burst.

The *horizon* of a window is its size. It is not possible for a task to read data beyond the horizon of the window connected to a stream. The *burst* is a parameter that can be specified when connecting a window to a stream in an `output` or `input` clause, and defines how many elements will be produced (in output streams) or consumed (in input streams).

OpenStream also offers some support for broadcast operations, that is multiple tasks that all read the same value from the same stream (*i.e.* the data is not consumed until all readers have read the data) through the special `peek` and `tick` clauses that replace the `input` clause. `peek` is equivalent to reading with a burst of 0, *i.e.* not advancing

the stream; `tick` is equivalent to a task that reads with a burst of 1 and immediately discards the value.

OpenStream relies on a transformation pass, in order to turn OpenMP pragmas as well as streams themselves into valid C code. An OpenStream stream is not presented to the programmer as an object (*i.e.* there is no `Stream` class), but rather as a scalar, structure or array annotated with a directive `__attribute__((stream))`. Insertion of data into a stream is performed either through direct assignment (`x = 12` with `x` a stream), or through assignment into a window connected to a stream (`win[0] = 12, win[1] = 13` with `win` an array of at least two elements connected to a stream). Insertion cannot be implicit: it always occurs through an assignment. Conversely, removal of data from a stream is performed either through direct assignment (`result = x`, with `x` a stream), or through a connected window, (`i1 = win[0], i2 = win[1]` with `win` an array of at least two elements connected to a stream). In a similar way to insertion, removal cannot be implicit and always occurs through an assignment.

In order to support dynamic graphs of tasks, with tasks writing to or reading from multiple streams at the same time, OpenStream allows the use of variadic windows and streams, where whole arrays can be specified as streams and as windows, with their size not known at compile-time.

Example We provide a small example of OpenStream in Listing 2.33, alongside a few comments to explain it.

Positioning OpenStream extends the OpenMP specification, and therefore an OpenStream program must be compiled with a dedicated compiler. Our first contribution, dataflow explicit futures in Chapter 3 also requires support in a compiler, whereas our second and third contributions, PromisePlus and FIFOPlus, in Chapter 4 are built as libraries. The pros and cons presented in 2.2.1 apply.

OpenStream, as its name implies, is focused on adding streaming capabilities to the OpenMP framework, whereas PromisePlus and FIFOPlus are suited to be used in multiple different environments. In fact, PromisePlus was used inside an OpenMP program in order to add some form of streaming without having to rely on annotations, and used in an OpenMP-free environment as well.

Streams in OpenStream are first-class objects whose interface is not explicitly exposed. The OpenStream runtime abstracts away a lot of the underlying operations: synchronization and task activation are not a concern of the programmer. This is a consequence of OpenStream building itself on top of OpenMP and extending its syntax: OpenMP pragmas are subjected to a transformation pass during compilation and the same applies to the modified pragmas in OpenStream. Using pragmas hides most of the complexity of the program, and allows the programmer to focus on expressing the flow of data between tasks through streams. On the flip side it can make OpenStream programs non-intuitive to newcomers. For instance, `x = 2` usually stores the value 2 inside the variable `x`, and reading from `x` immediately afterwards would yield 2. In OpenStream, if `x` is a stream, `x = 2` pushes 2 inside the stream, and reading from the same stream afterwards will retrieve

Listing 2.33: MWE of OpenStream code

```
int x __attribute__((stream));
int win_out[3]; // Output window, 3 elements horizon
int win_in[2]; // Input window, 2 elements horizon

#pragma omp parallel
{
  #pragma omp single
  {
    while (...) {
      // Output three elements at a time (burst of three)
      #pragma omp task output(x << win_out[3])
      {
        // Do stuff
        win_out[0] = ...;
        win_out[1] = ...;
        win_out[2] = ...;
      }

      // Read one element at a time (one element burst)
      #pragma omp task input(x >> win_in[1])
      {
        int a = win_in[0]; // Get first item
        // ...
      }
    }
  }
}
```

the next available value. Programmers that are used to OpenStream would probably prefer `x = 2` instead of a more explicit `stream_push(x, 2)` as it is shorter. We believe that abstraction is a good thing, but one should remain mindful of the least surprise principle when creating abstractions.

Finally, OpenStream lets programmers choose the values of the window and the burst, whereas our final tool, FifoPlus, attempts to automate the deduction of a good granularity of data transfer.

2.4.2 StreamIt: A Language for Streaming Applications

StreamIt [10] is a programming language, as well as a compiler [54], both designed specifically to allow efficient stream-oriented programming. The language features skeletons that allow programmers to express the structure of a stream.

Presentation We first begin with an in-depth presentation of StreamIt, with a focus on the technical aspects, before performing a high-level comparison with our own con-

tributions. We first present the execution context of StreamIt, and then present the technical parts. A StreamIt program is usually executed on multiple different threads, and also on multiple different machines (also called *nodes*) at the same time.

In terms of syntax, the StreamIt language is similar to the Java language, and is equipped with classes. The central class in StreamIt is called `Filter`. A `Filter` is a class that contains a method called `work`, which represents a transformation operated inside a pipeline of transformations. Other classes in the StreamIt library define composition skeletons to assemble processes that can run in parallel.

A `Filter` has an *input* and an *output*. Specifically, it can have at most one of each. In order to split the work into subtasks the programmer would have to use one of the *filter composition* tools offered by StreamIt: the `Pipeline` which corresponds to a *linear* sequence of filters: there are no branches, you simply have a structure akin to a singly linked list.

The `SplitJoin` which corresponds to the *split-join* pattern (sometimes called the *fork-join* pattern), where one `Filter` may send data to two or more other different filters, and at some point all branches join at a new common filter.

The last composition filter is the `FeedbackLoop`, which represents a backtrack in the sequence of filters. The communication between the different filters is done through the use of FIFOs.

A `Filter` must define its data rates, *i.e.* how much elements it will push from its input FIFO or pop to its output FIFO during an execution of its `work` method. This is done in the `init` method. These data rates are *constant* in the sense that they are not allowed to change at runtime.

Finally, StreamIt is also a compiler, for programs written in the StreamIt language. This compiler can perform aggressive optimization. Some of these optimizations involve the partitioning of filters across machines, or fusing or splitting filters. Fusing filters can be used to improve cache efficiency, by removing some transfers to and from the FIFOs between filters. Splitting filters can be used to improve parallelism by redistributing the filters across available nodes.

Listing 2.34 presents a typical StreamIt program. This program creates a producer-consumer, in which the producer pushes elements by groups of 10, and the consumer retrieves elements by groups of 10.

Positioning The abstractions we propose in Chapter 4 can be used in a streaming context, they are both designed as libraries that can be integrated into existing languages. As discussed earlier the approach of designing a library rather than a programming language has different consequences. In particular StreamIt is able to perform much more optimization than we do, while our solutions are easier to integrate into existing applications and to use in conjunction with other constructs for parallel programming.

A final point of interest in StreamIt is the initial attempt at implementing a concept of “reconfiguration” into their streams. This is tied to a concept of granularity of synchronization in the streams: when a producer or consumer registers on a stream, it must specify how much data it will add or remove at once, and this rate is not allowed

Listing 2.34: Typical StreamIt program

```
class Producer extends Filter {
  int n;

  void init() {
    n = 0;
    setOutput(Integer.TYPE);
    setPush(10);
  }

  void work() {
    output.push(n);
    ++n;
  }
}

class Consumer extends Filter {
  void init() {
    setInput(Integer.TYPE);
    setPop(10); setPeek(10);
  }

  void work() {
    int data = input.pop();
    // Do something with data
  }
}

class Main extends Pipeline {
  void init() {
    add(new Producer());
    add(new Consumer());
  }
}
```

to change as time goes on. “Reconfiguring” a stream means allowing these rates to change as time goes on. One may want to change this rate for instance in compression algorithms where the flow of data is not constant in time. Despite the creators’ initial wish to add this concept to StreamIt, it was never implemented. In contrast, one of our abstractions, FifoPlus in Section 4.2, integrates this concept of reconfigurable rate of data production/consumption in a stream.

We will provide a detailed positioning relatively to StreamIt in Section 5.3.

2.4.3 FastFlow: High-Level and Efficient Streaming on Multi-Cores

FastFlow [55] is a C++ library that aims to provide programmers with high-level abstractions that allow the creation of streaming applications. FastFlow has *nodes* that are the programming entities performing the computations (similar to *filters* in StreamIt).

Like SkePU, FastFlow exposes skeletons modeling various kind of parallel programming computations: the pipeline, the farm, and the loop. The *pipeline* is a sequence of successive transformations performed on a stream of data. A *farm* is a set of workers all connected to a single emitter and a single receiver, with the emitter creating a stream of independent data that is distributed among the workers. Workers then process this data and transfer it to the receiver. A *loop* models a cycle in a streaming graph, where one node A sends data to a node B that appeared before on the path to A.

The communication between nodes in FastFlow is done through wait-free Single Producer Single Consumer Queues [56, 57]. More complex communication patterns, such as Multiple Producers Single Consumer Queues, or Single Producer Multiple Consumer Queues are implemented using these wait-free SPSC Queues alongside arbiter threads. This ensures efficiency in the communication. These efficient queues are also exposed to the programmer if what they want to realize is not feasible with the existing FastFlow skeletons.

FastFlow builds its efficiency on its avoidance of memory fences. The SPSC queues do not use any atomic operations (so no fences) and are wait-free by design; when combined with arbiter threads, it allows for the creation of MPSC / SPMC / MPMC queues that do not use fences either and remain efficient. This is achieved by performing a copy of the data when it is inserted into a queue: a copy is cheaper than a fence.

Listing 2.35 presents the pipeline pattern in FastFlow.

An `ff_node` is the equivalent of a StreamIt `Filter`, it is a stage in a pipeline. It exposes an `svc` method, which performs work on an input. The `ff_send_out` method is used to send data to the next stage of the pipeline. The use of skeletons allows programmers to focus entirely on writing the stages of the pipeline without focusing on communication, and the runtime automates everything: the `svc` method of a stage of the pipeline is automatically invoked whenever an input is ready.

Positioning When it comes to ease-of-use and efficiency, FastFlow can rightfully claim to be extremely good at what it does. The pipeline skeleton abstracts away everything a programmer may have to do in order to properly spawn the threads and connect them through FIFOs. However, FastFlow also comes with a rather natural drawback from libraries that aim to automate everything: it is complex to do something that escapes the scope of the framework. For instance, FastFlow's pipelines do not come with replication, unlike StreamIt's. If a programmer wants to have replication in a pipeline, they either use the `Farm` skeleton in conjunction with the `Pipeline` skeleton, or they outright write their own `PipelineWithReplication` skeleton. Another example is the impossibility of creating a branching pipeline using only the native skeletons of FastFlow, despite some algorithms using such a pipeline (for instance the `dedup` algorithm in the PARSEC Benchmark Suite).

Listing 2.35: The Pipeline pattern in FastFlow

```
class InitStage : public ff_node {
public:
  void* svc(void*) {
    for (int i = 0; i < 10; ++i) {
      ff_send_out(i);
    }
  }
};

class ProcessStage : public ff_node {
public:
  void* svc(void* data) {
    int value = (int)data;
    ff_send_out(value * 2);
  }
};

class Adder : public ff_node {
  int sum = 0;
public:
  void* svc(void* data) {
    sum += (int)data;
  }

  int total() const { return sum; }
};

int main() {
  ff_pipeline pipeline;
  pipeline.add_stage(new InitStage());
  pipeline.add_stage(new ProcessStage());
  pipeline.add_state(new Adder());
  pipeline.run_and_wait_end();
  return 0;
}
```

2.4.4 Futures for Streaming Data in ABS

The Abstract Behavioral Specification language [19, 58] (ABS) is an active-object language (Section 2.2.4). Recall that in these kinds of languages, objects are called actors, single-threaded entities that communicate through asynchronous method calls. An asynchronous method call produces a future that will hold the result of said call. Futures can be handled through the standard `get` statement that blocks the calling object until the future is resolved, or through the cooperative statement `await` that will release control of the object if the future is not yet resolved.

In [59, 60], the possibility of streaming was added to ABS. The idea is to use a mechanism similar to futures. A streaming method with a return type `T` can be annotated with the `stream` keyword in order to have it produce a stream of data, each data being of type `T`. In such a method, the programmer may use the `yield` keyword in order to add a value to the stream, similar to how one may produce a value from the execution of a coroutine [61] in a functional language. Unlike coroutines however, execution of a `yield` statement does not suspend the execution of the task. Streams in ABS are push-based rather than pull-based.

Calling a streaming method produces a value of type `Stream<T>` with `T` denoting the return type of the method. Programmers may then use either the `get-finished` or `await-finished` statements to access the content of the stream. `get-finished` extracts a value from the stream by waiting until one is available. `await-finished` peeks inside the stream, releases control of the current object if no value is available and resumes once one is, or continues execution immediately if a value is available. This is similar to the `get` and `await` keywords when working with futures.

Streams in ABS may be destructive or non-destructive, with the difference being decided at runtime, rather than statically. The destructiveness of a stream is one of its properties, and is decided at stream creation. The difference between a destructive and a non-destructive stream lies in how the internal cursor that indicates which value is the next in the sequence moves after a read. In a destructive stream, all reader threads share the same read cursor on the stream, and reading a value will advance this shared cursor. In a non-destructive stream, every reader thread has its own cursor on the stream, ensuring all threads have access to the entirety of the stream. Destructive streams may suffer from race conditions if the order in which threads read the elements is not the one that was expected, whereas non-destructive streams avoid races. On the other hand, the memory management of non-destructive streams is much more complicated, since ABS uses garbage collection, and streams cannot be freed even if no reader threads are manipulating them since a new reader thread may appear at any time. Destructive streams on the other hand are freed once they have been terminated and all data has been read.

Termination of a stream is automatically done once the streaming method performs a `return` statement, similar to how a `return` statement resolves the (possibly) associated future of the method. This is manifested as a special token added to the stream that denotes end-of-file.

Listing 2.36 provides an example of an ABS program with streams. A producer (`Producer`) streams all values from 0 to 99 and a consumer (`Consumer`) reads them. Recall that `!` denotes an asynchronous function call, and the result must be acquired through a call to `get`.

Positioning Streaming in ABS draws inspiration from futures, and as such is similar to one of our tools, PromisePlus, that itself draws inspiration from promises (recall that promises are manually handled futures). A consequence of the similarity with futures is that ABS streaming provides the same guarantees and properties as futures: assuming a

Listing 2.36: Example of streaming in ABS

```
interface IProducer {
  Unit run();
  Stream<int> request();
}

class Consumer {
  void consume(IProducer source) {
    Stream<int> s = get(source!request());
    bool last = false;
    while (!last) {
      // value stores the next value in the stream once
      // await finished completes
      int value = s.await finished {
        // Block of code executed only upon
        // getting last value from stream.
        last = true;
      };
      // ...
    }
  }
}

class Producer implements IProducer {
  Stream<int> stream;

  // Should be called first to initialize the stream
  Unit run() {
    if (stream == null) {
      stream = this!start();
    }
  }

  Stream<int> request() {
    return stream;
  }

  int stream start() {
    int i = 0;
    while (i < 100) {
      yield i; // Add value to stream
      i = i + 1;
    }
    return -1; // Terminate stream
  }
}
```

well-formed program, all streams will eventually receive a completion token, avoiding infinite wait; this can be enforced by making sure all paths in a streaming method contain a return statement[15]. This is a stronger guarantee than what we get with PromisePlus.

The use of `yield` gives the streaming method more flexibility than what is possible with futures, where resolution is tied to a return statement. Multiple `yield` statements

[15]Whether these returns can be reached is undecidable, however we are assuming the program is well-formed so infinite loops are excluded

may exist in the same method at different points in the control-flow graph of the method, similar to what can be achieved when using promises rather than futures. Unlike promises however, `yield` statements cannot escape a streaming method: the ABS compiler makes sure `yield` statements only appear in a function tagged `stream`. Promises, are allowed to be passed around, which gives more flexibility to the programmer, but makes synchronization less clear.

Streaming in ABS does not have support for configurable or reconfigurable granularity: data is available as soon as it is produced, and synchronization is performed on a per-item basis rather than on chunks. This is in contrast to our tools that give the programmer control over the granularity of synchronization, or attempt to automatically find one that is suitable for the situation.

2.5 Miscellaneous Tools

In this section we present synchronization tools that are not frameworks, nor libraries, and that are not specifically geared towards streaming, but still allow programmers to write safe code.

2.5.1 Distributed Futures

Distributed futures [62] were proposed by Leca with the objective of allowing good interactions between task parallelism and data parallelism. Task parallelism involves multiple tasks running in parallel. These tasks can be functionally different, they may work on completely unrelated data and perform unrelated computations. Data parallelism allows parallelism by distributing data to multiple processes that perform a computation on a part of the data. The actor model introduced before is a good example of task parallelism, and the parallelization of some embarrassingly parallel problems, like the functional *map* operation is a good example of data parallelism.

Task parallelism and data parallelism do not encounter the same problems: task parallelism usually involves synchronization between tasks, and data parallelism involves the communication of data. Task parallelism is quite flexible when it comes to task synchronization and data communication: we have seen many tools in the previous sections that are used in the context of task parallelism. Data parallelism on the other hand is more restricted: synchronizations and data exchanges are not as prevalent as they are in task parallelism. As a consequence there are fewer tools specifically geared towards data parallelism.

Distributed futures target High Performance Computing, where task parallelism and data parallelism are usually used together and very large amount of data are processed. In such a context, futures become limited as synchronization and communication tools. Consider the following context: a matrix of several million elements is distributed between multiple processes in order for some computation to be performed in parallel over it. Using a future to synchronize on the completion of this distributed computation suffers from two major drawbacks:

1. A future holds the result of a computation, and the entire result must be available at once. In this context, the result of the computation is split between multiple processes, so there would be a need for multiple futures.
2. Because futures must hold the result of a computation, the total memory footprint of all the futures needed to synchronize on the end of the computation would be that of the initial matrix itself. In a distributed context, communication bandwidth can become a bottleneck, so transferring the entire matrix multiple times would be disastrous.

Distributed futures are similar to futures in that they allow programmers to synchronize on the completion of a task and retrieve the result of said task. Unlike traditional futures that hold the actual result of a task, distributed futures hold descriptions of fragments of the result of a distributed computation. For instance, a *map* operation on a million elements array could be distributed to N processes, each one of them performing a part of the computation. Process I would work on indices $\frac{1000000}{N} * I - 1$ to $\frac{1000000}{N} * I$. Once process I is done, it would write a 4-tuple in the distributed future: its PID, a unique identifier for this specific computation, the offset in the global array in bytes ($I - 1 * \frac{1000000}{N} *$ size of data type) and the number of bytes produced ($\frac{1000000}{N}$). The synchronization completes once all processes have written a description of their result in the distributed future.

This not only allows a single future to work as the synchronization entity over multiple processes that perform a distributed computation, but it also reduces the communication cost when synchronizing over the future. Unlike traditional futures, synchronization does not produce a usable value. Rather, it produces the information needed to get a usable value.

Positioning Distributed futures are used as a synchronization point on data that has been sliced and distributed across multiple processes. In this regard, they go in the same direction as us. However, distributed futures do not allow partial synchronization: accessing a distributed future's content blocks until the entirety of the data is available, regardless of the availability of a slice. While they do have benefits, mainly in terms of use of the communication bandwidth, distributed futures do not offer better control over the granularity of synchronization.

2.5.2 Message Sets

Message sets are a programming abstraction proposed by Frolund and Agha in [63]. The purpose of this abstraction is to make the coordination of distributed objects easier. Frolund and Agha describe distributed objects as "reactive": they emit requests that get answered, and they react according to the answer. In the context of object-oriented programming, a request, or message send, would correspond to a method call, and the reaction would be the execution of said method. This is even more explicit in the active-objects paradigm, where the objects react to messages sent to them in an asynchronous way.

The problem observed by Frolund and Agha is as follows: the request-response model of the object paradigm is not suited to the construction of distributed systems. In distributed systems it is not uncommon to have a many requests-many responses scheme, for instance if a process needs to wait until multiple other processes have completed a computation. In the object paradigm, such a scheme would be implemented using multiple intermediate variables, each of them indicating if a given process has completed the computation. This complicates the implementation of objects, as programmers need to consider two orthogonal concepts: when to react and how to react. Message sets are an abstraction that encapsulates both these concepts in a declarative way.

Message sets are composed of three concepts: activated commands, activators and receptionists. A receptionist is a first-class entity that contains a mailbox. As first-class entities, receptionists may be passed around through function parameters or return values. Programmers can put messages in the mailbox of a receptionist at will, but there is no dedicated operation to consult the messages or delete them. These operations are performed by the activators. Depositing a message in a mailbox is asynchronous and non-blocking. To give an example, `recep<int> int_recep;` declares a receptionist whose mailbox will only contain integers. `int_recep(12)` asynchronously deposits the message 12 in this receptionist.

An activator watches one or more receptionists for the arrival of messages. Unlike receptionists that are first-class entities in the code, activators are a purely syntactic construct, and get translated into runtime objects. An activator defines *when* to react to a message or a set of messages. *When* is extremely flexible: it can be reacting to any message sent to a given receptionist, or to a specific message sent to a specific receptionist, or to a specific set of messages each sent to a specific receptionist etc. For instance, considering we still have our previously declared receptionist of integers `int_recep`, the expression `int_recep? 12` defines an activator that will react to the message 12 being sent to the receptionist `int_recep`. The expression `int_recep? x` defines an activator that will react to any message sent to the receptionist `int_recap` and will bind the value of the message sent to the name `x`. Activators can be combined, using the `&&` and `||` operators, the combination of which yields an activator that will behave as the logical combination of all the underlying activators.[16] `int_recep? 12 && bool_recep? true` is an activator that gets triggered once the `int_recep` receptionist has received the message 12 and (logical and) once the `boot_recep` receptionist has received the message `true`. Therefore, an activator can be seen as a receptionist combined with a boolean guard (see Hoare's input guards in CSP [16] and Dijkstra's guarded commands [64]).

Finally, activated commands define *how* to react to the activation of one or more activators. An activated command is of the form `activator` → `command`. This translates to the following: once `activator` gets triggered, execute the command (code) specified by `command`. For instance, `int_recep? x -> printf("%d\n", x)` is an activated command that will print the value received by the `int_recep` receptionist upon receiving any message.

[16] In the original paper by Froslund and Agha, the `&&` operator was called `and`, and the `||` operator was called `or`

We will not discuss the questions of fairness between activators when a message can activate multiple at once, or when multiple messages can activate the same activator, as this goes beyond the scope of this chapter.

Message sets effectively decouple the aspects of how and when to react to a set of messages. Unlike the traditional request-response model of the object paradigm, they allow messages to be emitted at any point of code, similarly to promises. However, there are some differences with promises, the most prominent one being the inability to access the content of the mailbox of a receptionist directly. Unlike promises which are still wrapped in the request-response model (synchronization on a promise is resolved once it gets a value, a single value), activators can be combined to react to the arrival of a set of messages. An alternative form of synchronization similar to activators can be found in the ParT [20] framework in the Encore language, where futures are given monadic combinators that allow to react to the resolution of all the futures in a set, or to the resolution of *any* future in a set. This is of course constrained by the lack of flexibility of futures when it comes to the point of resolution.

Positioning Message sets make synchronization explicit, in the same way as promises. Message sets are more expressive than promises, in that they allow to easily express complex synchronization patterns that depend on the availability of a combination of data. This data can be specific (*e.g.*, a hardcoded value like 12) or more general (any value of a given type). Furthermore, message sets encapsulate the operations performed on the values provided by the synchronization, something which promises do not allow.

There is also a difference in the way the synchronization is resolved. With a promise, a `get` unblocks as soon as the promise is resolved with a value, and this resolution occurs exactly once. When using promises, programmers write code that looks sequential. With message sets, the use of conjunctions, disjunctions and the ability for a receptionist to receive multiple messages makes synchronization more difficult to follow. The flow of execution becomes dependent on a number of factors.

2.5.3 Joins

Join patterns were proposed by Fournet and Gonthier in [65]. A join pattern is made of two components: a body and a guard. The guard is made of a set of events that must all be triggered in order for the body to be executed. Consider Listing 2.37, taken verbatim from Haller and Cutsem paper "Implementing Joins using Extensible Pattern Matching" [66].

Listing 2.37: Join Patterns from [66]

```
1 public class Buffer {
2   public async Put(int x);
3   public int Get() & Put(int x) { return x; }
4 }
```

This code presents a traditional FIFO, written using joins in the Cω language (in this language, joins are called “chords”). Unlike most traditional implementations of such a FIFO, there is neither a dedicated data structure to store the data, nor a body for the `Put` operation. In this code, `Put` and `Get` are both methods of the `Buffer` class, and message sending operations, similar to the actor model. Like in the actor model, an instance of the class keeps a list of the messages it has received. Unlike in the actor model, a received message may not lead to a processing of the message. In this example, receiving a `Put` message does not do anything, as the method does not have a body. Instead, the `Buffer` merely stores the message.

The syntax `Get() & Put(int x)` defines a join pattern: in order to process a `Get` message, a `Put` message must have been received before, and not have been processed. If a `Get` message is received after a `Put` message, then the body of `Get` is executed and the `Put` message is consumed, its value bound to `x`. A join pattern may contain any number of messages in order to express different synchronization patterns. In [67], Turon and Russo present how to use join patterns to implement reader-writer locks, semaphores and barriers among others.

The key feature of join patterns is their ability to express the different pieces of code involved in a synchronization in a declarative way, and which data dependencies are involved. In the previous example, it is made evident by the signature of `Get` that the synchronization is resolved by a call to `Put`. This can be contrasted against any implementation of an unbounded FIFO queue that uses atomics or mutex and condition variable to protect a data structure manipulated by both `Put` and `Get`: the programmer knows there is a synchronization because synchronization tools are used, but they do not know what the synchronization is used for.

In [66], Haller and Cutsem further refine the declarative aspect of join patterns by improving on Turon and Russo previous work [67]. In particular, they add a pattern-matching based approach to reacting on events, which allows to statically declare that a reaction to a join pattern should be triggered if a given message is received multiple times and not consumed. They give the example of a reader writer lock, where multiple readers may have read access on a shared data as long as there is not a writer active, but only a single writer can have write access on it at the same time.

Positioning Join patterns offer a declarative way of expressing synchronization, which makes data dependencies and code explicit compared to approaches like promises. The synchronization offered by promises is extremely flexible, as it can happen anywhere, which also makes the purpose of the synchronization more obscure. When it comes to granularity of synchronization, the join patterns augmented with pattern matching of Haller and Cutsem are a step, although they suffer from a massive drawback: a join

pattern is a static construct. While expressing a synchronization on a statically known number of received messages would be easy with joins, expressing a dynamically chosen number of received messages would require extra code that is no longer part of the join pattern itself but part of the body of the pattern, which is one of the criticism Haller and Cutsem leveled at the original join patterns.

Overall, join patterns share many similarities with message sets. In turn, their strengths and weaknesses when compared to promises are the same as with message sets: they are strictly more expressive, but they make synchronization more difficult to follow.

2.6 Summary

We have presented several languages, libraries, paradigms and tools that can be used by programmer to write parallel applications that are efficient and safe. We now present our different contributions. We begin with our work on the dataflow explicit futures of Godot, with a formalization in a stateful calculus, a complete implementation in the Encore language and a proof of the equivalence between `return` and `forward*`. We next present PromisePlus, a library tool based on promises that is focused on efficient communication of arrays between threads through a concept of configurable granularity of synchronization, while also being easy-to-use and safe. Finally, we present FifoPlus, a library tool that can be used to perform efficient streaming between threads, with a concept of automatic configuration of granularity of synchronization through an analytical model.

Chapter 3

Dataflow Explicit Futures

In the previous chapter, we presented the programming abstraction known as "futures". A future is an entity that represents the result of an ongoing computation. It is used when launching a sub-task in parallel with the current task to later retrieve the result. A future initially does not hold a value and is said to be *unresolved*. A future that holds a value, a result of a parallel sub-task, is said to be *resolved* ("fulfilled" is also used).

Historically, futures were categorized as either explicit or implicit, depending on whether there are dedicated operations to access a future's content (explicit) or not (implicit). [44] and [1] demonstrated that the way synchronization is performed is a more distinctive feature. Synchronization can be either control-flow or dataflow; this difference is observed when synchronizing on a chain of futures, that is a future resolved with another future. Control-flow synchronization is driven by the flow of execution: synchronizing on a future resolved with a future will yield a future. Dataflow synchronization is driven by the availability of data: synchronizing on a future resolved with a future will follow the entire chain of futures and yield a non future value. There is a link between synchronization, explicitness and typing: control-flow futures are traditionally explicit and use parametric types, such as `Fut[T]` for a future resolved with a value of type `T`. This is the case in mainstream languages such as C++ or Java. Implicit futures are dataflow by nature; [44] showed that explicit futures can be dataflow as well.

[1] proposed a formalization of dataflow explicit futures, using [44] as a foundation. It also provided a type system as well as runtime semantics for these futures. There are two key features in the type system of [1]: a *collapse* operation that prevents the appearance of nested futures at the type system level; and an introspection operation that can tell whether a future is resolved with another future or with a non future value.

In this chapter we build upon [1] and investigate the use of dataflow explicit futures compared to usual ones. Our contributions are as follows:

- In Section 3.1 we propose a new core calculus, called `DeF`, that is dedicated to the study of dataflow explicit futures. This calculus is not meant to be a new programming language, but rather a minimalist formal language, expressive enough to be representative of existing mainstream languages. Because we target mainstream languages with imperative aspects, we define this calculus with a mutable state

and some standard imperative constructs. It also features asynchronous calls that take advantage of dataflow explicit futures. This provides a main improvement over Godot: their calculus was based on λ-calculus, and as such did not support a mutable state.

- In Section 3.2 we formally and experimentally study one of the most promising optimization enabled by dataflow explicit futures: according to [1], return or forwarding a dataflow explicit future is equivalent, although the authors did not provide a proof. We prove this equivalence and investigate the benefits that it can bring. We formalize also the `forward*` primitive of the dataflow explicit futures in our calculus. Furthermore, we provide a novel typing rule for a function that performs a `forward*`; this typing rule remains safe even when the function that performs a `forward*` is invoked synchronously, something that was lacking from Godot.
- Finally, in Section 3.3 we present both a concrete implementation of dataflow explicit futures in the Encore language. In particular, we discuss implementation choices regarding introspection and the typing of the `forward*` statement in the context of a programming language. We illustrate the expressiveness and ease of programming brought by dataflow explicit futures, with both illustrative examples and by showing, in Section 3.3.4, that control-flow futures can be encoded from dataflow explicit futures. As such we provide the first complete implementation of an encoding of the Godot approach.

From now on, we use the word *flow* to designate a dataflow explicit future.

3.1 The `DeF` and `DeF+F` languages

This section presents two core languages called `DeF` (for data-flow explicit futures) and `DeF+F` that extends `DeF` with a `forward*` operator. `DeF` features data-flow explicit futures exclusively, functions that can be called synchronously or asynchronously, and a global state that enables imperative programming. We designed `DeF` as a minimalistic calculus but expressive enough so that our results on data-flow futures would still be relevant on a wide range of more complex calculi.

Our languages are equipped with a type system. Compared to our implementation of data-flow explicit futures in Encore we do not encode objects or actors and consequently data-races exist in `DeF` but not in Encore.

3.1.1 Syntax of `DeF`

We use the following notations in our syntax. A bar over an expression, e.g. $\overline{q}$ denotes a list. All lists are ordered, except the set of futures in a configuration. $\varnothing$ is the empty list, and $q\#\overline{q}$ is the ordered list $\overline{q}$ with q prepended to it. As for sets of futures, FF' simply denotes the union of the sets F and F'. $\oplus$ denotes any usual integer or boolean binary

Table 3.1: Static syntax of `DeF`.

P	$::=$	$\overline{T\ x}\ \ \overline{M}\ \{\overline{T\ x}\ s\}$	program
M	$::=$	$T\ \mathtt{m}(\overline{T\ x})\ \{\overline{T\ x}\ s\}$	function
s	$::=$	$\mathtt{skip}\ \mid\ x = z\ \mid\ \mathtt{if}\ v\ \{\,s\,\}\ \mathtt{else}\ \{\,s\,\}\ \mid\ s\ ;\ s\ \mid\ \mathtt{return}\ v$	statements
z	$::=$	$e\ \mid\ \mathtt{m}(\overline{v})\ \mid\ !\mathtt{m}(\overline{v})\ \mid\ \mathtt{get}{*}\ v$	right-hand-side of assignments
e	$::=$	$v\ \mid\ v \oplus v$	expressions
v	$::=$	$x\ \mid\ \textit{integer-and-boolean-values}$	atoms
B	$::=$	$\mathtt{Int}\ \mid\ \mathtt{Bool}$	basic type
T	$::=$	$B\ \mid\ \mathtt{Flow}[B]$	Type

Table 3.2: Runtime Syntax of `DeF`.

cn	$::=$	$a\ \rangle\ \overline{F}$	configuration
F	$::=$	$\overline{f(\overline{q})}\ \overline{f(w)}$	set of futures in configuration (unresolved / resolved)
q	$::=$	$\{\ell \mid s\}$	stack frame
w	$::=$	$f\ \mid\ b$	runtime values: future identifiers and basic values
b	$::=$	$\textit{integer-and-boolean-values}$	values of basic types
ℓ, a	$::=$	$[\overline{x} \mapsto \overline{w}]$	local and global store
s	$::=$	$\mathtt{skip}\ \mid\ x = z\ \mid\ \mathtt{if}\ v\ \{\,s\,\}\ \mathtt{else}\ \{\,s\,\}\ \mid\ s\ ;\ s\ \mid\ \mathtt{return}\ v$	statements
v	$::=$	$x\ \mid\ w$	variable or runtime value
e	$::=$	$v\ \mid\ v \oplus v$	expressions with runtime values
z	$::=$	$e\ \mid\ \mathtt{m}(\overline{v})\ \mid\ !\mathtt{m}(\overline{v})\ \mid\ \mathtt{get}{*}\ v$	right hand side of assignments

operator. Table 3.1 shows the static syntax of `DeF`. A program P is made of a list of typed global variable declarations, a list of function definitions, and a main function (s is the body of the main function). Each function M has a return type, a name, a list of typed arguments, a list of typed local variables, and a statement that is the function body. Asynchronous function calls are supported via the $!\mathtt{m}(\overline{v})$ syntax. If B is a basic type, $\mathtt{Flow}[B]$ denotes the type of a data-flow explicit future that is to be resolved by a value of type B.

Table 3.2 describes the runtime syntax of `DeF`. The configuration of a running `DeF` program contains a global store a, a set of resolved futures $f(w)$ and a set of unresolved futures $f(\overline{q})$ each associated with a running call stack $\overline{q}$. Each frame q of a call stack contains a local store ℓ and a statement to be executed s. Each store is defined as a mapping from variable names to runtime values where runtime values w are basic values b and future identifiers f. Note that we also allow expressions and future values to contain future identifiers, this is useful for evaluating `get*` statements.

To evaluate a program $P = \overline{T\ x}\ \ \overline{M}\ \{\overline{T'\ x'}\ s\}$ one must place it in an *initial configuration.* We first suppose that all variables have an initialization value denoted[1] 0. The initial configuration for P is: $a\ \rangle\ f_0(\{\ell|s\})$ where f_0 is any future identifier, the global store $a = [\overline{x} \mapsto \overline{0}]$ maps all global variables to an initialization value, and $\ell = [\overline{x'} \mapsto \overline{0}]$

[1]Defining initial values for each existing type is not detailed here, note that 0 is a valid value for a `Flow[int]`, corresponding to an already-resolved future.

maps all local variables of the main body to an initialization value.

The sequence s ; s is associative and `skip` is neutral as the statement has no effect; thus we can rewrite any statement s under the form s' ; s'' where s' is not a sequence (and s'' might be `skip` if s is a single statement). In the following we suppose that every statement is rewritten under this form (this simplifies the operational semantics).

Configurations are identified modulo reordering of futures (hence $F f$ picks any f in the set, not necessarily the last one) and future identifiers are unique; if $f(w) \in cn$ and $f(w') \in cn$, necessarily $w = w'$. Thus a configuration can be considered as a mapping from future identifiers to call stacks or values.

3.1.2 Semantics of DeF

Figure 3.1 details the small-step operational semantics of DeF that uses three notations:

- Similarly to other languages with binding of methods or functions, see e.g. [19], we rely on a bind operator instantiates a new stack frame with the local environment and the body of the function to be executed. Suppose the program that is evaluated defines a function $T\ \mathtt{m}(\overline{T\ x})\ \{\overline{T\ y}\ s\}$, we have: $\text{bind}(m, \overline{w}) = \{[\overline{x} \mapsto \overline{w}, \overline{y} \mapsto \overline{0}] \mid s\}$
- Given two stores a and ℓ, $(a + \ell)$ is the union of the two stores with values taken in ℓ in case of conflict: $(a + \ell)(x) = \ell(x)$ if $x \in \text{dom}(\ell)$ and $(a + \ell)(x) = a(x)$ otherwise[2].
- Given two stores a and ℓ, $(a + \ell)[x \mapsto w]$ is defined as $(a, \ell[x \mapsto w])$ if $x \in \text{dom}(\ell)$, or $(a[x \mapsto w], \ell)$ otherwise.

Our semantics features asynchronous calls. Invk-Async spawns an asynchronous task by adding an unresolved future to the configuration. From this point, the callee executes the spawned task in parallel with the caller. Once the spawned task is completed, the callee fulfils the future through the rule Return-Async. The semantic rules for synchronous calls are more standard. Invk-Sync pushes a new stack frame initialized in accordance with the function called and the arguments provided, and Return-Sync pops the current stack frame and resumes the execution of the caller with the return value properly propagated.

The `get*` operator retrieves the value of a future f, defining the synchronization points of a DeF program. Indeed, rules Get-Future and Get-Data are only enabled when getting a future of the form $f(w)$, that is, a fulfilled future. Consequently, performing a `get*` on an unresolved future blocks the process trying to access the future.

Once the relevant future is fulfilled, repeated applications of Get-Future will follow a sequence of futures and, unless there is a loop of futures or a deadlock, Get-Data will finally provide the result of the `get*` operation.

Concretely, if there is a sequence of futures $f_0(f_1) \ldots f_{n-1}(f_n)\ f_n(w)$ in the configuration cn (with w not a future), a statement y = `get*` f_0 will become a y = `get*` f_1 statement thanks to the Get-Future semantic rule, then y = `get*` f_2, and so on, until yielding a y = `get*` w statement. At this point, the Get-Data rule can be applied, reducing the statement to $y = w$. This resolution takes place at every `get*` statement: another `get*` f_0 will lead to the same series of Get-Future applications. In this example

[2]We suppose that the program is type-checked and every variable is declared.

$$\frac{}{[\![w]\!]_\ell \;=\; w} \qquad \frac{x \in \mathrm{dom}(\ell)}{[\![x]\!]_\ell \;=\; \ell(x)} \qquad \frac{[\![v]\!]_\ell = k \qquad [\![v']\!]_\ell = k'}{[\![v \oplus v']\!]_\ell \;=\; k \oplus k'}$$

$$\textsc{Skip}\quad \frac{}{a \rangle F\ f(\{\ell \mid \mathtt{skip}\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid s\}\#\overline{q})}$$

$$\textsc{Assign}\quad \frac{[\![e]\!]_{a+\ell} = w \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{a \rangle F\ f(\{\ell \mid x = e\ ;\ s\}\#\overline{q}) \rightarrow a' \rangle F\ f(\{\ell' \mid s\}\#\overline{q})}$$

$$\textsc{Invk-Async}\quad \frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \mathrm{bind}(m, \overline{w}) = q' \qquad f' \ \text{fresh}}{a \rangle F\ f(\{\ell \mid x = !\mathtt{m}(\overline{v})\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid x = f'\ ;\ s\}\#\overline{q})\ f'(q')}$$

$$\textsc{Invk-Sync}\quad \frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \mathrm{bind}(m, \overline{w}) = q'}{a \rangle F\ f(\{\ell \mid x = \mathtt{m}(\overline{v})\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(q'\#\{\ell \mid x = \mathtt{m}(\overline{v})\ ;\ s\}\#\overline{q})}$$

$$\textsc{Return-Async}\quad \frac{[\![v]\!]_{a+\ell} = w}{a \rangle F\ f(\{\ell \mid \mathtt{return}\ v\ ;\ s\}) \rightarrow a \rangle F\ f(w)}$$

$$\textsc{Return-Sync}\quad \frac{[\![v]\!]_{a+\ell'} = w}{a \rangle F\ f(\{\ell' \mid \mathtt{return}\ v\ ;\ s\}\#\{\ell \mid x = \mathtt{m}(\overline{v})\ ;\ s'\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid x = w\ ;\ s'\}\#\overline{q})}$$

$$\textsc{Get-Future}\quad \frac{[\![v]\!]_{a+\ell} = f'}{a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ v\ ;\ s\}\#\overline{q})\ f'(w') \rightarrow a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ w'\ ;\ s\}\#\overline{q})\ f'(w')}$$

$$\textsc{Get-Data}\quad \frac{[\![v]\!]_{a+\ell} = b}{a \rangle F\ f(\{\ell \mid y = \mathtt{get*}\ v\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid y = b\ ;\ s\}\#\overline{q})}$$

$$\textsc{If-True}\quad \frac{[\![v]\!]_{a+\ell} = \mathtt{true}}{a \rangle F\ f(\{\ell \mid \mathtt{if}\ v\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\}\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid s_1\ ;\ s\}\#\overline{q})}$$

$$\textsc{If-False}\quad \frac{[\![v]\!]_{a+\ell} \neq \mathtt{true}}{a \rangle F\ f(\{\ell \mid \mathtt{if}\ v\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\}\ ;\ s\}\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid s_2\ ;\ s\}\#\overline{q})}$$

Figure 3.1: Semantics of `DeF`. The set of futures F in a configuration is not ordered.

`DeF` futures differ from control-flow explicit futures, as for the latter getting from f_0 to w would have needed $n+1$ explicit `get` statements. When n cannot be defined statically such a sequence is not expressible in the case of control-flow explicit futures. A sequence of $n+1$ futures is the result of $n+1$ levels of asynchronous delegations. Control-flow explicit futures follow the *control-flow* of the program contrarily to data-flow ones.

Example We show here a few examples of application of the semantics inspired by the Encore program in Listing 3.1. The following step-by-step reduction illustrates the progression of the runtime configurations, starting with the asynchronous invocation of `foo` on Line 17 to the resolution of the `get*` on Line 18.

Listing 3.1: Encore with DeF (our extension)

```
active class B
  def bar(t: int): int
    return t * 2
  end

  def foo(a: Flow[int]): Flow[int]
    val t = get*(a) + 1
    val beta = new B()
    return beta!!bar(t)
  end
end

active class Main
  def main(): unit
    val alpha = new B()
    -- Asynchronous call using !!, returns a flow
    val x: Flow[int] = alpha!!foo(1) -- this lifts 1 from int to
      ↪ Flow[int]
    var y: int = get*(x) + 3
    println(y) -- 5
  end
end
```

$$
\begin{array}{l}
\varnothing \rangle\, f_0(\{\varnothing \mid x = !foo(1)\ ;\ ...\}) \\
\rightarrow (\textsc{Invk-Async})\ \varnothing \rangle\, f_0(\{\varnothing \mid x = f\, ;\, tmp = \mathtt{get*}\, f\ ;\, ...\})\ f(\{[a \mapsto 1] \mid tmp = \mathtt{get*}\, a\ ;\, ...\}) \\
\rightarrow (\textsc{Get-Data})\ \varnothing \rangle\, f_0(\{unchanged\})\ f(\{[a \mapsto 1] \mid tmp = 1\ ;\ t = tmp + 1\ ;\ ...\}) \\
\rightarrow (\textsc{Bin-Op})\ \varnothing \rangle\, f_0(\{unchanged\})\ f(\{[a \mapsto 1, tmp \mapsto 1]\ ;\ t = 2\ ;\ tmp2 = !bar(t)\ ;\ ...\}) \\
\rightarrow (\textsc{Invk-Async})\ \varnothing \rangle\, f_0(\{unchanged\})\ f(\{[a \mapsto 1, tmp \mapsto 2]\ ;\ tmp2 = f'\ ;\ \mathtt{return}\ tmp2\}) \\
\qquad f'(\{[t \mapsto 2]\ ;\ tmp = t * 2\ ;\ ...\}) \\
\rightarrow (\textsc{Bin-Op})\ \varnothing \rangle\, f_0(\{unchanged\})\ f(\{unchanged\})\ f'(\{[t \mapsto 2]\ ;\ tmp' = 4\ ;\ \mathtt{return}\ tmp'\}) \\
\rightarrow (\textsc{Return-Async})\ \varnothing \rangle\, f_0(\{unchanged\})\ f(\{[a \mapsto 1, tmp \mapsto 1, tmp2 \mapsto f']\ ;\ \mathtt{return}\ f'\})\ f'(4) \\
\rightarrow (\textsc{Return-Async})\ \varnothing \rangle\, f_0(\{[x \mapsto f]\ ;\ tmp = \mathtt{get*}\ f\ ;\ ...\})\ f(f')\ f'(4) \\
\rightarrow (\textsc{Get-Future})\ \varnothing \rangle\, f_0(\{[x \mapsto f]\ ;\ tmp = \mathtt{get*}\ f'\ ;\ ...\})\ f(f')\ f'(4) \\
\rightarrow (\textsc{Get-Future})\ \varnothing \rangle\, f_0(\{[x \mapsto f]\ ;\ tmp = \mathtt{get*}\ 4\ ;\ ...\})\ f(f')\ f'(4) \\
\rightarrow (\textsc{Get-Data})\ \varnothing \rangle\, f_0(\{[x \mapsto f]\ ;\ tmp = 4\ ;\ ...\})\ f(f')\ f'(4)
\end{array}
$$

3.1.3 Syntax and Semantics of DeF+F

We now extend the language with a `forward*` statement. `forward*` takes a `Flow` as parameter and can be used instead of `return` to terminate the execution of a function. `forward* f` delegates the computation performed by the current task to the task that is

computing `f` but only if `forward* f` is in the body of a function called asynchronously (rule FORWARD-ASYNC). If the function is called synchronously, `forward*` behaves like `return` (rule FORWARD-SYNC)[3]. This behavior in the synchronous case is a design decision: in the case of a synchronous call, there is no future whose resolution needs to be delegated. As a result, we chose to make `forward*` explicitly behave like `return` in the synchronous case. Listing 3.2 illustrates the difference between the synchronous and asynchronous use of `forward*`.

Listing 3.2: Difference between synchronous and asynchronous `forward*`

```
1  active class Forwarder
2    def forwarder(x: int): int
3      return x * 2
4    end
5  end
6
7  active class Example
8    def f(x: int): Flow[int]
9      forward*(new Forwarder()!!forwarder(x))
10   end
11 end
12
13 active class Main
14   def main(): unit
15     var ex = new Example()
16     var e1: Flow[int] = ex!!f(12)
17     var e2: Flow[int] = ex.f(12))
18   end
19 end
```

For the sake of simplicity, we will assume it is possible to perform a synchronous call on an active object as seen on Line 17.

This code creates an instance of the `Example` class and calls its method `f` twice: first in an asynchronous way on Line 16, and then in a synchronous way on Line 17.

Method `f` of class `Example` creates an instance of the `Forwarder` class, and applies the `forward*` operator to an asynchronous call to the method `forwarder` of this class.

The asynchronous call The call `ex!!f(12)` asynchronously launches the computation of `f(12)`. It creates a flow, we call it f_1, that will be resolved with the result of `f(12)`. When the execution of `f` reaches the `forward*` statement, two things happen: 1) A new flow, f_2, is created to be resolved with the value of `forwarder(x)`; 2) Flow f_1 is chained to f_2: when f_2 is resolved with a value v, f_1 is resolved with that same value v.

[3]Please note that this is not the entirety the equivalence between `forward*` and `return` we mentioned in the introduction to this chapter. The equivalence also holds in the asynchronous case and will be proven in Section 3.2

FORWARD-ASYNC
$$\frac{[\![v]\!]_{a+\ell} = f'}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ;\ s\}) \rightarrow a \rangle F\ f(\texttt{chain}\ f')}$$

FORWARD-SYNC
$$\frac{[\![v]\!]_{a+\ell} = w}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ;\ s\}\#q\#\overline{q}) \rightarrow a \rangle F\ f(\{\ell \mid \texttt{return}\ w\ ;\ s\}\#q\#\overline{q})}$$

FORWARD-DATA
$$\frac{[\![v]\!]_{a+\ell} = b}{a \rangle F\ f(\{\ell \mid \texttt{forward}*\ v\ ;\ s\}) \rightarrow a \rangle F\ f(b)}$$

CHAIN-UPDATE
$$\frac{}{a \rangle F\ f(\texttt{chain}\ f')\ f'(w) \rightarrow a \rangle F\ f(w)\ f'(w)}$$

Figure 3.2: Additional rules for the semantics of `DeF+F`.

The synchronous call The call `e.f(12)` synchronously calls `f(12)`. It does not create a flow. When the execution of `f` reaches the `forward*` statement, it asynchronously launches `forwarder(x)` on a new instance of the `Forwarder` class. This creates a new flow f_1, that is immediately returned. This ends the call to `f(12)`.

We present the consequences of the addition of `forward*` on syntax and semantics. The static syntax of `DeF+F` is the same as `DeF` plus a `forward*` statement:

$$s ::= \texttt{skip} \mid x = z \mid \texttt{if}\ v\ \{\,s\,\}\ \texttt{else}\ \{\,s\,\} \mid s\ ;\ s \mid \texttt{return}\ v \mid \texttt{forward}*\ v$$

The runtime configurations of `DeF+F` have one more kind of future: `chain`ed futures.

$$F ::= \overline{f(\overline{q})}\ \overline{f(w)}\ \overline{f(\texttt{chain}\ f')}$$

$f(\texttt{chain}\ f')$ expresses the idea that once f' is resolved with a value w, f gets resolved with w as well.

Figure 3.2 defines the four semantic rules associated with `forward*`. FORWARD-SYNC is similar to RETURN-SYNC and allows using `forward*` with the same semantics as `return` in synchronous calls context. FORWARD-DATA is similar to RETURN-ASYNC but limited to the trivial case of a `forward*` of a non-future value.

The rule FORWARD-ASYNC complements them by handling the forwarding of future values in an asynchronous context. In contrast to the behavior of RETURN-ASYNC that would have inserted an $f(f')$ into the context, FORWARD-ASYNC instead inserts a $f(\texttt{chain}\ f')$. Then, an application of CHAIN-UPDATE will replace the chained future with a resolved future.

Concretely, if there is a sequence of futures $f_0(\texttt{chain}\ f_1) \ldots f_{n-1}(\texttt{chain}\ f_n)\ f_n(w)$ in the configuration *cn*, CHAIN-UPDATE replaces $f_{n-1}(\texttt{chain}\ f_n)$ with $f_{n-1}(w)$, then $f_{n-2}(\texttt{chain}\ f_{n-1})$ with $f_{n-2}(w)$, and so on[4]. The n-th CHAIN-UPDATE updates f_0 to

[4]This semantic rule act as a kind of magical solution to the problem of observing resolution of chains. In a real world runtime system, the observation of chain resolution would preferably be done when a `get*` or `return` is performed.

Listing 3.3: Listing 3.1 modified to use `forward*`

```
1  active class B
2    def bar(t: int): int
3      return t * 2
4    end
5
6    def foo(x: Flow[int]): Flow[int]
7      val t = get*(x) + 1
8      val beta = new B()
9      forward*(beta!!bar(t))
10   end
11 end
```

$f_0(w)$. At this point, assuming w is not a future, a `get` f_0 statement only needs a single Get-Future to reach a Get-Data transition.

The `forward*` construct proposes similar behaviour to `return` in `DeF`, but with operations occurring in a different order: in the case of `forward*`ed futures, future values are propagated as soon as possible, from the inner future to the outer one, and the resolution is performed only once per future, no matter how many delegated invocations there are. If `return` is used instead, several successive future retrieval operations occur until the inner future is reached.

Example We show here how a configuration evolves differently when `forward*` is used instead of `return`. For that purpose, as in Section 3.1.2 we consider a configuration resulting from the execution of the Encore program in Listing 3.1, slightly modified so that `foo` uses a `forward*` to return its result. This modification is presented in Listing 3.3.

When the execution reaches line 9, if we are in a similar state as the example in the previous section we can apply the Forward-Async rule:

$$\varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ x\})\ f(\{[x \mapsto 1, y \mapsto f'] \mid \texttt{forward}*\ y\})\ f'(4)$$
$$\rightarrow \varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ x\})\ f(\texttt{chain}\ f')\ f'(4)$$

Since f' is a resolved future, Chain-Update can be applied immediately:

$$\varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ x\})\ f(\texttt{chain}\ f')\ f'(4)$$
$$\rightarrow \varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ x\})\ f(4)\ f'(4)$$

Then, the $y = \texttt{get}*\ x$ statement can be reduced, fetching the future value with only one Get-Future and one Get-Data.

$$\varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ x\})\ f(4)\ f'(4)\ \rightarrow\ \varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = \texttt{get}*\ 4\})\ f(4)\ f'(4)$$
$$\rightarrow\ \varnothing \rangle\ f_0(\{[x \mapsto f] \mid y = 4\})\ f(4)\ f'(4)$$

Somehow, the Chain-Update transition replaces one of the Get-Future applications of the example in Section 3.1.2. Section 3.2 will show that despite this difference, `return` and `forward*` have in fact equivalent semantics in `DeF+F`.

3.1.4 The Type Systems of DeF and DeF+F

Figure 3.3 gives a type system for `DeF`, and Figure 3.4 introduces an additional typing rule for `DeF+F`. There are four kinds of type judgements: $\Gamma \vdash z : T$ types any expression z; $\Gamma \vdash_{\mathtt{m}} s$ types a statement s that belongs to the function `m`; $\Gamma \vdash T''\ \mathtt{m}\ (\overline{T\ x})\{\overline{T'\ x'}\ \ s\}$ checks that a function definition is well-typed; and $\Gamma \vdash \overline{T\ x}\ \overline{M}\ \{\overline{T'\ x'}\ \ s\}$ checks the correct typing of a program. Due to the fact that a flow already encodes an arbitrary number of asynchronous delegations of a computation, flows of flows are forbidden at the type system level. The notation $\downarrow$ represents the collapsing of a flow type: $\downarrow\mathtt{Flow}[\mathtt{Flow}[T]]$ reduces to $\downarrow\mathtt{Flow}[T]$. The interested reader should refer to [1] for a complete specification of collapse in a type system that supports parametric types. In our simplified context, the description can be summarised by the following rules:

$$\downarrow B = B \qquad \downarrow\mathtt{Flow}[\mathtt{Flow}[T]] = \downarrow\mathtt{Flow}[T] \qquad \downarrow\mathtt{Flow}[T] = \mathtt{Flow}[\downarrow T] \text{ if } T \neq \mathtt{Flow}[T']$$

In Figure 3.3, rule T-Subtype allows any basic type to be considered as a flow type. T-Invk-Async states that the type resulting from an asynchronous function invocation is a flow containing the type returned by the function; the collapse operator prevents nested flow types. Symmetrically, T-Get states that the result of the synchronization on a flow is necessarily a basic type. Indeed as flows on flows do not exist and basic types can be lifted to flow types, a `get*` operation can always be typed and always returns a basic type. Other type-checking rules are standard.

Concerning the rule T-Forward shown in Figure 3.4, it types the `forward*` statement. It checks that the function's return type is $\mathtt{Flow}[T']$ and that the returned type is compatible with this signature. This ensures that a synchronous call to a function that performs a `forward*` must consider the result of type $\mathtt{Flow}[T]$. Because of the subtyping rule, e could be of basic type B but T' cannot be of the form $\mathtt{Flow}[T'']$. Our solution contrasts with what was adopted in Encore, where synchronously calling a function that contains a `forward` performs an implicit synchronization on the future returned by the function; see Section 3.3.3 for a discussion of the advantages and disadvantages of each solution.

Properties of the Type System The type system is not particularly original compared to the one of the Godot system [1] except the rule T-Forward that elegantly solves the typing of synchronous calls with forward, as explained above. It shares the classical properties: *progress* (typing ensures that a well-formed program blocks only when a `get*` is performed on an unresolved future) and *preservation* (reduction does not break typing). Both properties are expressed on `DeF+F`, and also valid on `DeF` which is a subset of `DeF+F`.

For stating and proving preservation one needs to extend the type system in order to type configurations. The extension raises no particular difficulty: each thread is type-checked separately, we check that the type of values in the store fits with the type of the declared variables, and that for each future f of type `Flow[T]`, T is the type of the value stored in the future f (or computed by the thread that computes f), and a

Type judgements for expressions:

$$\text{(T-Var)}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-Subtype)}\ \frac{\Gamma \vdash z : B}{\Gamma \vdash z : \mathtt{Flow}[B]} \qquad \text{(T-Expression)}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v' : T' \quad \oplus : T \times T' \to T''}{\Gamma \vdash v \oplus v' : T''}$$

$$\text{(T-Invk-Async)}\ \frac{\Gamma(\mathtt{m}) = \overline{T} \to T' \quad \Gamma \vdash \overline{v} : \overline{T}}{\Gamma \vdash !\mathtt{m}(\overline{v}) :\downarrow \mathtt{Flow}[T']} \qquad \text{(T-Get)}\ \frac{\Gamma \vdash v : \mathtt{Flow}[B]}{\Gamma \vdash \mathtt{get*}\ v : B} \qquad \text{(T-Invk-Sync)}\ \frac{\Gamma(\mathtt{m}') = \overline{T} \to T' \quad \Gamma \vdash \overline{v} : \overline{T}}{\Gamma \vdash \mathtt{m}'(\overline{v}) : T'}$$

$$\text{(T-If)}\ \frac{\Gamma \vdash v : \mathtt{Bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathtt{if}\ v\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\}}$$

Type judgements for statements:

$$\text{(T-Assign)}\ \frac{\Gamma(x) = T \quad \Gamma \vdash e : T}{\Gamma \vdash_{\mathtt{m}} x = e} \qquad \text{(T-Seq)}\ \frac{\Gamma \vdash_{\mathtt{m}} s \quad \Gamma \vdash_{\mathtt{m}} s'}{\Gamma \vdash_{\mathtt{m}} s\ ;\ s'} \qquad \text{(T-Return)}\ \frac{\Gamma \vdash e : T' \quad \Gamma(\mathtt{m}) = \overline{T} \to T'}{\Gamma \vdash_{\mathtt{m}} \mathtt{return}\ e} \qquad \text{(T-Skip)}\ \frac{}{\Gamma \vdash_{\mathtt{m}} \mathtt{skip}}$$

Type judgements for programs and functions:

$$\text{(T-Method)}\ \frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}][\overline{x'} \mapsto \overline{T'}] \quad \Gamma' \vdash_{\mathtt{m}} s}{\Gamma \vdash T''\ \mathtt{m}\ (\overline{T\ x})\{\overline{T'\ x'}\ \ s\}} \qquad \text{(T-Program)}\ \frac{\Gamma = [\overline{x} \mapsto \overline{T}] \quad \Gamma[\overline{x'} \mapsto \overline{T'}] \vdash s \quad \forall M \in \overline{M}.\ \Gamma \vdash M}{\Gamma \vdash \overline{T\ x}\ \overline{M}\ \{\overline{T'\ x'}\ \ s\}}$$

Figure 3.3: Type system; each operator $\oplus$ has a predefined signature.

$$\text{(T-Forward)}\ \frac{\Gamma \vdash e : \mathtt{Flow}[T'] \quad \Gamma(\mathtt{m}) = \overline{T} \to \mathtt{Flow}[T']}{\Gamma \vdash_{\mathtt{m}} \mathtt{forward*}\ e}$$

Figure 3.4: Type system addition for `DeF+F`.

`get` f operation returns a value of type T. To type a runtime configuration, we need an extended typing environment of the form:

$$\Omega ::= \Gamma \quad \overline{f \mapsto \overline{\Gamma, m}} \quad \overline{f \mapsto T}$$

The first Γ is the type of the global store, then for each not yet resolved future, the future identifier is mapped to a stack of Γ, m where each Γ is the typing environment that types the function body and m is the function name that provides the returned type, and the type of the future for the last m of the stack; finally we have a second mapping for resolved futures that maps each future identifier to the type of the future value. The initial configuration of a well-typed program is well-typed. Then we can state the preservation theorem:

Theorem 1 (Preservation)**.** *A well-typed configuration remains well-typed during reduction.*

$$\Omega \vdash a \rangle F \quad \wedge \quad a \rangle F \rightarrow a' \rangle F' \implies \exists \Omega'.\ \Omega' \vdash a' \rangle F'$$

Ω and Ω' are additionally constrained: the global Γ is the same in Ω and Ω', and each future defined in Ω is also defined in Ω' with the same type.

Like Encore and most of the languages with futures, our language has imperative features. Thus, contrarily to the Godot system, it is possible to create cycles of processes where each process references the future that is to be resolved by the next process in the cycle. In imperative languages with futures, and therefore in `DeF` and `DeF+F`, it is possible to have deadlocks in such situations. We only ensure progress in the absence of such deadlocks. This restriction of the progress property is in fact an advantage of our model: it shows that `DeF` and `DeF+F` model faithfully the deadlocks that exist in mainstream languages with futures.

We introduce the *Unresolved* predicate that checks whether a future is unresolved. It is defined by: $\mathit{Unresolved}(f, F) = \nexists w.\, f(w) \in F$. We use it to state a progress property:

Theorem 2 (Progress)**.** *In a runtime configuration, each element that is an unresolved future can progress except if it tries to perform a* `get` *on an unresolved future.*

$$\begin{aligned}\Omega \vdash a \rangle f(\overline{q})\ F \implies & (\exists a'\ F'.\ a \rangle f(\overline{q})\ F \rightarrow a' \rangle F' \wedge f(\overline{q}) \notin F') \\ & \vee (\exists v, \ell, y, s, \overline{q'}.\ \overline{q} = \{\ell \mid y = \texttt{get}{*}\ v\ ;\ s\}\#\overline{q'} \wedge \mathit{Unresolved}([\![v]\!]_{a+\ell}, F))\end{aligned}$$

This theorem ensures that any chosen task can progress unless it is trying to access an unresolved future. Consequently, a configuration that cannot progress only consists of tasks blocked on a future access, which implies that there is a cycle of futures. More generally, any chosen task is able to terminate except if it depends (indirectly) on itself or on a task that contains a non-terminating computation (e.g. an infinite recursive call). Indirect dependency on oneself is the result of an actor asynchronously calling one of its own methods and waiting on the future (or flow) returned by this call. As actors are single threaded, the caller will block the actor in the synchronization operation, which will prevent the asynchronous call from being processed.

3.2 The forward-return equivalence

The previous section showed the runtime semantics for `return` in `DeF`/`DeF+F` and `forward*` in `DeF+F`. This section proves that these two constructs can be used interchangeably in `DeF+F`, by showing formally that the use of `forward*` instead of `return` does not alter the semantics of a `DeF+F` program. This positions the choice between `return` and `forward*` in `DeF+F` as an optimization matter rather than a semantics matter.

To prove this, we provide a translation from a `DeF+F` program to a `DeF` program that replaces `forward*` by `return` statements. We then show, using a branching bisimulation, that the translated program has the same observable behaviour as the original one.

Note on Bisimulations. Bisimulations allow the comparison of the semantics of asynchronous programs [68]. Given a binary relation $\mathcal{R}$, two transition systems are said to be strongly bisimilar if for two states related by $\mathcal{R}$, a series of transitions on either side can be associated with the same series of transitions on the other side, with each pair of intermediate states linked by $\mathcal{R}$. Most of the time a strong bisimulation is too strict to compare programs and one needs to consider that some of the transitions do not need to be simulated. A weak bisimulation relaxes the definition of strong bisimulation by allowing a subset of transitions to take place at any time even without an equivalent in the other transition system. These transitions are called non-observable, or τ-transitions.

This relaxation has a cost: two weakly similar programs might have some different properties, e.g. concerning the presence of deadlocks. Branching bisimulation is a compromise between strong and weak bisimulation that guarantees more properties but allows the presence of non-observable transitions. With a branching bisimulation a program that performs a τ-transition must remain in relation with the same states, guaranteeing that τ-transitions have almost no effect on the program state. A divergence-branching simulation is another compromise that further requires an infinite sequence of τ-transitions on either side to correspond to an infinite sequence of τ-transitions on the other side.

Below, we prove a branching bisimulation between a `DeF+F` program and its translation to a `DeF` program with every `forward*` replaced by a `return`. Our semantics cannot ensure divergence-branching simulation because of the different synchronization strategies between `return` and `forward*`: getting the value of a future that is in a cycle will block on the `DeF+F` side, but enter an infinite sequence of τ-transitions on the `DeF` side.

3.2.1 Translation from `DeF+F` to `DeF` and Program Equivalence

A `DeF+F` program can be translated into a plain `DeF` program using the semantics-preserving transformation $[\![]\!]_{fwdElim}$ defined as follows:

$$[\![\texttt{forward}*\ v]\!]_{fwdElim} \triangleq \texttt{return}\ v$$

Terms other than `forward*` are unchanged. To prove that $[\![]\!]_{fwdElim}$ actually preserves the semantics of a `DeF+F` program, we define in Figure 3.5 a relation $\mathcal{R}$. It relates a `DeF+F` configuration cn_{F} and a `DeF` configuration cn_{D} that represents a similar state of execution. We prove that two configurations related by $\mathcal{R}$ are bisimilar. More precisely, the equivalence we prove is a branching bisimulation that does not observe the update of intermediate futures in a sequence of chained futures (i.e. considers Get-Future and Chain-Update as τ-transitions) as the precise resolution status of a future is an internal state that does not matter for the observable state of a program.

The trivial rules $\mathcal{R}$-Id-Store and $\mathcal{R}$-Id-Resolved state that two identical configurations are similar. $\mathcal{R}$-ForwardElim deals with syntactic equality modulo `forward*` elimination, simply replacing `forward*` by `return`.

These rules are not sufficient, as Chain-Updates can happen at any time on the `DeF+F` side, making the executions of a `DeF+F` program and its `DeF` counterpart slightly

$\mathcal{R}$-ID-STORE
$$\frac{}{a \;\mathcal{R}\; a}$$

$\mathcal{R}$-ID-RESOLVED
$$\frac{cn_{\text{F}} \;\mathcal{R}\; cn_{\text{D}}}{cn_{\text{F}}\; f(w) \;\mathcal{R}\; cn_{\text{D}}\; f(w)}$$

$\mathcal{R}$-FORWARD-ASYNC
$$\frac{cn_{\text{F}} \,\mathcal{R}\, cn_{\text{D}}}{cn_{\text{F}}\; f(\texttt{chain}\; f') \;\mathcal{R}\; cn_{\text{D}}\; f(f')}$$

$\mathcal{R}$-FORWARDELIM
$$\frac{cn_{\text{F}} \,\mathcal{R}\, cn_{\text{D}}}{cn_{\text{F}}\; f(\{\ell \mid s\}\#\overline{q}) \,\mathcal{R}\, cn_{\text{D}}\; f(\{\ell \mid [\![s]\!]_{fwdElim}\}\#\overline{[\![q]\!]_{fwdElim}})}$$

$\mathcal{R}$-CHAIN-UPDATE
$$\frac{cn_{\text{F}}\; f(\texttt{chain}f')\; f'(w) \;\mathcal{R}\, cn_{\text{D}}}{cn_{\text{F}}\; f(w)\; f'(w) \,\mathcal{R}\, cn_{\text{D}}}$$

$\mathcal{R}$-GET-FUTURE-F
$$\frac{cn_{\text{F}}\; f(f')\; f''(\{\ell_{\text{F}} \mid y = \texttt{get*}\; f; s_{\text{F}}\}\#\overline{q}) \;\mathcal{R}\; cn_{\text{D}}}{cn_{\text{F}}\; f(f')\; f''(\{\ell_{\text{F}} \mid y = \texttt{get*}\; f'; s_{\text{F}}\}\#\overline{q}) \,\mathcal{R}\; cn_{\text{D}}}$$

$\mathcal{R}$-GET-FUTURE-D
$$\frac{cn_{\text{F}} \;\mathcal{R}\; cn_{\text{D}}\; f(f')\; f''(\{\ell_{\text{D}} \mid y = \texttt{get*}\; f; s_{\text{D}}\}\#\overline{q})}{cn_{\text{F}} \;\mathcal{R}\; cn_{\text{D}}\; f(f')\; f''(\{\ell_{\text{D}} \mid y = \texttt{get*}\; f'; s_{\text{D}}\}\#\overline{q})}$$

Figure 3.5: Relation between `DeF+F` configurations and `DeF` configurations.

different. We still want these configurations to be related by $\mathcal{R}$, which is required by the first item of Theorem 3, as CHAIN-UPDATE is a τ-transition.

$\mathcal{R}$-FORWARD-ASYNC and $\mathcal{R}$-CHAIN-UPDATE deal with the fact that some futures are chained and others are not. The rule $\mathcal{R}$-FORWARD-ASYNC states that chaining a future to another one, as the semantic rule FORWARD-ASYNC does, is semantically equivalent to fulfilling it with this same future, as RETURN-ASYNC does. Rule $\mathcal{R}$-CHAIN-UPDATE can be used to undo the future chaining operation.

As the resolution of futures is done in a different order, and possibly at a different time, when `forward*` is used instead of `return`, the rule $\mathcal{R}$-CHAIN-UPDATE and both $\mathcal{R}$-GET-FUTURE rules are needed to associate configurations in which some futures are at different stages of resolution. This different ordering only occurs upon resolution of the `get*` statement, and is handled by the two GET-FUTURE rules.

This different ordering of future resolution also implies that there is not a one-to-one mapping between CHAIN-UPDATE transitions in a `DeF+F` execution and GET-FUTURE transitions in the context of that same program after `forward*` elimination. All these facts are formalised by Theorem 3.

3.2.2 Branching Bisimulation between `DeF+F` and `DeF`

Theorem 3 (Correctness of the translation from `DeF+F` to `DeF`)**.** $\mathcal{R}$ *is a branching bisimulation between the operational semantics of the* `DeF+F` *program* P *and the operational semantics of the* `DeF` *program* $[\![P]\!]_{fwdElim}$.

Let R *range over observable transitions. If* $cn_F \,\mathcal{R}\, cn_D$ *then:*

$$cn_F \xrightarrow{\tau}{}^{*} cn'_F \implies cn'_F \,\mathcal{R}\, cn_D \qquad\qquad cn_D \xrightarrow{\tau}{}^{*} cn'_D \implies cn_F \,\mathcal{R}\, cn'_D$$
$$cn_F \xrightarrow{R} cn'_F \implies \exists\, cn'_D.\; cn_D \xrightarrow{\tau}{}^{*} \xrightarrow{R} cn'_D \wedge cn'_F \,\mathcal{R}\, cn'_D$$
$$cn_D \xrightarrow{R} cn'_D \implies \exists\, cn'_F.\; cn_F \xrightarrow{\tau}{}^{*} \xrightarrow{R} cn'_F \wedge cn'_F \,\mathcal{R}\, cn'_D$$

The transitions GET-FUTURE *and* CHAIN-UPDATE *are non-observable, both of them are labelled τ. The observable transitions* FORWARD-ASYNC *and* FORWARD-DATA *are labelled* RETURN-ASYNC*, and* FORWARD-SYNC *is labelled* RETURN-SYNC*. All the other transitions are labelled with their original rule name.*

As the use of `forward*` does not change the moment at which futures are created but only how they are resolved, if $cn_{\mathrm{F}} \mathcal{R} cn_{\mathrm{D}}$ then the identifiers of the futures of cn_{F} are exactly those of cn_{D}, e.g. we have $f \in cn_{\mathrm{F}} \iff f \in cn_{\mathrm{D}}$. However, the value of f in cn_{F} and the value of f in cn_{D} may differ. In the following, we denote that some futures $f \in cn_{\mathrm{F}}$ and $f' \in cn_{\mathrm{D}}$ actually share the same identifier by $f = f'$ or $f \equiv f'$.

First, we introduce the notion of sequence of futures, and define a few lemmas on properties implied by $cn_{\mathrm{F}} \mathcal{R} cn_{\mathrm{D}}$. This will help to prove the branching bisimulation.

Definition 4 (Sequence of futures)**.** *In a* `DeF` *or* `DeF+F` *configuration cn,* $(f_0 \dots f_n)$ *such that* $\forall i, f_i(f_{i+1}) \in cn \vee f_i(\texttt{chain } f_{i+1}) \in cn$ *is called a* sequence of futures.

Given a future resolved or chained to another one in a `DeF+F` configuration, Lemmas 1 and 2 state the possible forms of the corresponding future in an associated `DeF` configuration. Conversely, given a future resolved to another one in a `DeF` configuration, Lemma 3 states the possible forms of the corresponding future in the associated `DeF+F` configuration. Lemma 5 generalises Lemmas 1 and 2: given a sequence of futures in a `DeF+F` configuration, it states the possible forms of the corresponding sequence in `DeF`, while Lemma 6 generalises Lemma 3 in a similar manner. As for Lemma 4, it formalizes that the local store and statements of a task are not altered by $[\![]\!]_{fwdElim}$, apart from `get*` statements that can express a different stage of future resolution.

Lemma 1 (Matching a chained future)**.** *If* $cn_F \mathcal{R} cn_D$ *and* $f(\texttt{chain } f') \in cn_F$*, then* $f(f') \in cn_D$*.*

The case of a future resolved on the `DeF+F` side is more complicated, as such a future can not only come from a `return` statement, but also from a CHAIN-UPDATE. The following lemma illustrates that the `chain` construct can flatten chains of futures.

Lemma 2 (Resolved future in `DeF+F`)**.** *If* $cn_F \mathcal{R} cn_D$ *and* $f(w) \in cn_F$*, then there exists* $f_0(f_1) \dots f_{n-1}(f_n) \in cn_D$ *such that* $f_0 \equiv f$*,* $f_n(w) \in cn_D$*, and* $\forall i, f_i(w) \in cn_F$*.*

The previous two lemmas dealt with futures resolved on the `DeF+F` side, the next one deals with futures resolved on the `DeF` side.

Lemma 3 (Resolved future in `DeF`)**.** *If* $cn_F \mathcal{R} cn_D$ *and* $f(w) \in cn_D$*, then:*

- *Either w is a future and* $f(\texttt{chain } w) \in cn_F$*.*
- *Or there exists* w' *such that* $f(w') \in cn_F$ *and there exists* $f_0(f_1) \dots f_{n-1}(f_n) \in cn_D$ *such that* $f \equiv f_0$*,* $f_n(w') \in cn_D$*, and* $\forall i, f_i(w') \in cn_F$*.*

Given a resolved or chained future in a `DeF` or `DeF+F` configuration, the previous lemmas gave the form of the corresponding future on the other side. The next lemma deals with the last case: futures not yet resolved, that still have a task attached to them. As the $[\![]\!]_{fwdElim}$ transformation is fairly simple, the local store and most statements will be identical on both sides, but `get*` statements may differ. Indeed, following a sequence of futures by the rule Get-Future being non-observable, $\mathcal{R}$ has to relate `get*` statements at different stages of update. In this case, walking back the chain of Get-Future leads to the same initial future.

Lemma 4 (Matching tasks)**.** *If* $cn_F \, \mathcal{R} \, cn_D$, *then* $\exists s.\, f(\{l \mid s\}\#\overline{q}) \in cn_F$ *if and only if* $\exists s'.\, f(\{l \mid s'\}\#\overline{[\![q]\!]_{fwdElim}}) \in cn_D$. *In this case:*

- *Either* s *is of the form* $y = \texttt{get*}\ w; s_1$, *and* s' *of the form* $y = \texttt{get*}\ w'; [\![s_1]\!]_{fwdElim}$, *with*

$$\exists w_0 \dots w_n \in cn_F.\ \exists w'_0 \dots w'_m \in cn_D \begin{cases} \forall i < n\ w_i(w_{i+1}) \in cn_F \\ \forall i < m\ w'_i(w'_{i+1}) \in cn_D \\ w_0 \equiv w'_0 \\ w_n = w \\ w'_m = w' \end{cases}$$

- *Or* $s' = [\![s]\!]_{fwdElim}$.

The next, and final, two lemmas are generalizations of the three first ones. They answer the question: given a sequence of futures, on the `DeF` side or on the `DeF+F` side, what can we say about the other side? Lemma 5 is about the `DeF` to `DeF+F` case, while Lemma 6 handles the other direction.

Lemma 5 (Sequence of futures: `DeF` to `DeF+F`)**.**
If $cn_F \, \mathcal{R} \, cn_D$ *and* $f_0(f_1) \dots f_{n-1}(f_n)\ f_n(w) \in cn_D$ *with* $\nexists f \in cn_D.\, w = f$, *then there exists* $k_0 \leqslant \dots \leqslant k_l$ *such that for all* $i < l$ *either* $f_{k_i}(f_{k_{i+1}}) \in cn_F$ *or* $f_{k_i}(\texttt{chain}\ f_{k_{i+1}}) \in cn_F$ *with* $k_0 = 0$ *and* $k_l = n$.

Lemma 6 (Sequence of futures: `DeF+F` to `DeF`)**.**
If $cn_F \, \mathcal{R} \, cn_D$ *and* $f_0(f_1) \dots f_{n-1}(f_n)\ f_n(w) \in cn_F$,
then there exists $f'_0(f'_1) \dots f'_{l-1}(f'_l) \in cn_D$ *such that* $f'_0 = f_0$ *and* $f'_l(w) \in cn_D$.

The proof of Theorem 3 is done by induction and a classical case analysis on the reduction rule applied, proving that the bisimulation relation is maintained. Details of the proof are not relevant to this manuscript and can be found in [69], courtesy of co-author Nicolas Chappe. The most interesting cases are the Get-Data rules, where a future is about to be resolved on one side, but there may still be multiple τ-transitions needed to get to the resolution on the other side.

In this section we have shown that, with data-flow explicit futures, the behavior of the `forward*` primitive is the same as the behaviour of a standard `return`. This highlights the fact that `forward` provides a form of data-flow synchronization for control-flow

explicit futures. More interestingly this shows that if necessary any `return` statement that returns a data-flow future could be compiled similarly to a `forward*` statement, which should in general improve performance.

3.3 Implementation in Encore and Benchmarks

In this section we present our implementation of dataflow explicit futures in the Encore language. This implementation allowed us to explore some of the possibilities offered by these new futures. In particular we built control-flow futures on top of our own dataflow futures: we present our implementation as it can be used as a basis for other languages to do the same. Recall that in Section 3.1.4 we provided a typing rule for `forward*` that slightly differs from the one in [1], and is safer. We discuss the differences in practice between these two choices, and why the typing rule we chose in this implementation is the one of [1] rather than the safer one. Finally, we investigate the performance of different implementations and synchronization strategies on programs that express various communication patterns.

3.3.1 Prerequisites — The Encore Programming Language

Since our implementation was made in Encore, a basic understanding of how an Encore program is executed is required. In this section we consider the original version of Encore, without flows. Section 3.3 onward will present our implementation.

The Encore runtime Encore is a compiled language. The compiler, written in Haskell, first translates an Encore program to C code that is compiled and linked with the Encore runtime (libencore) and the Pony runtime together into a single executable. The Pony runtime [70] is an actor runtime; Encore uses it to handle the active-objects parts of the language, namely stack switching and message transmission. The Encore runtime contains the definitions of built-in types and operators that are used by programmers.

The Encore runtime is written in C to achieve higher performance than with raw Encore code. The translation from Encore to C will usually yield less efficient C code than raw written C, as the translation process does not perform a lot of optimization. Moreover, the most crucial primitives, like `future_get` are directly implemented in C, with optimized and lower-level code. The runtime can also be used to allow Encore programs to access system calls and other libraries functions.

Translation of Encore to C and handling of messages The Encore runtime is linked against the Pony runtime, written in C, which handles the actor aspects. Given a function definition `f`, the Encore compiler translates this function once, and write multiple wrappers around it[5]. Each wrapper corresponds to a different call mechanism:

[5]To keep things simple, we limit ourselves to the translation of functions in this paragraph. The process is similar for methods, but the presence of classes makes it slightly more complex.

synchronous call, asynchronous call, asynchronous call that ignores the returned future etc. When translating a call to `f`, the compiler emits a call to the appropriate wrapper.

Let us focus on the asynchronous C version, as it will be the source of problems. When an Encore program executes the statement `async(f)`, this is translated in the C program as a call to the asynchronous version of `f`, let us call it `f_async`. The code of `f_async` will use the Pony runtime in order to create a new message, which will be processed at a later time. In addition, `f_async` creates the future that will contain the result of the message, and returns it to the caller. Finally, `f_async` stores the future inside the message itself, so the target of the message can fulfill it. Processing the message consists in - invoking the synchronous C version of `f`, and storing the result inside the future associated with the message.

A primer on genericity in Encore In Section 3.3.2.3 we will present a difficulty in implementing the `get*` operation in Encore due to generic types. This paragraph quickly introduces genericity and the common compilation schemes to deal with it, *i.e.* type erasure and compile time expansion. Listing 3.4 shows an example of genericity in Encore. It defines the identity function, called `id`, as a generic function over any type `t`. In other words, the function will accept any value of any type as its sole parameter.

Listing 3.4: Example of genericity in Encore

```
1 fun id[t](x: t) : t
2   return x
3 end
```

Programming languages usually handle generic types in one of two ways: type erasure or compile time expansion. Compile time expansion is the process used by C++ generic types, the templates. This process consists in compiling a different version of the function for each value given to its generic types and replacing every call to the generic function with a call to the appropriate version. For instance, if the identity function was implemented in a C++ program, and used with twelve different types then the compiler would generate twelve different versions of the function, one for each different type. The main advantage of this process is that the compiler knows the exact types used in every invocation of functions. Moreover, as expansion is performed early during the compilation process, further typing operations, as well as optimization and code generation will be performed on this new code where genericity has been completely removed.

Type erasure chooses to abstract away types under a generic name. This is the process used in many languages such as Java or C#. In the previous example (Listing 3.4), typechecking will behave as if `t` is an existing type, and the typechecking rules defined in the language will be applied. In other words, the compiler has no knowledge of the true type of the values given as parameters: the type is effectively erased. Encore uses type erasure for its implementation of generic types. During the translation from Encore to C, Encore generic types become generic C pointers `void*`. While this results in a smaller sized executable, it also prevents the compiler from reasoning on types which may impose constraints on optimization and expressivity. Moreover, it impacts code

generation, as the compiler will emit code that must work on every possible type used to instantiate the generic type.

Translation of Encore generics into C Depending on the programming language, generics may be handled in different ways. In the case of Encore, since it translates to C, the (limited) features of C that allow genericity must be used. In the Encore runtime, the concept of any type is represented as an *untagged* union that contains the following type: integers, booleans, floating point numbers and pointers. Compound C types, such as structures or unions, are not typed as-is in the union, as their instances are stored by address rather than by value. In the context of the Encore runtime, the union does not need to be tagged due to how the compiler processes generics. Whenever the compiler translates a generic type, it translates it to the type of the untagged union. In Encore, the only operation available on values of a generic type is the assignment operation, and assignment of values of a generic type are safe. In C, assignment of values of the same union type to each other is safe, therefore the natural translation of operations on values of a generic type from Encore to C are all type safe. Because operations on values of generic type are limited to assignment, the question of accessing the content of the union only arises when the concrete type of the value is known. For instance, the following code does not type in Encore because the `+` operation is not defined on values of a generic type: `fun f[t](x: t, y: t): t return x + y end`. However, this code types, because the compiler is able to unify the generic type `t` with `int`: `println(id(12)+ id(13))`. In practice, the compiler will translate the Encore `id` function to a C function whose return type is the union of all possible types in Encore. The call `id(12)` will be translated to a call to this C function and it will read the value inside the union as an integer, since the compiler has unified `t` with the integer type.

3.3.2 Implementation of `Flow`

For the reasons evoked in the previous section, we implemented flows in the Encore runtime and in the compiler, rather than as a library written in Encore.

The process required to implement dataflow explicit futures in this way is made of three-steps: 1) A modification of the type system, in order to support the new built-in `Flow[T]` type; 2) A modification of the runtime itself in order to add the runtime code for `Flow` and 3) Modifying the syntax of the language and the compilation scheme in order to support the new primitive operators on dataflow futures. The main difficulty lies in the handling of the `get*` primitive for reasons we will detail in Section 3.3.2.3.

3.3.2.1 Typing

Compared to traditional control-flow futures, typing dataflow futures requires special care. Recall both the collapsing and subtyping rules in the type system. The collapsing rule, T-Invk-Async, that corresponds to the application of the $\downarrow$ operator, is used to prevent the appearance of nested flows at the static type level system. This typing rule

allows the creation of chains of flows of a statically unknown length, most commonly observed on recursive functions.

The subtyping rule is used for the process of *lifting*, which allows non-flow values to be promoted to their future selves. This allows code reuse, as a function with a parameter typed `Flow[int]` may now accept either an int or a flow as its parameter[6].

Implementation in Encore The implementation of these two typing rules in Encore is straightforward. The collapsing rule requires distinguishing two cases: written types, and temporary types. Written types are those given by the programmer, where nesting should be rejected as in `var a: Flow[Flow[int]] = 12`, and should be checked every time a declaration is typechecked. The type of a temporary is the type of a value produced by a function / method / operator call before its type gets checked against an expected type, as in `var res = f()`: in this case, the type of the expression `f()` should be collapsed in order to remove the nesting that may involuntarily arise. For instance, if we have `fun f ↪ (): Flow[int]`, then the value produced by `async*(f())` is not ill-typed, even though the type of this expression is, before collapsing, `Flow[Flow[int]]`.

About subtyping The typing rule T-SUBTYPE in Figure 3.3 states that any type `T` is a subtype of `Flow[T]`. In other words, any value of type `T` can be used in place of a value of type `Flow[T]`. In the case of a language like Encore that first compiles to another language, the subtyping rule needs to be implemented in both languages. In an Encore program, the expression `var a: Flow[int] = 12` is well-typed. The translation of this expression in C must remain well-typed, however the C language has a type system that does not allow subtyping and forceful conversion of values between types usually results in undefined behavior. The information stored in an integer and a flow integer is not the same, therefore their representations in memory do not have the same layout. In this case, the AST is modified prior to code generation and an explicit conversion to flow is inserted. This explicit conversion to flow will translate, during code generation, to a call to a function in the runtime that will promote the non-flow value to a flow value.

3.3.2.2 Creation of `Flow`

In the context of `DeF+F`, flows are created through the use of the “`!`” operator, and only through this one. The creation of a flow creates another stack frame, whose final value will be used to resolve the associated flow. We now present how flows are created in Encore.

Implementation in Encore In the context of Encore, flows are created in three different contexts: 1) The usage of the `!!` operator, which is used to asynchronously call a method on an active object; this mirrors the `!` operator, which performs an asynchronous call that returns a control-flow future; 2) When lifting non-flow values into their flow selves; and 3) When using the `async*` operator: it launches the evaluation

[6] With control-flow futures this would require the programmer to write two different functions.

of an expression in another task, and returns a future that will hold the result of the evaluation.

The creation of a `Flow` when lifting a non-flow value is a special case as it is the only one that does not result from an asynchronous call. If we consider the following declaration in Encore, `var a: Flow[int] = 2`, it is trivial that the flow `a` is resolved with the value 2 of type `int`. Now, if we consider the function in Listing 3.5, it is impossible to determine, locally[7] and at compile time, whether `t` is instanciated with a flow during a call or not.

Listing 3.5: Example of a generic value being lifted into a flow

```
1   fun f[t](x: t): int
2   var y: Flow[t] = x
3   -- ...
4   end
```

As a result, it is impossible to determine, at compile-time, whether flow `y` is resolved with a flow or not. The necessity to keep track of which values are flows and which are not is crucial in the implementation of the `get*` operator, and traces back to the GET-FUTURE and GET-DATA rules in the semantics of `DeF`.

Section 3.3.2.3 presents the problem encountered, and the solution we provided. Section 3.3.2.4 details a similar problem encountered when a value of a generic type should be lifted to a flow, but statically the type `Flow[T]` never appears.

3.3.2.3 `Flow` synchronization with `get*`

In `DeF` Recall that in Figure 3.2 we introduced two rules, GET-FUTURE and GET-DATA that are used to recursively traverse a chain of flows until a non-flow value is reached. GET-DATA can only be applied to a non-flow value. GET-FUTURE can only be applied to flow values. When we move outside of the theoretical world, this assumes there is a way for the compiler / interpreter to detect whether a value is a flow or not, in order to know which operation to perform.

In programming languages We need to distinguish three cases: languages without parametric types, languages with parametric types with introspection and languages with parametric types without introspection.

No parametric types In a language with no parametric types, static analysis is enough to detect, at the time of creation of a flow, whether it will be resolved with a flow or not. Indeed whether the flow is created by lifting of a non-flow value or by asynchronous call, inspecting the static type of a term is enough to know whether the dynamic type will be a flow or not. Consider the example in Listing 3.6, written in Encore without parametric types. `async*` asynchronously calls a function and returns a future that will hold the result. There are two flows created here, both on line 8 (by

[7] Due to type erasure, interprocedural analysis is not possible.

Listing 3.6: Synchronization with `get*` in C

```
1  fun f(flow: Flow[int]): int
2    val a: int = get*(flow)
3    return a * 2
4  end
5
6  active class Main
7    def main(): unit
8      var a: Flow[int] = async*(f(2))
9      println(get*(a))
10   end
11 end
```

Listing 3.7: Synchronization with `get*` in Encore

```
1  fun id[t](x: t): t
2    return x
3  end
4
5  fun g[t](x: t): Flow[t]
6    var res: Flow[t] = async*(id(x))
7    return res
8  end
9
10 active class Main
11   def main(): unit
12     var a: Flow[int] = async*(id(2))
13     var b: Flow[int] = g(a)
14     println(get*(b))
15   end
16 end
```

asynchronous calls or by lifting). Static type is enough to know in the example which types are flow or not.

With parametric types In a language with parametric types, static analysis may no longer be enough. Consider the example in Listing 3.7, written in Encore with parametric types. This code creates two generic functions: the identity function `id` and a toy function `g` that asynchronously calls the identity function, forwarding it its parameter.

In the `main` function flow `a` is created by asynchronously calling the identity function, and flow `b` is created by passing flow `a` as parameter to `g`. Then, we get the value of `b` and print it. `a` is resolved with a non-flow value, here 2. This resolution can occur at any moment due to the asynchronous nature of the call to `id(2)`. The resolution of `b` is

more complex, we give a step-by-step execution:

1. Flow `a` is created.
2. The call `g(a)` is performed. Control-flow enters the `g` function, with its parameter `x` bound to `a`.
3. In `g`, a flow `res` is created as the result of the asynchronous call to `id(x)`. `id(x)` is equivalent to `id(a)`. `res` is therefore a flow that will be resolved with a flow (`a`), that will be resolved with a value (2).
4. `g` returns `res`.
5. In `main`, `b` is now bound to `res`. This makes `b` a flow that will be resolved with a flow (`a`) that will be resolved with a value (2).

It is important to note that there is no “moment” at which the resolution of each of these flows can occur. The asynchronous model of computation of Encore offers no guarantee on when a resolution will occur. It is possible for `a` to already be resolved with 2 when the call to `g` is performed, and it is also possible for `a` to not even be resolved when the call to `get*(b)` is performed in `main`.

From a static point of view, in function `g`, `t` is not `Flow[T]`, it is a generic type. We cannot statically determine whether or not asynchronously calling this function will produce a flow resolved with a flow or with a value. Worse, the naive approach would be to simply check whether the return type is `Flow`, observe it is not, and conclude that the function will never produce a `Flow`. In this situation, the correct solution is to check the actual runtime type of this value. This can be done using the introspection capabilities of the language, or by adding runtime information about the type.

Listing 3.8 presents our implementation of `get*` in the Encore runtime as the `get_star` function. The `flow_t` type represents a `Flow` in the runtime. This type has a field `type` which represents the type of value stored in the `Flow`. This field is filled by the runtime. `ID_FLOW` is a constant that indicates a value has a type of `Flow`.

Listing 3.8: `get*` in the Encore runtime with reflection added

```
1 encore_arg_t get_star(flow_t* flow) {
2     // Checks for fulfillment not shown
3 
4     if (flow->type->id == ID_FLOW) {
5         return get_star((flow_t*)flow->value.p);
6     } else {
7         return flow->value;
8     }
9 }
```

3.3.2.4 Lifting in the context of a generic function

Lifting consists in inserting conversion from non-flow values to flow resolved with the value when needed. This conversion is typically needed in assignments like `a = b` with `b` being a non-flow and `a` being a flow. In most cases, this can be decided statically in a simple way: when the types of `a` and `b` are non-parametric, or when they both have the

same type (in which case the compiler may not know whether the type is a flow, but knows that no conversion is needed). However, the question of whether a conversion must be inserted cannot be decided statically in the presence of parametric types and `get*`. An example is given in Listing 3.9, where the assignment line 3 may or may not require a conversion depending on the runtime type of `t`.

Listing 3.9: Lifting in parametric functions in Encore

```
fun f[t](x: t) : t
  var flow: Flow[t] = async*(id(x))
  var res: t = get*(flow)
  return res
end
```

We focus on the statement `var res: t = get*(flow)`. We know, by definition, that the type of `get*(flow)` is not a flow. The naive translation of this statement would be an assignment of the result of `get*` to a value of the generic type in the runtime library. However, this raises an issue when the generic type `t` is instanciated with `Flow[T]`. At runtime, the left part of the assignment has type `Flow[T]`, but the right part has type `T` that is not a flow. This means the result of `get*` needs to be lifted. As this cannot be determined statically, we modified the compiler so it emits a check of the type used to instantiate `t`, which will determine whether the result of `get*` needs to be lifted or not.

Listing 3.10 presents the code emitted after modifying the compiler to account for the generic type. `encore_arg_t` is the C type that represents generic type (the union). The array `gen_types` provides a limited form of introspection. Entry 0 is associated with the Encore generic type `t` and indicates with what type it was instanciated during this specific call. `ID_FLOW` is the identifier of the `Flow` type in the runtime.

Listing 3.10: Translation of a generic Encore function using reflection to properly lift non-flow values to their flow selves if needed

```
encore_arg_t f_sync(encore_arg_t x, pony_type*[] gen_types) {
  flow_t* f = id_async(x);

  encore_arg_t result;
  if (gen_types[0]->id == ID_FLOW) {
    result.p = flow_mk_from_value(get_star(f));
  } else {
    result = get_star(f);
  }

  return result;
}
```

In both listings, identifier `ID_FLOW` (line 4 in Listing 3.8, line 5 in Listing 3.10) is used in the runtime to identify a value of type flow. The C structure representing a flow, `flow_t`, contains a field of type `pony_type` whose `id` is set to `ID_FLOW` if the value that resolves the flow is another flow, something else otherwise. In the implementation of

Figure 3.6: Typing rule for `forward*` in Godot

$$\text{(T-Godot-Forward*)}\quad \frac{\Gamma \vdash e : \texttt{Flow[T']} \qquad \Gamma(\texttt{m}) = \overline{T} \rightarrow T'}{\Gamma \vdash_{\texttt{m}} \texttt{forward*}\ e}$$

`get*`, if this field designates a flow, then `get_star` calls itself recursively on the resolving flow. Otherwise it merely returns its resolving value.

On line 1 of Listing 3.10, the array of `pony_type*` identifies the runtime types of every parametric type of the function during a given call. On line 5 we check if the runtime type of the parameter `x` of the current invocation of `f` is flow, and if it is we lift the result of `get*` into a flow. Otherwise we simply return the result of `get*`.

Summary The formal semantics proposed in Section 3.1.2 assumes the existence of an introspection operator that allows the runtime of a language to detect whether a value is a flow or not. In practice, languages without parametric types and without such introspection features can rely on static typing instead. In the context of languages with parametric types, it may be solved using built-in introspection facilities. If no such facilities exist, then the language developer should implement them in order for `get*` to work[8].

3.3.3 `forward*` : Typing and consequences in implementation

The `forward*` primitive allows delegating the resolution of a flow to another task. It is the dataflow equivalent of the `forward` primitive introduced in [48]. In Section 3.1.4 we introduced a typing rule for functions that perform a `forward*` and we pointed out that this typing rule slightly differs from the one presented in [1]. We argued that our typing rule is safer.

Godot's typing of `forward*`, and the original `forward` The typing rule in [1] for `forward*` would be rule T-Godot-Forward* in Figure 3.6. This would let us write Listing 3.11.

Listing 3.11: Using `forward*` in Encore with Godot's typing

```
1 fun f(): int
2   val x: Flow[int] = async*(id(12))
3   forward*(x)
4 end
```

We call this typing of `forward*` the *flexible way*. The choice for this typing rule was inspired by the typing of the `forward` primitive for control-flow futures in [48]. This

[8]Other solutions exist. Adding limited compile time expansion of generic functions is another one, although it can be more costly in executable size

Figure 3.7: Semantic rule for `forward` in a synchronous call, based on the behavior of such a call in Encore

$$\textsc{CeF-Forward-Sync}\quad \frac{[\![v]\!]_{a+\ell} = w \qquad y \text{ fresh variable}}{\begin{array}{c} a \rangle\, F\ f(\{\ell' \mid \texttt{forward}\ v\ ;\ s\}\#q\#\overline{q}) \\ \rightarrow a \rangle\, F\ f(\{\ell' \mid y = \texttt{get}\ w\ ;\ \texttt{return}\ y\ ;\ s\}\#q\#\overline{q}) \end{array}}$$

choice in [48] made sense when one considers the objectives of `forward`: mitigate the future proliferation problem (the appearance of nested futures at the type system level). However, when implementing `forward` in a language like Encore that supports both synchronous and asynchronous call of functions, this raises the question: what is the semantics of `forward` inside a method called synchronously? The solution adopted in Encore is to add an implicit synchronization on the forwarded future. We translate this behavior by the rule CeF-Forward-Sync in Figure 3.7[9].

In the context of `DeF+F`, if we used typing rule T-Godot-Forward* to type `forward*`, we would have an identical rule to CeF-Forward-Sync to describe the semantics of `forward*`, with `forward` becoming `forward*` and `get` becoming `get*`. In other words, we would need to introduce an implicit synchronization.

This is in contrast to the way we type `forward*` in `DeF+F`. We force the return type of a function performing a `forward*` to be a flow. In addition, because of the `forward*-return` equivalence, we can give a semantics for a synchronous call to a function performing a `forward*`: evaluate `forward*` like `return` statements. This safer typing rule is called *the strict way.*

We now need to choose how to implement the typing of `forward*`, following one of the two approaches presented above.

Comparison of the two approaches and argument of safety The main point of discussion with the typing of `forward*` is the semantics given to its execution. As we have seen, with the flexible typing of Godot, we need to perform an implicit synchronization.

This can lead to subtle deadlocks, as illustrated in Listing 3.12. This code will deadlock on line 3. We present below the code executed step by step.

1. The program starts with a single active thread that runs an instance of `Main` class and executes its `main` function.
2. In `main`, the call to `new Foo()` creates a new active object of type `Foo` that starts executing in its own thread. It waits for messages to process.
3. In the `main` function, the asynchronous call `foo!!foo()` creates an unresolved flow, we call it f_1. It also sends a message to `foo`.
4. The `main` thread enters `get*` and is blocked until f_1 is resolved with a value, *i.e.* it can follow a chain of flows until it reaches a value that is not a flow. The `foo`

[9]This rule is not part of the original `forward` paper [48], nor a part of Godot. It is our translation in `DeF+F` of the behavior of `forward` in a synchronous call in Encore.

Listing 3.12: Deadlock in Encore when using the flexible typing of `forward*`

```
active class Foo
  def foo(): int
    return this.bar()
  end

  def bar(): int
    -- In a synchronous call, translates to
    -- var f: Flow[int] = this!!baz()
    -- return get*(f)
    forward*(this!!baz())
  end

  def baz(): int
    return 12
  end
end

active class Main
  def main(): unit
    val foo: Foo = new Foo()
    println(get*(foo!!foo()))
  end
end
```

thread starts processing the call to `foo`.

5. In `foo`, a synchronous call is made to `bar` on the same object. Synchronous calls on oneself are allowed in the active-object model and correspond to traditional function calls in imperative programming. Thread `foo` enters `bar`.
6. In `bar`, a flow f_2 is created by `this!!baz()`. The current object sends a message to itself. Then, the current object performs `get*(`f_2`)`, as the currently active method was called synchronously. Because the current object is already processing the initial call to `foo` that created flow f_1 in `main`, this call to `get*` blocks the actor as said actor never enters a state in which it can process the call to `baz`.

This deadlock is especially difficult to spot for the programmer: the code does not perform any explicit synchronization within the body of the class. This breaks the property of explicit futures that ensures that all synchronizations are explicit.

On the other hand, the strict typing can be used to mitigate the problem. Listing 3.13 presents a modified version of Listing 3.12 that uses the strict way to type `forward*`. There are two modifications here: method `foo` now performs a `get*` on the result of the call to `bar`, and `bar` now returns a flow. Step-by-step execution, with step 6 changed and step 7 new:

1. The program starts with a single active thread that runs an instance of `Main` class and executes its `main` function.

Listing 3.13: Behaviors in Encore when using the strict typing of `forward*`

```
active class Foo
  def foo(): int
    return get*(this.bar())
  end

  def bar(): Flow[int]
    forward*(this!!baz())
  end

  def baz(): int
    return 12
  end
end

active class Main
  def main(): unit
    val foo: Foo = new Foo()
    println(get*(foo!!foo()))
  end
end
```

2. In `main`, the call to `new Foo()` creates a new active object of type `Foo` that starts executing in its own thread. It waits for messages to process.
3. In the `main` function, the asynchronous call `foo!!foo()` creates an unresolved flow, we call it f_1. It also sends a message to `foo`.
4. The `main` thread enters `get*` and is blocked until f_1 is resolved with a value, *i.e.* it can follow a chain of flows until it reaches a value that is not a flow. The `foo` thread starts processing the call to `foo`.
5. In `foo`, a synchronous call, `this.bar()`, is performed. The current thread starts executing `bar`.
6. In `bar`, a new flow, f_2, is created as a result of `this!!baz()`. The current object sends a message to itself. *The call to `bar` returns f_2.*
7. **Back in `foo`**, `get*` is used on f_2. This effectively deadlocks the actor as the asynchronous call to `foo` is still being processed, and can only progress once f_2 is resolved. However, as `foo` cannot progress, the actor never processes the message that resolves f_2.

Unlike in the previous scenario where we used the flexible typing rule, here the synchronization is explicit: `foo` uses `get*` on a flow. We argue our typing rule is safer as it makes the synchronization explicit, which makes deadlocks easier to investigate.

Our theoretical results of Section 3.2 are based on the rule of Figure 3.4 as it is the safest solution. In our implementation we chose to use the flexible typing of Godot to keep in line with what already existed in Encore. However, if language designers were to

introduce flows in a language, we would recommend using the strict typing rule.

3.3.4 Encoding `Fut[]` from `Flow[]`

In this section, we build control-flow explicit futures on top of the dataflow explicit futures we implemented in Encore. As our dataflow explicit futures are based on `DeF`, this asserts the backward compatibility of `DeF` with existing systems. We show that a language implementing only `Flow[]` can build `Fut[]` as a library. In Encore, a library cannot extend the syntax of the language, or add introspection features, we are thus limited to simple encodings.

We provide an implementation of `Fut[]` that relies on our `Flow[]` construct of Encore and on our implementation of `get*` and `async*`.

Definition Godot [1] suggests a construct for such control-flow explicit futures:

$$\texttt{Fut[]}\tau ::= \Box\,\texttt{Flow[]}\tau$$

$\Box$ is called the "box" operator. It encapsulates its argument in a structure of a different type, whose only available operation is `unbox`, where $\texttt{unbox}(\Box\ x) = x$. Intuitively, the $\Box$ operator stops type collapsing: $\downarrow \texttt{Flow}[\texttt{Flow}[T]]$ reduces to $\downarrow \texttt{Flow}[T]$, but $\downarrow \texttt{Flow}[\Box\texttt{Flow}[T]]$ is not collapsed (it reduces to $\texttt{Flow}[\Box\texttt{Flow}[T]]$).

The corresponding operation follows:

$$\texttt{get}\ e ::= \texttt{get*}\ (\texttt{unbox}\ e)$$

Implementation We define the class `Future[t]` and the `get_`[10] function as shown in Listing 3.14.

Listing 3.14: `Future[t]` and `get_` based on `Flow` and `get*`

```
1  read class Future[t]
2    val content: Flow[t]
3      def init(x: Flow[t]): unit
4      this.content = x
5    end
6  end
7
8  fun get_(y: Future[t]): t
9    return get*(y.content)
10 end
```

A restriction of our approach is that, in an Encore library, we cannot overload existing operators or functions. As such, we provide functions with the right behavior, but not the desired name: `get_`, `call_`, `await_`, or `async_`. However, even that is not sufficient. For instance, the method call operator (`!`) cannot be implemented with a mere function as it would require modifying the lexer of the language. Instead, we define the `call_` function

[10] Underscore appended to avoid name collision with the `get` reserved keyword.

that takes a `Flow[t]` as parameter and wraps it into a `Future[t]`. The expression `a!f()` is translated as `call_(a!!f())`. Similarly we define `forward_` to implement `forward`.

Using a simple preprocessor to generate the calls to our library from a syntax based on standard operators on control-flow futures would also be possible as the problem is only about syntactic sugar.

Summary With this implementation of control-flow explicit futures on top of data-flow explicit ones, we showed that control-flow explicit futures can be provided as an extension of `DeF`, and as a library. It is interesting to note that Godot provides an implementation in Scala of data-flow synchronization on top of control-flow explicit futures (the opposite of what we did here), but that implementation does not fully support parametric types, and extending it to handle parametric types seems challenging. Consequently, we believe that implementing `DeF` directly in the compiler is more flexible, and thus we advise, in the design of future programming languages with explicit futures, to first implement a data-flow synchronization, and then extend it with control-flow explicit futures.

3.3.5 Benchmarks

In this section, we provide a performance analysis of different implementations of futures[11]: the builtin control-flow explicit futures `Fut` of Encore, the data-flow explicit futures `DeF` that we introduced earlier, denoted as `Flow`, and the control-flow explicit futures that were built in the previous subsection, denoted as `Fut on Flow`.

We analyze several programs, using chains of futures of different length, or different memory management patterns. Our focus is to analyze the performance of futures, not the efficiency of the parallelization in Encore or the number of actors that can be created in the Encore system. This is why we do not compare ourselves with other frameworks, and why our programs are mostly sequential or featuring little parallelism. Reducing parallelism allows us to avoid the complex interactions that exist between additional objects, concurrent garbage collection, and memory contention: we only study the direct impact of different forms of futures.

All the benchmark results provided here are done on the same Dell XPS 13 9370, with a 8-core Intel Core i7-8550U and 16GiB of memory, running Ubuntu 20.04 and clang v10.0. Encore at version `ea5736869d2ac34cfbeee2be4a2988c819a215af` is run using the release configuration and the `-O3` flag.

3.3.5.1 Test Programs

We evaluate performance on four examples, the first uses non-nested futures, the next two stress the implementation with long chains of futures, and the last one is closer to real-world workloads.

`Word Counter` example. The Encore compiler is shipped with tests, including a word counter, from which we adapted this example. This test dispatches asynchronous

[11] All the code for those benchmarks is available at: `https://gitlab.inria.fr/datafut/fut-on-flow`.

Table 3.3: Running time of `WordCounter` (100 runs). The `OneWay` line corresponds to the version where futures are optimized out by the compiler.

Future used	Average running time (ms)	Std deviation	
`OneWay`	103.4 ms	4.6 ms	4.4%
`fun`	119.3 ms	2.4 ms	2.0%
`Flow`	184.6 ms	6.7 ms	3.6%
`Fut`	185.5 ms	6.5 ms	3.5%
`Fut on Flow`	190.4 ms	7.6 ms	4.0%

Table 3.4: Running time of the Ackermann benchmark (arguments: 3, 4 – 100 iterations)

Future used	Average running time	Std deviation	
`Fut` with `forward`	27.86 ms	1.2 ms	4.3%
`Fut on Flow` with `forward`	28.10 ms	1.0 ms	3.5%
`Flow` with `forward*`	31.49 ms	0.7 ms	2.2%
`Flow`	39.13 ms	1.0 ms	2.4%
`Fut`	44.63 ms	0.7 ms	1.7%
`Fut on Flow`	46.71 ms	0.8 ms	1.6%

hash table insertions to 32 actors. We re-implemented the standard library module `Big.HashTable.Supervisorh` with `Flow`, with `Fut on flow` and with sequential execution (referred to as `fun`). We do not use any form of `forward`, because there is no chain of futures. In the original version of the test shipped with the Encore compiler, the program was written in such a way that the result of the asynchronous calls was never needed. When the compiler identifies this, it removes the creation of the future resulting from the asynchronous call, and the callee does not resolve any future, which improves performance compared to keeping and resolving a useless future. In order for our comparison to be relevant, we rewrote the test in such a way that the futures resulting from the asynchronous calls are actually used. The results are displayed on the `Fut` line of Table 3.3. Additionally, in the interest of comparison, we also ran the original test that does not create futures in order to compare performance. The results for this version are displayed on the `OneWay` line.

Recursive calls on a single actor: Ackermann. We implemented the Ackermann function with recursive calls on a single actor. This program performs successive asynchronous invocations on the same entity (no parallelism, long delegation chains). In this benchmark, the chains of futures are of various lengths, because each call makes two recursive calls, one which is forwarded and one for which it waits. Because we only have one actor, this wait cannot be a synchronization, otherwise it would block the execution. We use here a cooperative yield (`await`) on the future before calling `get`. This problem does not arise with `forward`. Results are presented in Table 3.4.

Recursive calls on many actors without actor creation: recursive list

Table 3.5: Average running time of the recursive list summation (10000 actors, 100 iterations)

Future used	Average running time	Std deviation	
`Fut` with `forward`	16.03 ms	1.0 ms	6.4%
`Flow` with `forward*`	16.14 ms	0.9 ms	5.8%
`Fut on Flow` with `forward`	16.30 ms	1.0 ms	6.3%
`Flow`	26.02 ms	1.3 ms	5.1%
`Fut`	46.87 ms	3.9 ms	8.4%
`Fut on Flow`	47.64 ms	3.8 ms	8.0%

Table 3.6: Running time of 10 k-means steps (100 clusters, 10000 observations, 100 iterations)

Future used	Average running time	Std deviation	
`Flow`	468.8 ms	114.7 ms	24.5%
`Fut on Flow`	494.9 ms	125.8 ms	25.4%
`Fut`	500.7 ms	66.8 ms	13.3%

summation. We compute the sum of the numbers from 1 to n by creating a linked list of actors, each of them holding one of these numbers, and then traversing the list with successive asynchronous recursive invocations (one actor calls the sum function on the next actor and so forth). The measured time does not take into account list creation. There is here one long chain of futures of length n. When `forward` is used, as one future is delegated along the chain of futures, this sum is done in constant space. Results are presented in Table 3.5.

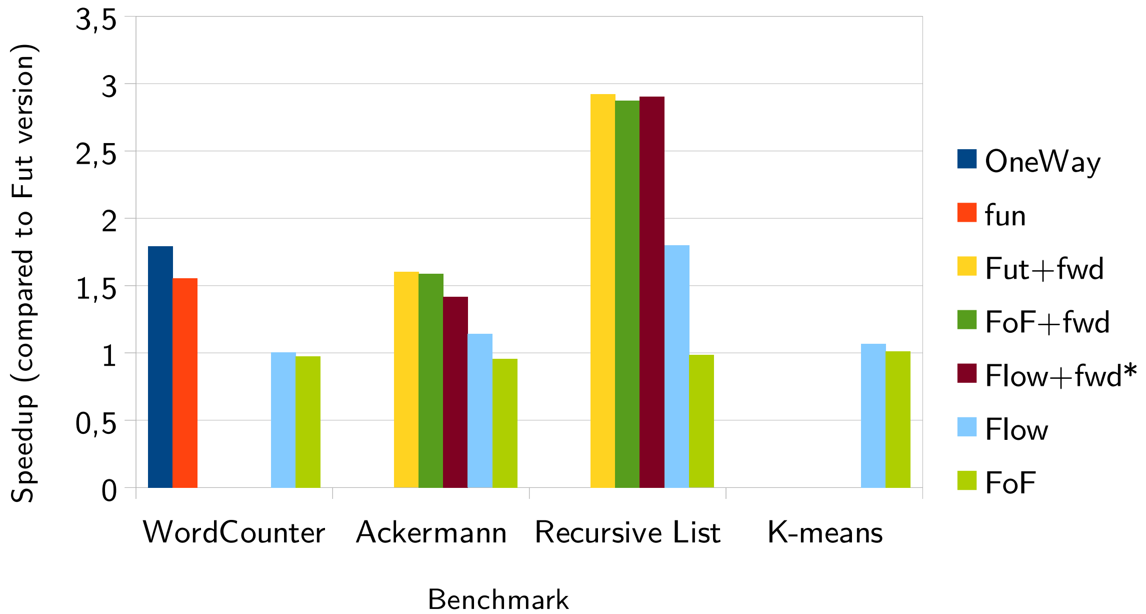
K-Means This benchmark runs a fixed number of iterations of the k-means clustering algorithm. Every compute intensive step is delegated to a large number of actors, then synchronized. This does not use `forward` since there are no chained futures. However, the benchmark is interesting because it allows us to compare the performance of Encore's native control-flow explicit futures and our re-implementation of it on top of data-flow explicit ones, on a realistic workload. Results are presented in Table 3.6.

3.3.5.2 Results and Discussion

The performance of each of the four benchmarks are shown in tables 3.3, 3.4, 3.5, and 3.6. Additionally, Figure 3.8 presents these results as an histogram.

The `Ackermann` example has a lower standard deviation than the other benchmarks, which might be caused by smaller future chains than the list summation example, or the absence of parallelism, unlike the `WordCounter` example. More surprisingly, `Fut on Flow` with `forward` seems faster than `Flow` with `forward*`, although `Fut on Flow` is a layer over `Flow` and calls `forward*`. This might be explained by the additional box objects that `Fut on Flow` introduces in the future chains, which might be beneficial to

Figure 3.8: Results of the DeF and DeF+F benchmarks as histograms



the Encore scheduler or tied to optimization performed by the Encore or the C compiler.

The `OneWay` version of the `WordCounter` example is unsurprisingly far faster than the others. It allows evaluating the overhead of the future synchronization over a plain one-way message.

The Ackermann and recursive list summation examples show the performance benefit of using `forward` or `forward*` over the plain `Fut` or `Flow` equivalent when there are delegated computations.

On all examples, `Fut` and `Fut on Flow` perform similarly, showing that a library-based implementation of `Fut` based on data-flow explicit futures reaches performance similar to a dedicated implementation within the compiler. There is a small overhead that can be explained by the additional "box" objects introduced by `Fut on Flow`.

Both Ackerman and the recursive list summation show the performance benefit of using `Flow` over `Fut`, since less synchronization is needed (for Ackermann, using `Flow` allowed us to directly return a `Flow` without the need for a costly `await`). This difference disappears when using `forward` on both versions since no synchronization is needed whatever the type of future.

The k-means benchmark models real-world workloads and has a larger standard deviation than the others. However the observed performance still show that adding data-flow explicit futures does not affect significantly the performance of futures.

In the `Fut` version of the benchmarks, `forward` explicitly changes the semantics to relax synchronization. As shown in Section 3.2, in the `Flow` version, it becomes an

optimization that does not change semantics. A clever compiler could have compiled the `Flow` version to `Flow with forward*` with the same semantics, and these benchmarks show that this is an interesting optimization. An implementation with `DeF` and implicit optimization of `return` into `forward*` (for long chains) would ally both the performance improvements of `forward` and the improved type handling of `DeF`.

3.4 Summary

This chapter presented our work on dataflow explicit futures, a safe and efficient synchronization tool. Our work further refined the safety aspects of dataflow explicit futures by providing a safer typing rule for the `forward*` operation. We also investigated more optimization by providing a proof of equivalence between the `return` and `forward*` operations on dataflow explicit futures, which opens the way for more optimization done by compilers. Additionally, we provided the first complete implementation of dataflow explicit futures in a production language, the Encore programming language.

In this chapter, we explored static, compilation driven, approaches to improve synchronization's efficiency and safety. In the next chapter, we explore library-driven approaches.

Chapter 4

Flexible Synchronization in Data Exchange – From Static to Automated Configuration

Introduction In the previous chapter we focused on futures and on the way they are synchronized. Our approach relies on static typing rules and a compilation-driven approach. In this chapter we explore the exchange of arrays and/or streams of data between tasks through the use of FIFOs and/or promises. In particular, we ask the question: how to create safe and efficient synchronization tools to exchange arrays and/or stream data?

Synchronization is a costly operation, as discussed in Chapter 2. Intuitively, too many synchronization operations (fine-grained synchronization) will result in CPU cores spending extra time synchronizing; not enough synchronization (coarse-grained synchronization) will result in CPU cores spending extra time idle. Our intuition is as follows: there exists an optimal, or at least "good", granularity of synchronization. In this chapter, we present two tools that allow programmers to specify / find a good granularity of synchronization. The first, PromisePlus, is presented in Section 4.1: a promise that abstracts the concept of array and allows programmers to synchronize on slices of a configurable size of the array with an interface similar to promises and to arrays. The second, FIFOPlus, is presented in Section 4.2: a FIFO that performs synchronization on a number of items rather than on a single one, and that embeds an online performance analysis algorithm that is able to perform an automatic configuration of the amount of items on which synchronization should be performed.

4.1 PromisePlus

We begin with a motivating example in Section 4.1.1 that illustrates the need for a synchronization tool based on promises that works efficiently on arrays. Section 4.1.2 presents this tool, called PromisePlus, that abstracts both an array and a promise, and

performs synchronization on slices of the array. Section 4.1.3 presents benchmarks results that show the efficiency of PromisePlus.

4.1.1 Motivation

As shown in Chapter 2, HPC applications often perform heavy operations on huge amounts of data stored in arrays, or matrices[1]. Many programming libraries based on arrays can be found in the literature, for example algorithmic skeletons [27], or to perform linear algebra operations like the BLAS library [71].

Data streaming allows multiple modules to work in parallel, with one module consuming data produced by another (or many others) modules. Data streaming comes with an inherent synchronization mechanism that ensures that a module (or kernel) cannot start working on data that has not been produced yet. However this synchronization is a costly operation, therefore it needs to happen at the right moment in order to fully exploit parallelism without wasting too much time.

A recurring observation across the tools presented in Chapter 2 is that programmers tend to use a low-level framework/tool as a base building block to create a higher-level API around this tool. The actor model can use MPI as its communication component and abstract away most, if not all, of MPI. The active-objects model further refines the actor model through its programming to interface discipline, and its usage of future prevents the apparition of data races. As a general rule, the efficiency of an abstraction is limited by the efficiency of its own building blocks. As a consequence, new building blocks should be made as efficient as possible.

Programmers may also want an easy-to-use tool that does not come with an entire framework: MPI is a great tool when it comes to communication, but the socket API can sometimes be a more straightforward solution. We think the same holds true of synchronization tools: they could be part of a framework to improve the efficiency of synchronization, but they could also be used as-is in an application that needs efficient synchronization.

To further motivate our approach, let us look at a toy program that composes a map and a reduce operation. Listing 4.1 presents a sketch of what we would need in order to implement a parallel version, here in C++. Note that splitting two map operations between two different threads is not the most efficient in the general case, but in some scenarios there is a need for different threads (e.g. different localisation or architecture, internal state of the map and reduce operations, . . .).

In this toy example, the `map1` function multiplies its input by 2, and the `array_sum` function sums the values in its input array. Both functions are called in parallel, and run in two separate threads. Synchronizations are performed through an array of atomic booleans that is shared by both threads, this avoids the cost of locking many different mutexes, while also bringing safety.

In this example the synchronization is extremely fine grained: a synchronization is performed for every value in the communication array. A coarse-grained variant of this

[1]For simplicity, unless noted otherwise, we will use “array“ to refer to both arrays and matrices

Listing 4.1: Sketch of the parallel composition of two maps together

```
void map1(int arr[], int N, int out[], atomic<bool> out_ready[]) {
  for (int i = 0; i < N; ++i) {
    out[i] = arr[i] * 2;
    out_ready[i].store(true, memory_order_release);
  }
}

int array_sum(int arr[], int N, atomic<bool> in_ready[]) {
  int s = 0;
  for (int i = 0; i < N; ++i) {
    while (!in_ready[i].load(memory_order_acquire))
      ;

    s += arr[i];
  }

  return s;
}

int main() {
  int N = 100;
  int arr[N] = { ... };
  atomic<bool> flags[N] = { false };
  int tmp[N] = { 0 };
  int res[N] = { 0 };

  thread th1(map1, arr, N, tmp, flags);
  thread th2(array_sum, tmp, N, flags);

  // ...

  return 0;
}
```

solution would be to replace the array of booleans with a single boolean that indicates to `array_sum` when the call to `map1` has processed every value in its input array and written all values in the output array. Our intuition is that, in general, none of these two solutions yield good performance: the overhead of synchronization is too high in the first version, and the second version is sequential. Instead, an intermediate solution, where synchronization is performed on slices of the communication array, should yield better performance than both naive solutions.

As discussed in Chapter 2, a synchronization tool should strive for simplicity, in addition to efficiency and safety, in particular it should feature a well-designed API. If we look at Listing 4.1, we recognize the communication pattern of promises: storing true in `out_ready[i]` after storing a value in `out[i]` is equivalent to resolving a promise with the value stored in `out[i]`. An array of promises would nicely abstract the array of boolean and the communication array under a known tool. Our second alternative, where we use a single boolean, would be equivalent to using a promise of array. Setting the boolean to true after computing the entire output array would be equivalent to resolving such a promise with said array. We believe promises are quite adapted to problems

involving the communication of arrays between tasks: unlike futures they do not tie their resolution to a specific instruction, which lets programmers write code extremely similar to a version without promises; unlike FIFOs, multiple readers may access the same promise at the same time as well as multiple times. Promises also have qualities of their own, namely some safety: as we saw before, the `get` operation on a promise never yields an undefined value, although unlike futures it may never yield a value at all, even in a program without infinite loops.

We introduce PromisePlus, a synchronization tool that works as a trade-off between an array of promises and a promise of array. It allows programmers to specify the granularity of synchronization and to stream data between tasks. Unlike usual streaming frameworks like OpenStream [52, 53], or the destructive streaming futures of ABS [59, 60], a PromisePlus acts as a non-destructive stream, with support for out-of-order access of its elements.

The core feature of PromisePlus is its configurable granularity of synchronization, denoted S. When $S = 1$, PromisePlus behaves like an array of promises. Conversely, a maximal granularity ($S \to +\infty$) makes PromisePlus behave like a promise of array.

In section 4.1.3 we show that specifying a fine-grained granularity of synchronization yields the same performance as using an array of promises. Similarly, specifying a coarse-grained granularity of synchronization yields the same performance as using a promise of array. PromisePlus comes with two guarantees: 1) A request to an element of the PromisePlus will unblock after at most S values have been produced and 2) Similarly to “usual” promises, a request for an element will never produce an undefined value, although it may never produce a value at all.

This motivates us to propose a new synchronization tool for streaming arrays between tasks, PromisePlus, that meets the following criteria:

- It makes data dependencies explicit: the programmer explicitly writes which data is shared between threads. PromisePlus embeds the data over which the synchronization occurs, contrarily to most concurrency frameworks like MPI or OpenMP where data dependencies are less explicit.

- It allows programmers to specify the granularity of the synchronization. This is not original compared to MPI or streaming frameworks, merely a necessity;

- It is not tied to a specific problem or a specific setup, and so it is reusable. PromisePlus can be used to synchronize processes that communicate through MPI, or to synchronize threads handled by OpenMP, or as the underlying communication channel between modules in a streaming framework, or even as-is in an application that performs synchronization over the elements of an array;

- When the granularity is configured to the minimum (resp. maximum) value, it behaves like an array of promises (resp. a promise of array), providing the same guarantees and similar performance; moreover, every request for the i-th value of an array unblocks after an amount of values equals to the granularity has been produced;

- Contrarily to classical streaming solutions it supports the existence of several consumers for the same data, and the access to the produced data in any order. This is similar to ABS non-destructive streaming futures [59];
- It provides a high-level API inspired by the promise API, and features low-level optimizations to improve performance.

4.1.2 Flexible Synchronization for Arrays

Presentation To introduce the principles of PromisePlus, Figure 4.1 presents a sketch of the tool at work. In the rest of this section, we will show the evolution of this Figure as operations are performed on the PromisePlus.

A PromisePlus holds an array that is initially empty. This array is sliced into contiguous chunks of a fixed width. In Figure 4.1 the elements of the array are represented by the black squares, and chunks are as a group of connected cells. The numbers above the cells are the index of each value. Each chunk has the same width (all chunks contain the same amount of cells), which is referred to as the `step`. In this example, the step is 2. Red bordered chunks are incomplete chunks, they hold at least one cell without a value: performing a `get` on them will block. Green bordered chunks are complete chunks, all their cells hold a value: performing a `get` on them will immediately return a value. Cells in a green chunk are accessible, while cells in a red chunk are not accessible. In particular, cell 6 was already assigned the value 13 by the producer, but this value was purposely not yet made available to the consumer, it will only be available when enough additional values will be produced.

In order to retrieve the value at the i-th cell of the array (0-based array), the programmer can use the `get` operation. Unlike with the usual promises, the programmer must give a parameter to `get`: the index of the cell. The behavior of `get` is as follows: consider the chunk where the requested value is stored, if all the cells inside this chunk hold a value, then `get` returns the value at the requested position. Otherwise, `get` waits until all the cells in the chunk hold the value before returning the value at the requested position. For instance, in Figure 4.1 a call to `get(1)` would immediately return the value 78. A call to `get(6)` would not return the value 13 because the chunk in which the value is stored still has some empty cells. In particular, a call to `get` on an index with no associated value will wait.

In order to store a value at a given index, the programmer use the `set` operation. Unlike with the usual promises, the programmer must give an extra parameter to `set`: the index at which the value must be written. Performing a call to `set(i, value)` will

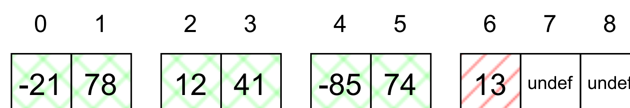


Figure 4.1: Idea of PromisePlus at work

not necessarily unblock a call to `get(i)`. After a call to `set(7, 12)`, the PromisePlus shown in Figure 4.1 will look like the PromisePlus shown in Figure 4.2. The previously red chunk holding values at indices 6 and 7 is now green as all its cells hold a value. A call to `get(6)` that would have previously blocked will now return the value 13.

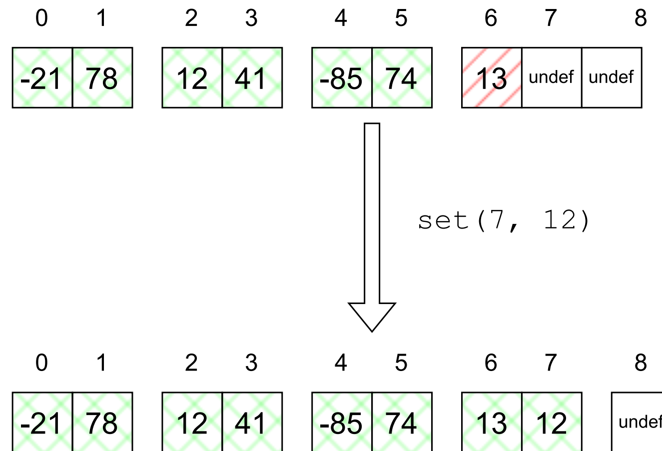


Figure 4.2: PromisePlus from Figure 4.1 after `set(7, 12)`

This behavior for `set` is not sufficient as it could lead to calls to `get` that never return a value. For instance, if a PromisePlus holds an array of size 9, and uses a step of 2 (chunks are composed of two cells), then the cell at the 8-th position will always be in an incomplete chunk (the chunk will only hold one cell rather than two). For optimization reasons detailed below, rather than having `get` check whether or not a chunk is incomplete, we provide a `set_immediate` primitive. Just like `set`, `set_immediate` writes a value at a given position in the array. Unlike `set`, it unblocks all the calls to `get` that are waiting on a lower index, regardless of whether the associated chunks are full. Furthermore, `set_immediate` changes the boundaries of all future chunks: regardless of the original slicing, following a `set_immediate(i, x)` the next chunk begins at index `i + 1`. Without this, a `set_immediate` would create a chunk that is smaller than the configured step. Figure 4.3 illustrates this difference in behavior between `set` and `set_immediate`. It shows the result of calling `set(8, 14)` (left arrow) and `set_immediate(8, 14)` (right arrow) on the PromisePlus of Figure 4.2. The call to `set` would result in an incomplete chunk, so calls to `get(8)` would never unblock. The call to `set_immediate` solves this problem by creating a smaller, but complete, chunk.

Synchronization mechanism PromisePlus revolves around two integer values: the *step*, which we denote S, and a value called *last*, which we denote L. As indicated in the presentation of PromisePlus, the step refers to the width of the chunks, in other words the granularity of the synchronization. *last* represents the highest index in the array that can be passed as parameter to `get` and not induce a wait. For instance, in Figure 4.1, $S = 2$ and $L = 5$. In Figure 4.2, $S = 2$ and $L = 7$.

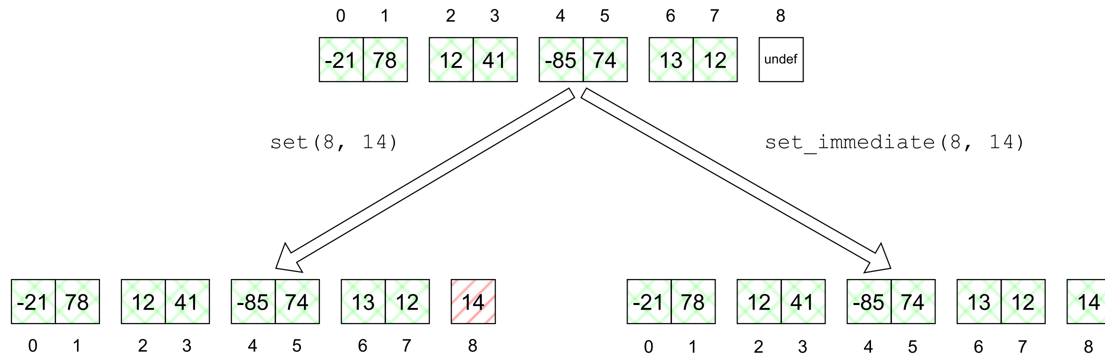


Figure 4.3: Comparison of the effects of `set` and `set_immediate` on a PromisePlus

S is configured by the programmer once, when the PromisePlus is created. It cannot change as the program runs. This is a limitation by choice, and in Section 4.2 we will investigate an analytical method that can be used to reconfigure the granularity of synchronization of a synchronization tool inspired by PromisePlus. The value of L is purely internal, the programmer cannot read it, nor change its value directly. It is initialized to -1[2] when the PromisePlus is created, denoting an empty array. Modifications of the value of L occur through the `set` and `set_immediate` methods. During a call to `set(I, value)`, if $I - L \geqslant S$, then $L \leftarrow I$. During a call to `set_immediate(I, value)`, $L \leftarrow I$ unconditionally.

API We present the PromisePlus API in a more formal way here. `T` denotes the type of the elements stored in the array.

The API exposes four functions: a constructor, `get`, `set`, and `set_immediate`.

The **constructor**, with signature `int` → `int` → `PromisePlus` is used to create a PromisePlus with a given size (denoted N) and a given granularity of synchronization, its step (as usual, denoted S).

The `get` function, with signature `int` → `T` gets the value at a given index. If the value of L is less than the requested index, the call to `get` blocks until the value of L is greater or equal to the requested index, at which point `get` returns the value associated with the requested index. Otherwise, if the value of L is already greater or equal to the requested index, the call immediately returns the value associated with this index.

The `set` function, with signature `int` → `T` → `void` writes a value at a given index. If the difference between the specified index and L is greater or equal to S, then L is updated to the specified index, and pending calls to `get` may now return if the value of L allows them to.

The `set_immediate` function, with signature `int` → `T` → `void` writes a value at a given index and forcibly resolves synchronizations on every lower index. The value of L is updated to the specified index regardless of the distance between the specified index

[2]As we will see in the *API* paragraph, L can be initialized to other values than -1.

and L. This function is useful when it comes to writing the very last value in the array, as it may be located in an incomplete chunk. It can also be used to shift synchronization windows if the programmer desires so.

Generalization It is possible to use other types than `int` to index the array. Let `Index` denote the type used to index the array. In order for such a type to be used as an index, the programmer must offer two operations called `null` and `successor`. `null` must return a value of type `Index` that denotes the element before the first element of the array. `successor`, given an `Index`, must return another `Index` that represents the index of the next element in the array. There must be a total order on the values of the `Index` type. Naturally, the signatures of `get`, `set` and `set_immediate` need to be updated accordingly to use an `Index` as parameter rather than an `int`.

When using integers to index the array, the `null` operation returns -1, and the `successor(x)` operation returns $x + 1$.

Constraints Calls to `set` and `set_immediate` must take consecutive indices as parameter. Given I the index of a call to either `set` or `set_immediate`, the next call to `set` or `set_immediate` is correct if and only if its index $I' = successor\ (I)$. As a consequence PromisePlus works at its best in a single producer environment. If multiple producers were to share a PromisePlus, they would need to come up with an extra synchronization tool that would properly order the calls to `set` and `set_immediate`. Naturally, these extra synchronizations may reduce performance.

Behaviour Algorithms for `get`, `set` and `set_immediate` are provided in Algorithm 1, Algorithm 2, and Algorithm 3 respectively. All these algorithms refer to a value W, called the *local index*, which we will present in the *Optimization* paragraph.

Algorithm 1 Algorithm for `get`

1: **function** GET(*index*) ▷ Get value associated with index
2:     **while** index > W **do** ▷ The local index avoids a cost-heavy read of L if it is greater than the requested index
3:         W ← L
4:     **end while**
5:     **return** value associated with index
6: **end function**

Algorithm 2 Algorithm for `set`

1: **procedure** SET(*index*, *value*) ▻ Associate value with index. If enough calls have been made, unblock calls to `get` with a lower index
Require: index is the next integer compared to the last call to `set`
2: Associate value with index
3: **if** (index - local_last) ⩾ step **then** ▻ Reading from a local copy of *last* in the producer thread avoids cost-heavy accesses to L
4: local_last ← index
5: L ← index ▻ Unblock `get(i)`, `i` ⩽ `index`
6: **end if**
7: **end procedure**

Algorithm 3 Algorithm for `set_immediate`

1: **procedure** SET_IMMEDIATE(*index*, *value*) ▻ Associate value with index. Unconditionally unblock all calls to `get` with a lower index
Require: index is the next integer compared to the last call to `set`
2: Associate value with index
3: L ← index ▻ Unblock `get(i)`, `i` ⩽ `index`
4: local_last ← index
5: **end procedure**

Optimizations Since PromisePlus behaves as a promise, synchronization is inevitable between read operations (`get`) and write operations (`set`, `set_immediate`). All these operations manipulate the L variable, therefore its access must be protected to avoid data races. A naive approach would be to use a mutex and a condition variable, however as discussed extensively in Section 2.1.2 higher performance can be achieved by using atomics, which sacrifices the energy efficiency of the passive wait on a condition variable with a busy wait on an atomic. Due to the compute intensive nature of HPC kernels, and the fact that these kernels usually run alone on a machine, busy waiting is, in our opinion, preferable, as it improves the reactivity of the application. Therefore, we chose to make L an atomic variable, with an acquire-release ordering. We were inspired by the design of WeakRB in [72] that uses atomics in a similar way to implement a safe and efficient SPSC FIFO.

Additionally, in order to alleviate the amount of access performed on this atomic variable, each consumer thread manipulating the PromisePlus is given a thread-local weak index W. W is a pessimistic view on L: it can be lower or equal to the actual value of L, but it can never be higher than L. It represents the highest index this thread knows will not cause `get` to block. W is read during calls to `get(I)`: if $I \leqslant W$, then the thread will not read the value of L. If $I > W$, then a synchronization is performed to read the value of L, and subsequently update W, until the `get` can be unblocked.

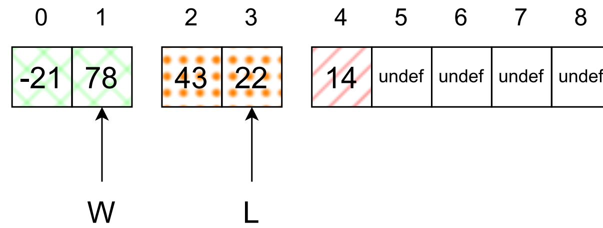


Figure 4.4: Effect of the different indices on the whether `get` will wait or not

Examples Figure 4.4 illustrates how the weak index W and the *last* index L work on a PromisePlus. This figure present a view a consumer has of a PromisePlus, with a step of 2. The value of L, common to all threads, is 3. For this consumer, its local W is 1. Green cells can be accessed without waiting and without accessing L. Orange cells can be accessed without waiting, but require an access to L. Red cells cannot be accessed without waiting.

Figure 4.5 illustrates the evolution of the weak index as parallel calls to `set` are done while `get` is waiting. The figure is divided into three areas: the top one represents the actions of the producer, the bottom one the actions of the consumer, the middle one the two values L and W as time moves forward. The PromisePlus is configured with a step of 2.

P denotes the actions of the producer, **C** the actions of the consumer, **W** is the weak (local) index of the consumer, and **L** the value of *last*, shared by the producer and the consumer. Blue colored calls to `set` are calls that update the value of L. Orange colored calls to `get` are calls that update the value of W.

There are three kinds of arrows on this figure:

- Arrows labeled “A” indicate that a `set` modifies the value of L;
- Arrows labeled “B” indicate that a `set` changed the value of L and the new value of L has been used to resolve a synchronization and update W;
- Arrows labeled “C” indicate that a `get` changed the value of W.

Intuitively, a “B” arrow is always paired with a “C” arrow.

We can observe the following on the figure:

- The call to `get(0)` waits until the call to `set(1, 21)` completes, even though there has been a call to `set(0, 12)`. This is normal: the step is 2, therefore indices 0 and 1 become available to `get` only once they are both filled.
- The call to `set(0, 12)` does not change the value of L, but the call to `set(1, 21)` does. Similar observations can be made for other calls to `set`: the second in a row updates L. This is, again, a consequence of the step. L indicates the highest available index, and by definition of the step, it updates every S calls to `set`.
- The call to `get(1)` is shorter than the calls to `get(0)` and `get(4)`. This is because W is already set to 1, therefore no synchronization is needed.
- The call to `get(1)` does not update the value of W. This is because the requested

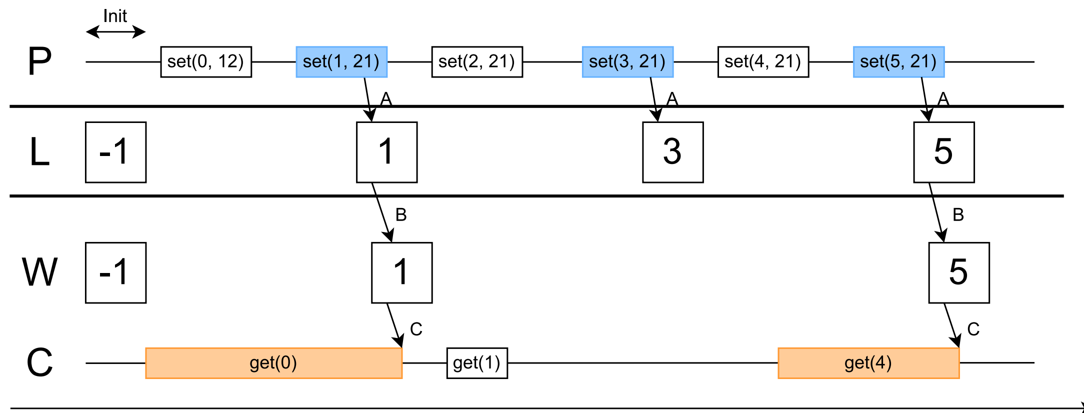


Figure 4.5: Evolution of W and L on a PromisePlus

index was not greater than W, therefore no synchronization was performed, and W is only updated when a synchronization occurs.

- Conversely, the calls to `get(0)` and `get(4)` update the value of W as they request an index greater than W, and therefore synchronize.

Guarantees There are two main guarantees with PromisePlus. The first is that a call to `get` will never produce an undefined value. This guarantee mirrors the behavior of traditional promises: a call to `get` on a traditional promise never produces an undefined value. The second guarantee is that a call to `get(I)` will be unblocked by a call to `set(J, V)` or by a call to `set_immediate(J, V)` with $J \in [I; I + S[$.

Rationale of the index in `set` The necessity to explicitly pass an index parameter to `set` and `set_immediate` can be seen as an argument against the simplicity we strive to achieve. It requires the programmer to keep track of which index was last given to make sure the program remains in a valid state. It would have been possible to write both `set`s functions without specifying the index and simply writing in the next available cell of the array inside the PromisePlus, but that is not what we wanted to do. PromisePlus abstracts both a promise and an array, and the API reflects this. `promise.get(I)` is similar to `array[I]`, `promise.set(I, V)` is similar to `array[I] = V`. In fact, languages with sophisticated type systems could implement `get` and `set` as operators that work on arrays (like the `[]` operator of C) to have an API that completely hides away the “promise” aspect of the tool.

4.1.3 Benchmarks

In this section we evaluate the performance of PromisePlus when it comes to synchronizing two composed compute kernels. We challenge the efficiency of our implementation when compared to an array of promises and a promise of array. First, a PromisePlus with a

step of 1 should perform similarly to an array of promises, and a PromisePlus with a maximal step should perform similarly to a promise of array. Second, there should be an optimal granularity of synchronization at which PromisePlus yields better performance than both an array of promises and a promise of array.

Chosen problem Our problem is inspired by the LU program in the NAS Parallel Benchmarks (NPB) [73]. "Inspired" in that we did not port the Fortran version in the NPB to a C++ version, but instead reproduced the data dependencies of the algorithm to create our benchmark program. Our program operates on a 4D matrix: it iterates over every element of this matrix, excluding boundary values. This program is parallelized at a single level using OpenMP.

The 4D input matrix is split into multiple 3D matrices. Each of these 3D matrices is given to a thread that performs some CPU heavy computation on it. Due to the nature of the computation, a data dependency will arise between threads, which will require a synchronization. Figure 4.6 presents the data dependencies in question, as well as how computations are distributed between threads.

Here, the 4D matrix is split in multiple 3D matrices, distributed between a number of worker threads, represented side by side. Bidirectional arrows at the bottom indicate which thread works on which part of the matrix. To keep the figure readable, these arrows are only represented on the X axis. Thread 1 will compute all the values that have their X coordinate as 1, Thread 2 will compute all values that have their X coordinate as either 2 or 3, Thread 3 will compute all values that have their X coordinate as either 4 or 5 etc.

The color of a point indicates which thread computes the value and is the same as the color of "Thread X" at the bottom of the figure. For instance, point (1, 1, 1) is computed by Thread 1, point (2, 1, 1) is computed by thread 2. Black points represent values that are not written, only read.

Unidirectional arrows represent data dependencies. An arrow going from a point A to a point B represents the idea that the computation of B involves the value of A (the value of B depends on the value of A). Green arrows are dependencies on values computed by the same thread. Red arrows are dependencies on values computed by a different thread. For instance, on Figure 4.6, there is a red arrow between a point labeled A and a point labeled B. A is a point computed by Thread 1, B is a point computed by Thread 2, therefore Thread 2 cannot compute B until Thread 1 has computed A.

In practice, given a non-boundary point at coordinates (X, Y, Z), its computation will involve the value of $(X-1, Y, Z)$, $(X, Y-1, Z)$ and $(X, Y, Z-1)$.

Synchronization in NAS-LU In NAS-LU, the synchronization is performed using an array of atomic booleans, all initialized to `false`. Thread N sets the $N-th$ atomic boolean to `true` once it had complete its work on its entire chunk of the matrix. Thread $N+1$ can not start working on its chunk of the matrix until the boolean of thread N is set to `true`. Thread 1 does not perform any synchronization to start its work.

Listing 4.2 presents the skeleton of the code as it was adapted in our program. `compute`

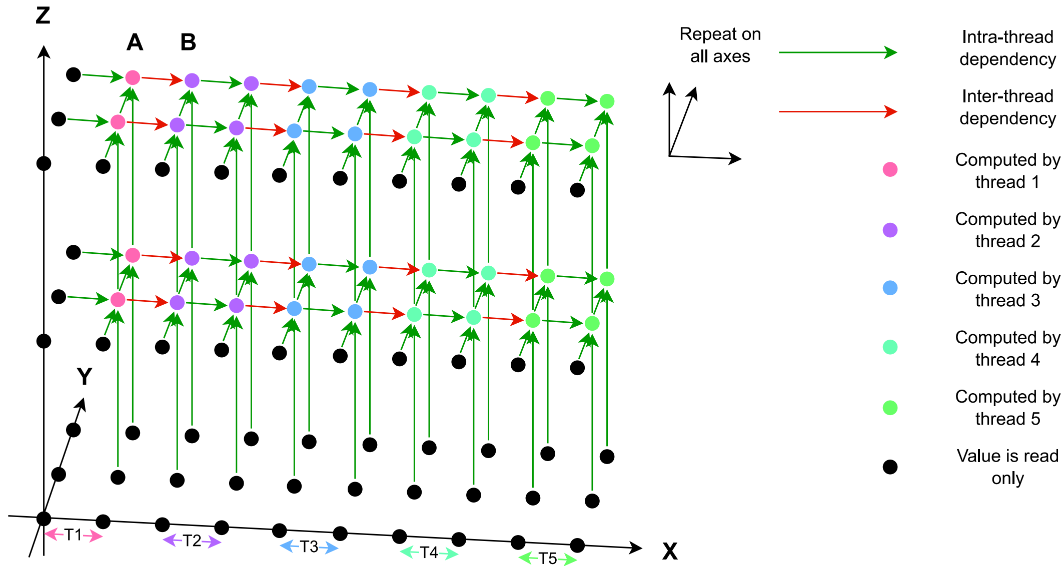


Figure 4.6: Representation of the data dependencies in NAS-LU (matrix chunk)

is the function performing the computation, it takes the work matrix, its dimensions, and the array of atomic booleans for synchronization, as parameters. The booleans are initialized to false[3]. `sync_left` and `sync_right` perform the synchronization. `sync_left`, for any thread that is not the first thread, waits until the boolean of the "previous" thread becomes true and then unblocks. `sync_right` simply sets the boolean of a given thread to `true`, which unblocks the "next" thread. `fn` is the function used to compute the value of a point at position (i, j, k, l) in the matrix. This computation involves the values of the points at positions $(i, j-1, k, l)$, $(i, j, k-1, l)$ and $(i, j, k, l-1)$.

`compute` uses a traditional OpenMP work sharing pattern. For a primer on OpenMP, refer to Section 2.2.3. The splitting of the second loop (over `j`) corresponds to the split into multiple 3D matrices described before. This can be seen as each thread working on a slice of a 3D matrix extracted from the 4D matrix. Synchronization is performed around the loop over the second dimension of the 4D matrix, corresponding to the first dimension of an extracted 3D matrix, which corresponds to what is shown on Figure 4.6. Intuitively, since the second loop is split between multiple threads of execution, the execution of `fn` must be delayed until a value is available at index $j-1$ during the first iteration of any thread (except the first one). This synchronization is rather coarse-grained: because the dependency arises on the use of the second iteration variable, thread N must wait until thread $N-1$ has completed all its iterations on the same 3D matrix before starting.

Our synchronization We rewrote `compute` in the following way: synchronization is now done using our PromisePlus. Listing 4.3 gives an overview of the code.

[3] Memory ordering is relaxed as the parallel region has not been entered yet, therefore there is no need

Listing 4.2: Skeleton of the code of NAS-LU adapted in PromisePlus microbenchmark

```
void sync_left(atomic<bool> sync[N]);
void sync_right(atomic<bool> sync[N]);
void fn(int matrix[][][][], int i, int j, int k, int l);

void compute(int matrix[W][X][Y][Z], int W, int X, int Y, int Z,
    ↪ atomic<bool> sync[N]) {
  for (int i = 0; i < N; ++i)
    sync[i].store(false, std::memory_order_relaxed);

  #pragma omp parallel
  for (int i = 0; i < W; ++i) {
    sync_left(sync);
    #pragma omp for schedule(static) nowait
    for (int j = 1; j < X - 1; ++j) {
      for (int k = 1; k < Y - 1; ++k) {
        for (int l = 1; l < Z - 1; ++l) {
          fn(matrix, i, j, k, l);
        }
      }
    }
    sync_right(sync);
  }
}

void sync_left(atomic<bool> sync[N]) {
  int num = omp_get_thread_num();
  if (num > 0) {
    while (!sync[num - 1].load(std::memory_order_acquire))
      ;
  }
}

void sync_right(atomic<bool> sync[N]) {
  int num = omp_get_thread_num();
  sync[num].store(true, std::memory_order_release);
}
```

Besides the use of PromisePlus, there are two structural changes: the OpenMP `for` directive has been moved to the third loop, and the order of the second and third loop has been flipped. This change comes from the fact that since the original version synchronized around the second loop, the only way to refine the synchronization would be to synchronize around the third or fourth loop. However, since the synchronization comes from the positioning of the `for` directive, it was necessary to move it as well. We

for a strong ordering constraint. Other threads do not even exist yet.

Listing 4.3: PromisePlus microbenchmark

```
void compute(int matrix[W][X][Y][Z], int W, int X, int Y, int Z, PromisePlus<void>
    ↪ promises[N], int N) {
  #pragma omp parallel
  int num = omp_get_thread_num();
  for (int i = 0; i < W; ++i) {
    for (int k = 1; k < Y - 1; ++k) {
      if (num != 0) {
        promises[num].get(k);
      }

      #pragma omp for
      for (int j = 1; j < X - 1; ++j) {
        for (int l = 1; l < Z - 1; ++l) {
          fn(matrix, i, j, k, l);
        }
      }

      if (num != omp_get_total_threads() - 1) {
        promises[num + 1].set(k);
      }
    }

    if (num != omp_get_total_threads() - 1) {
      promises[num + 1].set_immediate(Y - 1);
    }
  }
}
```

chose to move the synchronization around the third loop. The exchange of the second and third loops comes from a desire to have the same amount of synchronization between the PromisePlus version and the NAS-LU version when the PromisePlus is configured with the maximum step.

If the step is 1, the synchronization will occur every time the second loop completes an iteration. If the step is $X - 2$, the maximum possible, the synchronization will occur once the second loop has completed all its iterations, similarly to the original version.

Protocol As our objective is to study the influence of the amount of synchronization, we proceeded this way:

- We define three "shapes", i.e. three sets of dimensions, for our work matrix. We choose the shapes so that the amount of computation would be similar across the different shapes (variation of less than 1%), but the amount of calls to `set` would vary (up to 60% more calls to `set`). Table 4.1 presents the different shapes as well as the variations in ratios between each of them. The choice of the second dimension is the most important part, as it will dictate how much calls to `set` occur. In a matrix of shape $W * X * Y * Z$, X is the size of the array stored inside the PromisePlus, and the maximum index that can be passed to both `get` and `set`. As we need to see the influence of the step, we chose values of X that are not too high, as it makes extensive benchmarking easier.

Table 4.1: Number of synchronizations and computations per synchronization in the PromisePlus microbenchmark depending on the shape of the input matrix

Configuration	Shape ($X * Y$)	Synchronizations	Computations per synchronization
A	161 * 62501	62500	16261
B	126 * 80001	80000	12726
C	101 * 100001	100000	10201

- Next we run our benchmark program. For each matrix shape we generate a random matrix, and compute the result of executing our computation function without parallelism. Next, for every possible value of the step, we run the parallel compute function a hundred times and measure the average execution time. The input matrix is regenerated between every call to make sure CPU caches are cold when the parallel run begins.
- Finally, we wrote three additional versions of `compute`.
  - The first one uses an array of C++ Standard Library promises for synchronization. This will allow us to compare the performance of a PromisePlus with a step 1 with the promises that are most straightforwardly available to a C++ programmer. C++ promises use a passive wait, unlike PromisePlus. In order to make the comparison fairer, we also compare PromisePlus with home-made promises that use busy wait.
  - The second one performs synchronization on an array of home-made promises. These home-made promises use a busy wait to make the comparison with PromisePlus more meaningful. This will allow us to compare the performance of a PromisePlus with a step of 1 with an array of promises.
  - The third one performs synchronization on a home-made promise of array, that uses busy-wait. This will allow us to compare the performance of a PromisePlus with the maximum possible step for the problem at hand.

Benchmarking environement These tests were performed on a machine equipped with four Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz, with 96 threads without hyperthreading. Applications were built using GCC 8.3, C++17, and the $-O2$ flag in Release mode.

Results and analysis Figure 4.7 presents, for each matrix shape, the average time for running the computation function 100 times. The legend indicates which synchronization tool was used. "P[Arr]" is a home-made promise of array, "Arr[P]" is an array of home-made promises, "Arr[SP]" is an array of C++ Standard Library promises, "P+1" is a PromisePlus with a step of 1, "P+max" is a PromisePlus with the highest possible step, and "P+opt" is a PromisePlus with the best step we were able to find. Figure 4.8 presents, for each matrix shape, the average time for running the computation function

100 times when synchronizing with a PromisePlus with different steps (minimal step being 1 for all shapes, maximal step being the third dimension of each shape; using a higher step would give the same result as the maximal step: `set` would trigger no synchronization and all data would become available with the call to `set_immediate`).

We can observe that promises from the C++ Standard Library gives the worst performance: the higher the amount of synchronization, the more their performance gets degraded compared to any other implementation. This comes from the fact that while C++ STL promises are written to work efficiently in many different situations, some of their functionalities could be discarded to achieve better performance in some contexts.

When comparing PromisePlus with a step of 1 with an array of promises, and when comparing PromisePlus with the highest possible step with a promise of array, we see that they yield similar performance. The PromisePlus version performs slightly worse (5%) than the non PromisePlus version; we assume this is due to the additional checks performed inside `get` and `set`, compared to the straightforward implementation in home-made promises. This validates that PromisePlus performs similarly to an array of promises / a promise of array when configured with the extreme steps.

In all three cases, we can observe that there is a step that allows PromisePlus to perform better than any other synchronization pattern. This is more observable on Figure 4.8 where the performance curve has an inverted bell shape, with a minimum. This validates our theory that a synchronization with an intermediate granularity can yield better performance. The performance gain when compared to the second best option, array of home-made promises, can go up to 12.63 %, observed on the second shape, $101 * 126 * 80001 * 2(B)$, with a step of 82.

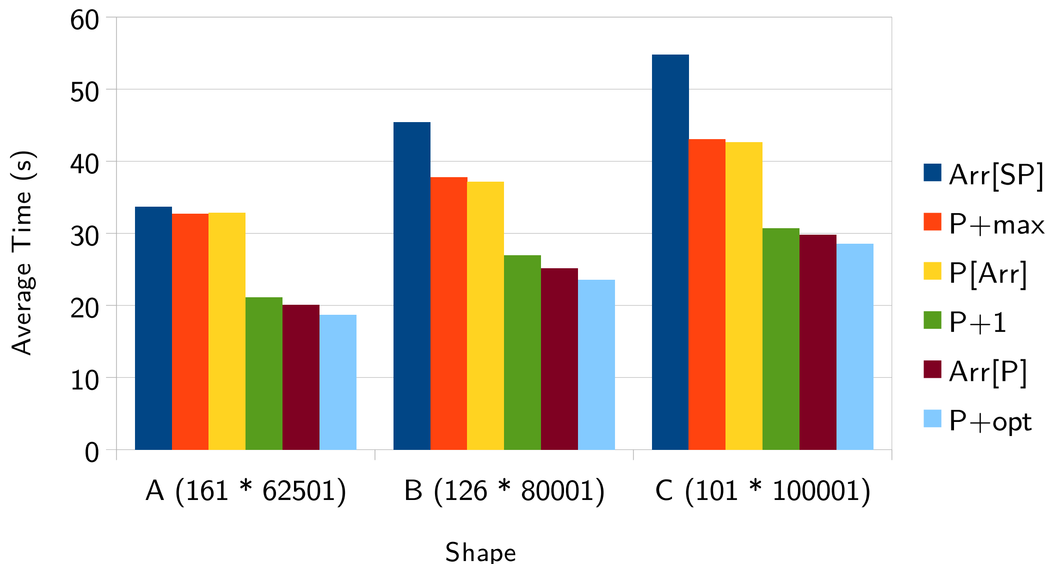


Figure 4.7: Execution time for each pattern on different matrix shapes

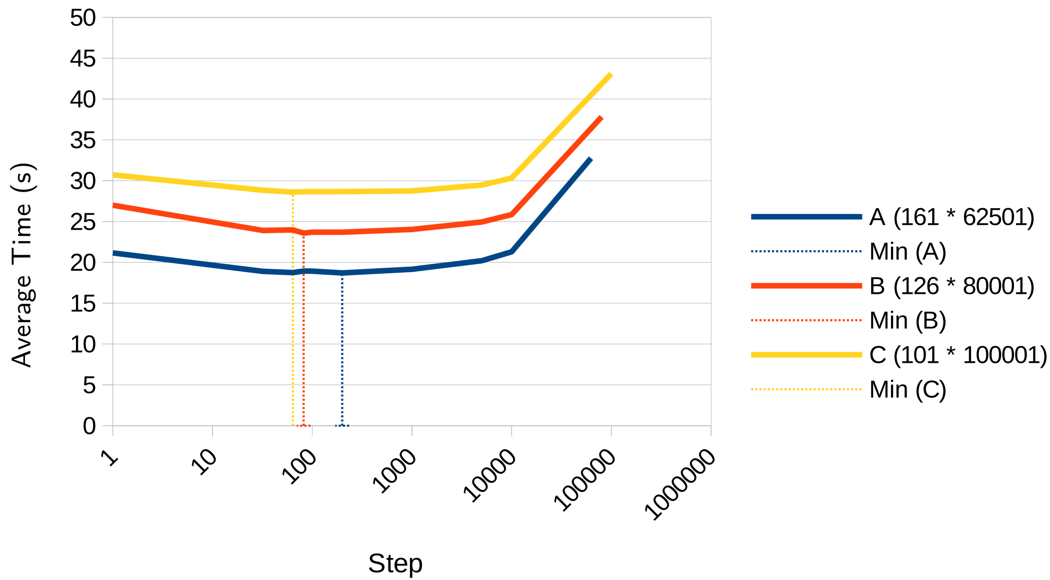


Figure 4.8: Execution time for different matrix shapes and different steps

4.1.4 Summary

In this section we presented PromisePlus, an abstraction over promises that allows parallel computations to synchronize on slices of arrays with a granularity chosen by the programmer. This allows them to express different synchronization patterns with a single tool. Moreover, PromisePlus is not tied to a specific framework, and as such can be used in multiple contexts. PromisePlus also features performance improvements without requiring programmers to extensively refactor their code. Finally, PromisePlus offers the same guarantee as classic promises: a call to `get` never produces an undefined value.

However, PromisePlus could benefit from some form of automation as programmers must select the synchronization step, and their choice may not always be optimal. We now propose FifoPlus, an abstraction over FIFO queues that reuses PromisePlus idea of a configurable granularity of synchronization, and combines it with an online performance analysis algorithm that leads to an automatic selection of the granularity.

4.2 FifoPlus – Automatic Deduction of the Granularity of Synchronization

In the previous section, we showed that the granularity of synchronization can have an impact on the performance of a parallel computation on an array. However, PromisePlus suffers from a major drawback: the granularity of synchronization is configured once, at the time of creation, and never changed after. As we have seen, the average performance

as the step grows evolves according to a bell shape: there is a window of steps that will yield optimal performance. Expecting the programmer to find this window themselves is not acceptable: it would require careful profiling of the application, or multiple runs with different steps to find out which step gives better performance. Indeed, not all applications will need the same step to reach peak performance and some applications are not run multiple times, either because the work they do does not need to be repeated, or because they work on an infinite input stream. This brings the need for an automated way of finding the step.

As we have seen in Chapter 2, the problem of finding the best configuration to efficiently solve a problem is nothing new. SkePU [26] uses an offline machine-learning algorithm [31] to configure its skeletons and get the best performance. The StarPU runtime [9] records scheduling decisions on different tasks in order to better schedule them the next time. Both these approaches have some drawbacks. Machine learning is notoriously opaque and yields results that cannot be explained and we believe programmers often need to understand scheduling and optimization decisions, e.g. to fine-tune the model to better suit their needs. StarPU's approach is a good source of inspiration, as it records the result of a choice and then selects the choice that gave the best result. This approach is easy to understand, and is adaptated to tasks that run multiple times in a single execution of a program, tasks that are part of frequently ran programs or a combination of both. It is however not as well adapted to tasks that run a single time inside a program that runs once.

In this section, we present two things. First, FifoPlus, a communication buffer used to exchange data between multiple threads with a configurable granularity of synchronization. Second, a theoretical model for pipelined applications that can be fed information gathered from an ongoing execution of such an application in order to configure the granularity of synchronization of the FifoPlus involved in the data exchanges in this pipeline.

Section 4.2.1 presents a case study of an algorithm that served as the driving motivation behind the creation of FifoPlus. Section 4.2.2 presents a high-level overview of FifoPlus. Section 4.2.3 presents our analytical performance model. Section 4.2.4 completes the high-level overview of FifoPlus by showing how the analytical performance model is integrated within FifoPlus, and how FifoPlus feeds measured data into the model. Section 4.2.5 presents benchmarks that evaluate the performance of FifoPlus on a microbenchmark and on the algorithm of our case study. Finally, Section 4.2.6 concludes.

4.2.1 Case Study – The PARSEC-Dedup Algorithm

In this section we present the dedup algorithm, a compression algorithm in the PARSEC Benchmark Suite [74]. We identified this algorithm as interesting because it heavily relies on communication between multiple threads, and initial investigation using a profiler showed promising possibilities for optimization.

The dedup algorithm is made of a pipeline of five different stages:

1. The **Fragment** stage, which takes the input to compress and fragments into

coarse-grained chunks; a chunk is basically a part of the file.

2. The **Refine** stage, which takes a coarse-grained chunk produced by `Fragment` and refines it into smaller, fine-grained chunks. Each of these chunks is identified by a pair of numbers.

3. The **Deduplicate** stage, which takes a fine-grained chunk produced by `Refine` and hashes it to see if a chunk with an identical hash has already been encountered. Effectively, this stage checks for duplicates. Depending on whether the received chunk is a duplicate or not, it may be sent to one of two stages:

   (a) If the chunk is a duplicate, it gets sent to the fifth stage, **Reorder**;

   (b) If the chunk is not a duplicate, it gets sent to the fourth state, **Compress**.

4. The **Compress** stage, which takes a fine-grained chunk produced by **Refine**, that has not already been encountered before, and compresses it using a configurable compression algorithm. Once compressed, the resulting chunk is sent to the fifth and final stage, **Reorder**.

5. The **Reorder** stage has two inputs: duplicated non-compressed fine-grained chunks from **Deduplicate**, and compressed chunks from **Compress**. The purpose of this stage is to store chunks until they can be ordered properly. The ordering of chunks is based on their identification numbers computed in the Fragment and Refine stage. Indeed, all the chunks corresponding to the first coarse-grain chunk must be written to the disk before the second coarse-grained chunk can be written.

As the PARSEC Benchmark Suite aims to test the performance of parallel architectures, the algorithm is parallelized as follows:

- Stages **Refine**, **Deduplicate** and **Compress** are replicated a number of times;

- The **Fragment** stage is not replicated, it distributes work to the **Refine**s in a round-robin fashion;

- The **Reorder** stage is not replicated either, and pulls from the different **Compress** and **Deduplicate**s in a round-robin fashion.

In both the sequential and the parallel version, the communication between the different stages is done through FIFOs. Figure 4.9 gives a view of the entire pipeline in the parallel version. Stages **Refine** and **Deduplicate** are replicated two times, stage **Compress** is replicated four times per Deduplicate stage. A rectangle represents a FIFO. A circle represents a thread of execution. Its text indicates which stage of the pipeline is executed by the thread. Arrows ending on a FIFO represent a push from the thread to the FIFO; arrows starting from a FIFO represent a pull initiated by the thread at the end of the arrow. Rhombuses represent the beginning and end of the pipeline.

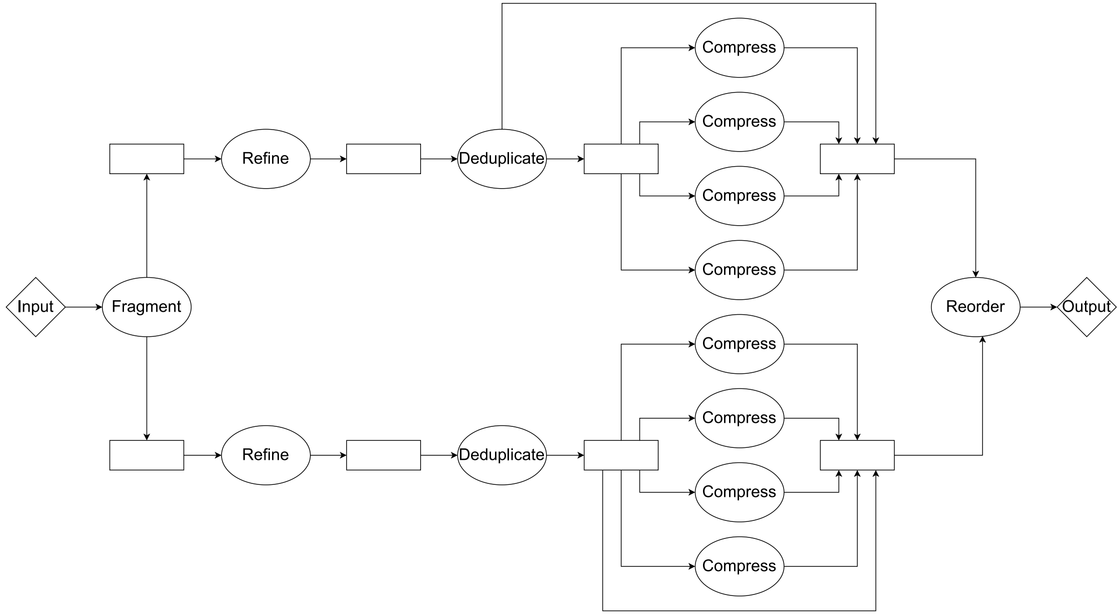


Figure 4.9: Representation of the parallel version of the dedup algorithm

4.2.2 A First Glance at FifoPlus

In this section we give a quick overview of FifoPlus. The purpose of this section is to give enough context so the reader may understand the next section, where we present our theoretical model.

FifoPlus is an abstraction over a Multiple Producers Multiple Consumers (MPMC) queue. It has two key features. The first is similar to PromisePlus: synchronization with a configurable granularity. A request for an element in the FIFO may not be answered immediately even if an element is present in the FIFO. Rather, the FIFO will wait until a given amount of elements is present in the FIFO before answering. The second is the possibility to reconfigure the granularity of synchronization during the execution, rather than having it set once when the FIFO is created. This reconfiguration is coupled with a theoretical model of performance that gives a good approximation of a good granularity of synchronization.

PromisePlus was designed to work in a Single-Producer Multiple-Consumer environment, hence its reliance on busy waiting for reactivity. FifoPlus works in a Multiple-Producer Multiple-Consumer environment, therefore it cannot realistically rely on busy waiting. We chose to use a mutex to ensure safety, at the cost of passive waiting. We considered wait-free and lock-free approaches, however at the time designing efficient lock-free or wait-free MPMC queues was notoriously hard[75]. An example of wait-free MPMC queue comes from FastFlow [55], although their queues require an additional arbiter thread. This thread is used to send data to queues, even if they are not asking for it. In our approach, we remove both the arbiter thread and unrequested data transfer: clients get data upon request. Although we use a mutex, we alleviated contention as

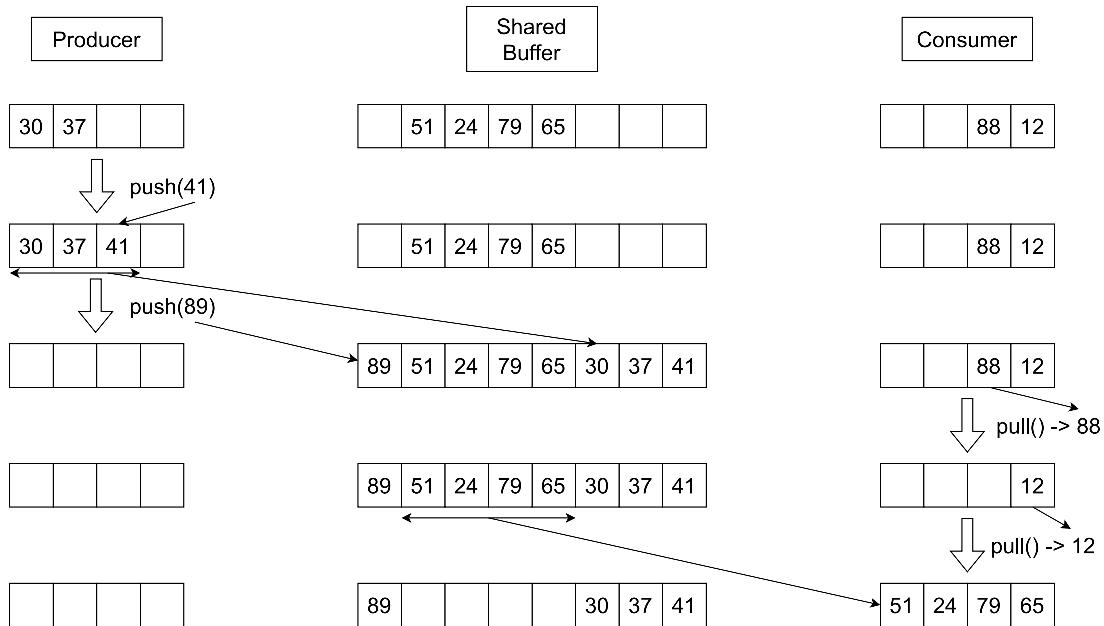


Figure 4.10: Local and shared buffers with FifoPlus

much as possible.

In practice, programmers do not interact directly with an instance of a FifoPlus to store or retrieve elements. Rather, they work on *views*. A *view* is a local buffer that is tied to a FifoPlus. This buffer has a size that is equal to the granularity of synchronization specified on the FifoPlus. A push operation on a view copies data into this local buffer, and only triggers a push into the associated FifoPlus if the view is full. In this case the entire content of the view is pushed into the associated FifoPlus. Similarly, a pull operation only triggers a pull from the associated FifoPlus, when the view is empty. In this case items are pulled from the associated FifoPlus until either the view is full or the FifoPlus is empty. These operations, push and pull on the FifoPlus, are the ones that require mutual exclusion. Our choice here was to have critical sections that last longer than the traditional "Lock the mutex, write a single value in the buffer, unlock the mutex", while reducing the amount of times a thread enters a critical section. The transfer between the local buffer of a view and the shared buffer of the FifoPlus is done using `memcpy`, which is generally implemented using vectorized operations that are more efficient than element-by-element copies. Figure 4.10 illustrates the difference between local and shared buffers on an example execution.

The scenario presented on Figure 4.10 is as follows: a single and a single consumer work in parallel. Communication occurs through a FifoPlus. The producer and the consumer each have a local buffer, and the shared buffer is used for data exchange. The size of each local buffer is 4, so the granularity of synchronization is 4 for both producer and consumer.

When the producer pushes the value 41 into the FifoPlus, it is added into the

producer's local buffer, as there is room in it. When 89 is pushed into the FIFOPlus, it is added into the producer's local buffer as there is still room in it. This completely fills the local buffer, so its content is transfered into the shared buffer.

Similarly, when the consumer pulls from the FIFOPlus, it pulls from its local buffer first as there is data available in it. The first pull yields 88, the second yields 12. The second pull empties the consumer's local buffer, so data is pulled from the shared buffer.

Unlike PromisePlus, the granularity of synchronization is not tied to the FIFOPlus itself. Rather, each view has its own granularity of synchronization, which allows producers and consumers to select a granularity that is more adapted to each of them.

The lifecycle of a FIFOPlus contains three stages: 1) A measurement phase, where the views are configured with a step of 1, and measurements are performed 2) A reconfiguration phase, in which the FIFOPlus computes the theoretical optimal step and reconfigures the views with the found steps and 3) The remainder of the execution, with views reconfigured.

4.2.3 An Analytical Performance Model

In this section we present our analytical model for computing an approximation of the time it would take for a producer-consumer scenario to run. We consider that we have N_p producers and N_c consumers. A total amount of I elements is produced and consumed. Communication between the producers and the consumers is made through the use of a FIFOPlus.

Listing 4.4 presents a minimal working example illustrating the use of FIFOPlus. In the example we have N_c threads running the `consume` function in parallel, and N_p threads running the `produce` function in parallel.

The objective of our analytical model is to give us the theoretical optimal step that would minimize the time taken to run programs that follow the skeleton above. To do so, we find a function that takes the step as parameter and expresses the time taken to run such a program when the FIFO is configured with this step. By minimizing this function, using the roots of its derivative, we find the optimal step.

Assumptions and notations We make the following assumptions:

- All producers have the same step, denoted S_p, and all consumers have the same step, denoted S_c. Producers and consumers may not have the same step.
- The total amount of data produced across all producers during a single synchronization is equal to the amount of data consumed across all consumers during a single synchronization.

$$S_p * N_p = S_c * N_c \tag{4.1}$$

- It takes the same time to push an element into a local buffer and to pull an element from a local buffer, ignoring synchronization times. This time is denoted C_{copy}.

Listing 4.4: Skeleton of the use of FifoPlus in a producer-consumer context

```
void consume(FIFOPlus<Data>& fifo) {
  for (int i = 0; i < work_limit(); ++i) {
    optional<Data> data = fifo.pop();
    // data will be null if there is no more data in the FIFO
    // and all producers have notified they have nothing more
    // to produce.
    if (!data)
      return;

    // Process data
    // ...
  }
}

void produce(FIFOPlus<Data>& fifo) {
  for (int i = 0; i < work_limit(); ++i) {
    Data data;
    // Produce the data
    // ...
    fifo.push(data);
  }

  fifo.producer_done();
}
```

- Overall, we assume *regularity*. The time taken to produce an element is the same for all elements. This assumption makes the model simpler to understand and manipulate. Moreover, while perfectly regular computations are rare, computations that feature slight, negligible variations are common. In fact, we suppose that the granularity inferred when taking this assumption is a good approximation of the optimal granularity.

Our notations are as follows:

- The amount of time required to produce an element is denoted W_p; it corresponds to the time spent in line 16 in the example above.
- The amount of time required to consume an element is denoted W_c; it corresponds to the time spent in line 8 in the example above.
- The time required to perform a synchronization, *i.e.* acquire the mutex and release the mutex when performing a transfer from/to the shared buffer, is denoted C_s.
- The time required to transfer the content of a local buffer into the shared buffer and vice-versa is denoted $C^p_{transfer}$ for producers, and $C^c_{transfer}$ for consumers. The asymmetry arises from the fact producers may not have the same step as consumers, which may cause them to spend more or less time transferring between buffers than consumers.

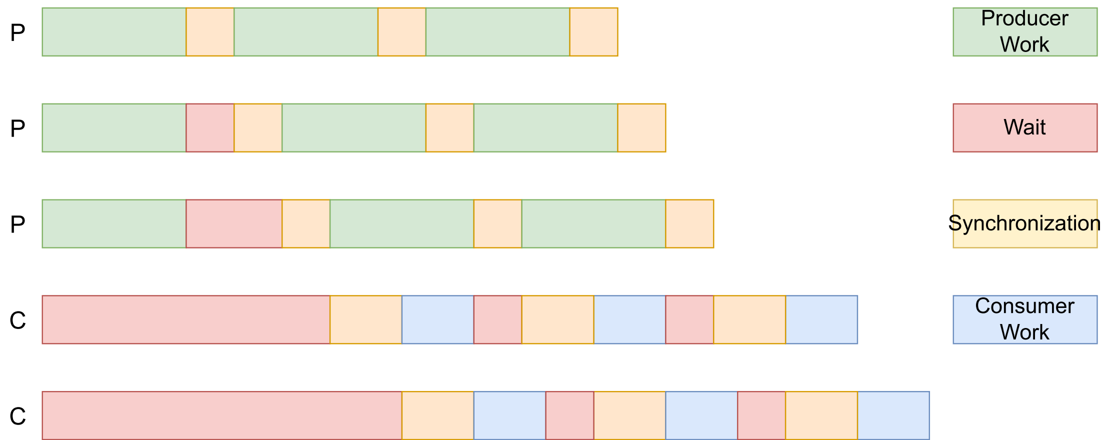


Figure 4.11: Producers as the limiting factor in a regular pipeline

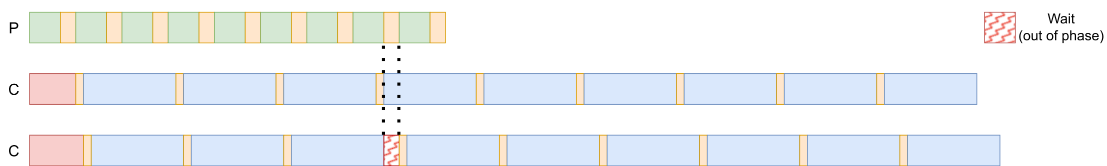


Figure 4.12: Consumers as the limiting factor in a regular pipeline

- We assume that $\forall x \in [p, c], C^{x}_{transfer} = 2 * C_{copy} * S_x$. This expresses the idea that a transfer between a local buffer and the shared buffer requires two operations, performed a number of times equal to the step. For instance, a consumer with a step of 10 will pop 10 items from the shared buffer and push these 10 items into its local buffer during a synchronization.

Formulas We now provide the formulas that give the total amount of time it would take for N_p producer threads and N_c consumer threads, configured with a step of S_p and S_c respectively, to produce and consume I elements.

Figures 4.11 and 4.12 show two different scenarios.

In the first figure, the producers are globally slower than the consumers: it takes more time for all the producers to produce n elements than for consumers to consume them. The opposite can be observed on the second figure. Our analytical model yields two different formulas corresponding to these two scenarios.

To determine what is the limiting factor, we compute two ratios: $CP = N_c * W_p$ and $PC = N_p * W_c$. If $CP > PC$ then producers are the limiting factor (as in Figure 4.11). Otherwise, consumers are the limiting factor (as in Figure 4.12). In both figures:

- Green rectangles represent the time when a producer is producing data, and storing data in the local buffer.
- An Orange rectangle represents a synchronization. This includes the time required to take the mutex, transfer content from the local buffer into the shared buffer /

from the shared buffer into the local buffer, and release the mutex.

- A Red rectangle represents waiting time, when a thread attempts to acquire the shared mutex, but another thread already holds it, or when a thread awaits some data.
- Blue rectangles represent the time when a consumer is consuming data, and retrieving data from the local buffer.

Case 1: Producers are the limiting factor The time taken in a producer-consumer scenario in which producers are the limiting factor can be expressed as :

$time_p$ = WorkAndCopy + ProducerCriticalSection + OtherProducersLastSync + LastConsumersSync + LastConsumerBatch.

We can explain each of these components by using Figure 4.11 as a reference:

1. WorkAndCopy corresponds to the green time of a single producer. Symbolically, it takes W_p time to produce an element, and C_{copy} time to add it to the local buffer. Thus it takes $I * (W_p + C_{copy})$ time to produce all of them for a single producer.

2. ProducerCriticalSection represents the total orange time (transferring from local to shared buffer) of a single producer. A single orange block lasts $C^p_{transfer} + C_s$ time. Taking the step into account, there are $\frac{I}{S_p}$ synchronizations performed, so $\frac{I}{S_p} * (C^p_{transfer} + C_s)$ is spent synchronizing and transferring.

3. OtherProducersLastSync represents the time it takes for all producers, except the first one, to perform their last synchronization (excluding wait time). Similarly to the item above, $C_s + C^p_{transfer}$ is the time taken to perform a single synchronization and the associated transfer.

4. LastConsumersSync represents the time it takes for all consumers to perform their last synchronization (excluding wait time). This is similar to the two points above and can be expressed as $N_c * (C_s + C^c_{transfer})$.

5. LastConsumerBatch represents the time it takes for the last consumer to perform its last batch of work. Consuming an item first require extracting the item (C_p), then actually consuming it (W_c). This is done S_c times, therefore the total can be expressed as $S_c * (W_c + C_{copy})$.

This gives us the symbolic expression of $time_p$:

$$time_p = I * (W_p + C_{copy}) + \frac{I}{S_p} * (C^p_{transfer} + C_s) + (N_p - 1) * (C_s + C^p_{transfer}) + N_c * (C_s + C^c_{transfer}) + S_c * (W_c + C_{copy})$$

S_c and $C^c_{transfer}$ can be expressed, per assumptions, as $S_c = \frac{S_p * N_p}{N_c}$ and $C^c_{transfer} = 2 * C_{copy} * \frac{S_p * N_p}{N_c}$, which gives us the time as a function of S_p.

$$
\begin{aligned}
time_p(S_p) = I * (W_p + C_{copy}) + \frac{I}{S_p} * (C^p_{transfer} + C_s) + \\
(N_p - 1) * (C_s + C^p_{transfer}) + \\
N_c * (C_s + 2 * C_{copy} * \frac{S_p * N_p}{N_c}) + \\
\frac{S_p * N_p}{N_c} * (W_c + C_{copy})
\end{aligned}
$$

Case 2: Consumers are the limiting factor The following formula expresses the time taken to run a producer-consumer scenario in which consumers are the limiting factor. We assume that I, N_p, N_c, W_p, W_c, C_{copy}, C_s, and $C^c_{transfer}$ are constant.

$$
\begin{aligned}
time_c = I * (W_c + C_{copy}) + \frac{I}{S_c} * (C^c_{transfer} + C_s) + \\
(N_c - 1) * (C_s + C^c_{transfer}) + N_p * (C_s + C^p_{transfer}) + S_p * (W_p + C_{copy}) + Contention
\end{aligned}
$$

S_p and $C^p_{transfer}$ can be expressed, per assumptions, as $S_p = \frac{S_c * N_c}{N_p}$ and $C^p_{transfer} = 2 * C_{copy} * \frac{S_c * N_c}{N_p}$, which gives us the time as a function of S_c.

$$
\begin{aligned}
time_c(S_c) = I * (W_c * C_{copy}) + \frac{I}{S_c} * (2 * C_{copy} * S_c + C_s) + \\
(N_c - 1) * (C_s + 2 * C_{copy} * S_c) + \\
N_p * \left(C_s + 2 * C_{copy} * \frac{N_c * S_c}{N_p} \right) + \\
\frac{N_c * S_c}{N_p} * (W_p + C_{copy}) + Contention
\end{aligned}
$$

The explanation of these components is similar to the ones in the previous formula, with the producers and consumers roles reversed. However, in this scenario there is an additional component representing some contention on the mutex. This contention is due to the fact that the consumers may be blocked by the producers pushing data into the shared buffer. It is represented as the zig-zagged part of the bottom consumers timeline.

This contention is exclusive to the case in which consumers are the limiting factor of the computation. Here, as consumers are slower than producers, contention may move each consumer out of phase with the others, which may lead to a consumer being blocked by a producer.

Finding the theoretical optimum We now find the value of S_p (resp. S_c) that minimizes $time_p$ (resp. $time_c$). To do so, we compute the derivative of $time_p$ (resp.

$time_c$) and find the value at which it equals 0. We recall the formulas.

$$time_p(S_p) = I * (W_p + C_{copy}) + \frac{I}{S_p} * (2 * C_{copy} * S_p + C_s) + (N_p - 1) * (C_s + 2 * C_{copy} * S_p) + N_c * \left(C_s + 2 * C_{copy} * \frac{N_p * S_p}{N_c}\right) + \frac{N_p * S_p}{N_c} * (W_c + C_{copy})$$

$$time_c(S_c) = I * (W_c + C_{copy}) + \frac{I}{S_c} * (2 * C_{copy} * S_c + C_s) + (N_c - 1) * (C_s + 2 * C_{copy} * S_c) + N_p * \left(C_s + 2 * C_{copy} * \frac{N_c * S_c}{N_p}\right) + \frac{N_c * S_c}{N_p} * (W_p + C_{copy})$$

We removed the contention from the expression of $time_c$ as experimental measures show that it is negligible near the point at which the derivative gets canceled.

The derivatives are given below.

$$time'_p(S_p) = \frac{-I * C_s}{{S_p}^2} + (N_p - 1) * (2 * C_{copy}) + 2 * C_{copy} * N_p + \frac{N_p * (W_c + C_{copy})}{N_c}$$

$$time'_c(S_C) = \frac{-I * C_s}{{S_c}^2} + (N_c - 1) * (2 * C_{copy}) + 2 * C_{copy} * N_c + \frac{N_c * (W_p + C_{copy})}{N_p}$$

We can now find the values for which both of these functions equal 0, which gives us an estimate of the optimal step in each case: $S_{p_{opt}}$ when producers are the limiting factor, $S_{c_{opt}}$ when consumers are the limiting factor. The derivation process, as well as finding the root of the derivative is straightforward and left to the reader.

$$time'_p(S_p) = 0 \Leftrightarrow S_p = \sqrt{\frac{I*C_s}{(N_p-1)*(2*C_{copy})+2*C_{copy}*N_p+\frac{N_p*(W_c+C_{copy})}{N_c}}} \triangleq S_{p_{opt}}$$

$$time'_c(S_c) = 0 \Leftrightarrow S_c = \sqrt{\frac{I*C_s}{(N_c-1)*(2*C_{copy})+2*C_{copy}*N_c+\frac{N_c*(W_p+C_{copy})}{N_p}}} \triangleq S_{c_{opt}}$$

Once we have the step for the producers (resp. consumers), we use the relation $N_p * S_p = N_c * S_c$ to find the step for the consumers (resp. producers).

Summary These two formulas provide a theoretical approximation of the optimal step, under some assumptions. This model is rather simple but it provides a satisfactory estimation and is easy to understand. We now present the API of FifoPlus, in particular how the analytical model is calibrated using measured data in order to automatically configure the granularity of synchronization.

4.2.4 The FifoPlus Libraries

In this section, we present the API of the different libraries that compose FifoPlus as well as how we achieve efficiency. This section is divided in three different parts: the "FIFO" part of the API which allows the programmer to transfer data to and from a FifoPlus; the reconfiguration API that allows the programmer to provide information to a FifoPlus so it can find the theoretical optimal step; and a discussion on optimization.

4.2.4.1 Data exchange functions

The data exchange API features the standard `push` and `pop` operations. As presented in the introductory section, these operations are performed on local views rather than on the shared buffer directly. This API also features operations to feed data into the analytical model in order to dynamically compute the optimal step. Additionally, it features operations to improve performance in presence of a fork in a pipeline. These were added after some observations on the `dedup` algorithm.

Let `T` denote the type of the elements inside the FIFO. The public API is as follows:

The class **FIFOPlusMain** represents the shared buffer. This class exposes a single public function, **view** that creates a view on this buffer. The programmer first creates an instance of `FIFOPlusMain`, then, for every producer and consumer, calls `view`.

The class **FIFOPlus** represents a view on the FifoPlus: a local buffer, with a step S. This class exposes the following functions:

- **pop**: `void` → `Optional<T>`. This function extracts and returns the next available value in the local buffer. If the local buffer is empty, S elements are extracted from the FIFO. If all producers are done and the shared buffer is empty, this function returns an empty value.
- **cross_pop**: `Timer` → `FIFOPlus*` → `(Optional<T>, bool)`. This function takes as parameters a timer and a secondary FIFO. It works like `pop`, with a single change. If the timer expires before an element can be extracted from the shared buffer, then the function performs a `force_push` (see below) on the secondary FIFO. It returns a set of values: the extracted value (if any) and whether the timer expired or not.
- **push**: `T` → `void`. This function pushes a new element in the local buffer. If the local buffer is full, it performs a transfer into the shared buffer.
- **force_push**: `void` → `void`. This function forcibly triggers a transfer into the shared buffer, regardless of the amount of data present in the local buffer. This function does not add data into the local buffer. In effect, it is similar to the `set_immediate` of PromisePlus, except it does not add data.
- **terminate**: `void` → `void`. This function notifies the shared buffer that a producer has completed the production of all its data. It is used to prevent consumers from getting stuck in an endless loop if the shared buffer definitely runs out of data.

The `push` function has two alternative versions:

- **timed_push**: T → `void`. This function behaves like `push`, while also performing measurements in case a transfer from the local buffer to the shared buffer occurs. The function measures the time spent acquiring the mutex, the time spent transfering

and the time spent notifying consumers and releasing the mutex. This measurements are then given to the analytical model for calibration.

- **generic_push**: T → `bool`. This function acts as a wrapper around `timed_push` and `push`. If the analytical model requires more data to be calibrated, `generic_push` will call `timed_push`. Otherwise it will call `push`. The function returns a boolean indicating if the analytical model requires more data. The main purpose of this return value is to indicate to the programmer when they can stop calling the `add_work_time` function (see 4.2.4.2).

`force_push` is to be used when data produced by a thread may go into two or more shared buffers. This is the case in the Deduplicate stage of dedup, where data may either go to the Reorder stage, or to the Compress stage, depending on whether data is a duplicate or not. The problem solved by `force_push` is that, due to local buffers not transferring into the shared buffer with every push, either branch of the pipeline may stall despite data being ready to be sent. As a result, a programmer may forcibly push to prevent that one branch of the pipeline is getting a lot of work and not the other.

`cross_pop` solves a different problem. In cases where it is possible for a thread to occasionally wait a long time for some data, forcibly transferring towards the next stage of the pipeline can be relevant. Since the step may prevent data from being sent immediately, it is possible for multiple levels of the pipeline to stall simultaneously, due to data remaining in local buffers. `cross_pop` solves this problem by pushing the content of a local buffer into a shared buffer if the pop operation takes too much time.

These two operations are not taken into account in our analytical model, due to their unpredictable nature. Their existence comes from the need to adapt synchronization in cases that are irregular either along one pipeline or between two branches of a pipeline.

4.2.4.2 Reconfiguration Library

The reconfiguration API exposes the **Observer** class. This class is used to trigger the reconfiguration of the step of instances of `FIFOPlus`.

An `Observer` is in charge of multiple `FIFOPlus` that are registered by the programmer. To each `FIFOPlus`, the `Observer` associates a set of time measurements. These time measurements correspond to the data required by our analytical model: time taken to produce or consume an element, time taken to perform a synchronization etc. Once the total amount of data provided by all FIFOs reaches a user defined threshold, the `Observer` triggers a reconfiguration of the step of each `FIFOPlus`.

An `Observer` exposes the following methods:

- **add_producer**: `FIFOPlus*` → `void`. This function is used to register a `FIFOPlus` that acts as a producer.
- **add_consumer**: `FIFOPlus*` → `void`. This function is used to register a `FIFOPlus` that acts as a consumer.
- **add_work_time**: `FIFOPlus*` → `uint64_t` → `void`. This function is used to add the time taken to produce or consume an item. The `Observer` automatically deduces whether a `FIFOPlus` is a producer or consumer.

Both `add_producer` and `add_consumer` must be called before time measurements start. This is a choice that allows us to avoid the termination problem: if producers or consumers can get registered at any point in time, then we cannot decide when a FIFO is done, which makes it impossible to properly terminate consumers.

The definition of "time taken to produce/consume an item" is the time spent between the end of pull and the beginning of the next push. In our current prototype we ask the programmer to measure this manually, both for flexibility and engineering reasons. In particular, our own experience shows that it is sometimes necessary to tweak the definition depending on the problem, therefore an automated measurement would result in incorrect measures. As a future work, we may provide an option to automate these measures.

As an example, in our study case, for the dedup algorithm:

- On the Refine, Deduplicate and Compress stages, the time taken to produce an item is evaluated as the time elapsed between the last successful pull from an input FIFO and the beginning of the next push to an output FIFO.
- On the Fragment stage, the time taken to produce an item is evaluated as the time taken to read a chunk of the input file and process it all the way up to, but not including, pushing it into an output FIFO.
- On the Reorder stage, the time taken to consume an item is evaluated as the time elapsed between the end of a pull and the beginning of the very next pull from one of the input FIFOs.

The time taken to perform a transfer from/to a local buffer to/from the shared buffer is measured internally by the instance of `FIFOPlus` and automatically added to its `Observer`s, which automates part of the programmer's work.

4.2.4.3 Efficiency and Implementation Details

Efficient reconfiguration The reconfiguration API allows a reconfiguration of the step of the different local buffers which is the main way to achieve efficiency. The reconfiguration process needs to be written carefully, so it does not become a source of contention.

Observers The `Observer` keeps track of how much data has been produced by all producers and all consumers separately. We use atomic variables and atomic fetch-add operations to increase counters without having to lock an expensive mutex.

Every time a FIFO produces a data item, it checks whether the overall amount of data produced by producers and data consumed by consumers separately is enough to trigger a reconfiguration, according to a threshold defined by the user. Due to the multithreaded nature of applications that use `FIFOPlus` and the `Observer`s, it is possible to have multiple FIFO operations that trigger a reconfiguration simultaneously on the same `Observer`. To prevent multiple reconfiguration from happening simultaneously, the `Observer` contains two atomic booleans that state whether the first (resp. second) reconfiguration has been performed. This is illustrated in Listings 4.5 and 4.6 that show the skeleton of the data registration function and of the reconfiguration function.

Listing 4.5: Function used to register a production/consumption time in an `Observer`

```
// Add time for the first reconfiguration. src is the FIFO that is
// adding time.  Returns a boolean indicating if the source FIFO
// should keep measuring time.
bool Observer<T>::add_time_first_reconfiguration(FIFOPlus<T>* src,
                                                  uint64_t time) {
  uint32_t operations, max_operations,
              other_operations, max_other_operations;
  if (src->_producer) {
    operations = _prod_operations.fetch_add(1,
                             std::memory_order_acq_rel);
    other_operations = _cons_operations.load(
                             std::memory_order_acquire);
    // User defined threshold, never written again
    max_operations = _max_prod_operations;
    // User defined threshold, never written again
    max_other_operations = _max_cons_operations;
  } else {
    // ... Similar, flip prod and cons
  }

  // ... Skip some micro performance checks and time registration

  if (operations == max_operations &&
             other_operations >= other_max) {
    trigger_reconfiguration(true);
    return false;
  }

  return true;
}
```

Listings 4.5 and 4.6 present the skeleton of the data registration function and of the reconfiguration function, respectively, to better illustrate the points.

The `atomic_compare_exchange_strong` (`cmp_xchg_str` for short) on booleans has the following prototype: `bool compare_exchange_strong(bool& expected, bool new, memory_order succ, memory_order fail)`. It atomically:

1. Checks if the value inside the atomic is the same as `expected`;
2. If it is, then it replaces this value with `new`. This read-modify-write operation has the memory order specified by `succ`.
3. If it is not, then it loads the value stored in the atomic into `expected`. This read-write operation has the memory order specified by `fail`.
4. The function returns whether the exchange was successful or not.

Listing 4.6: Function used to trigger a reconfiguration from an `Observer`

```
// First is a boolean indicating if this reconfiguration is the
// first or second.
void Observer<T>::trigger_reconfiguration(bool first) {
  bool expected = false;
  if (first) {
    if (_first_reconfiguration_done.compare_exchange_strong(
        expected,
        true,
        std::memory_order_acquire,
        std::memory_order_relaxed)) {
      // Proceed with reconfiguration.  Exactly one thread will
      // execute this on this instance of Observer.
    }
  } else {
    // Same as above
    if (_second_reconfiguration_done.compare_exchange_strong(...))
       ↪ {
      // Same observations
    }
  }
}
```

Once the best step has been computed, it needs to be propagated to the different views. Each operation checks for the value of an atomic boolean that indicates whether a reconfiguration is needed. When the `Observer` computes the best step, it writes this step in a dedicated attribute of the view and sets the atomic boolean to true. This allows us to maximize the time spent working, ensuring the view remains available most of the time and the right step is adopted.

4.2.5 Benchmarks

In this section we evaluate the performance of FifoPlus in two different scenarios. In the first, presented in Section 4.2.5.1, we run a hand-written producer consumer program that is regular. This is a favorable case for our analytical model. This allows us to study the influence of the step on execution time and the accuracy of our analytical model. In the second scenario, presented in Section 4.2.5.2, we run the dedup kernel from the PARSEC Benchmark Suite. This scenario tests the limits of our tool in a real world application that does not feature regularity, and consists in a multi-stage pipeline rather than a traditional producer-consumer scenario around a monitor.

Listing 4.7: Microbenchmark used to evaluate performance of `FIFOPlus`

```
void produce(int iterations, int work, FIFOPlus<int>* fifo, Observer<int>* obs) {
  bool need_obs = true;
  for (int i = 0; i < iterations; ++i) {
    time_point begin = now();
    for (volatile int j = 0; j < work; ++j)
      ;

    time_point end = now();
    if (fifo->generic_push(obs, i) && need_obs)
      need_obs = obs->add_work_time(fifo, end - begin);
  }
}

void consume(int work, FIFOPlus<int>* fifo, Observer<int>* obs) {
  bool need_obs = true;
  while (optional<int> res = fifo->pop()) {
    time_point begin = now();
    for (volatile int j = 0; j < work; ++j)
      ;

    time_point end = now();
    if (need_obs) {
      need_obs = obs->add_work_time(fifo, end - begin)
    }
  }
}
```

4.2.5.1 Microbenchmark

In this section, we study the influence of the step on a simple producer-consumer program, where one producer and one consumer communicate with each other using a FifoPlus. Both the producer and the consumer exhibit a regular behavior, *i.e.* it takes the same amount of time to produce any element, and the same amount of time to consume any element. Listing 4.7 presents the (simplified) code of both the consumer, in function `consume`, and the producer, in function `producer`.

This benchmark is parameterized by six things: the amount of items produced (I), the amount of work needed to produce an item (W_p), the amount of work needed to consume an item (W_c), whether the FIFOs should actually perform a reconfiguration, the original step of the producer and the original step of the consumer.

Figure 4.13 presents the results of our benchmark on a graph. It shows the evolution of the time needed to run the microbenchmark (circles and squares represent measured values, dotted lines represent the analytical model) as the step takes the values 1, 2, 4, 5 ... ($\lfloor\sqrt{2}^i\rfloor \forall i \in [0;34]$) with the following parameters:

- There is no reconfiguration. The algorithm that measures the time taken to perform synchronizations, and to produce / consume items is disabled.
- A total of 6,000,000 items is produced.
- The amount of work needed to produce an item and to consume an item is set to 1000.

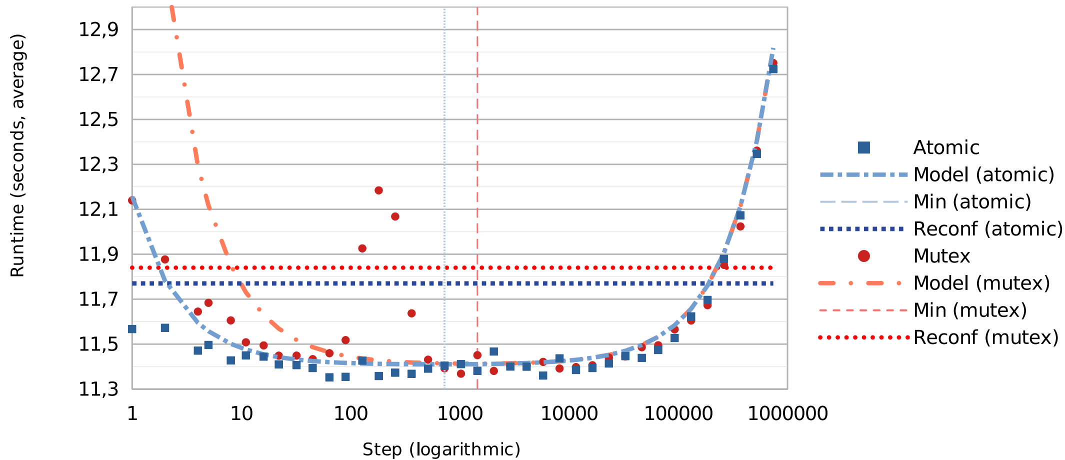


Figure 4.13: Evolution of the run time of the microbenchmark as the step grows

- Both the producer and the consumer have the same step.

This is our baseline, which will allow us to validate our analytical model;

Accuracy of the analytical model The red circles represent the average runtime in function of the step when the synchronization in FifoPlus is performed using a mutex and a condition variable (the default). We plot this against our analytical model, the orange dash-round-dotted line. We can see that our model does not perfectly match against reality: the points between steps 1 to 100 are slightly off, and the points between steps 100 and 500 suffer from a performance degradation that was not predicted by the analytical model. Overall, our model is 1% pessimistic, but the shape of the curve matches reality which is the crucial part.

The discrepancy between 1 and 100 can be explained by our assumption that the cost of synchronization is a linear function of the step. In reality, caching mechanisms allow a Cpu to lock a mutex faster on repeated lock operations.

We suspect the performance degradation between steps 100 and 500 may come from the passive wait triggering extra system calls, and the way the Linux kernel implements POSIX mutexes. To test this hypothesis, we reimplemented the mutex and condition variables used for the synchronization in FifoPlus with atomic variables and busy-waiting and ran the benchmark again. The result is represented by the blue squares on the figure. We can see that the points between steps 100 and 500 are no longer completely off. This seems to confirm our intuition that the passive wait of mutexes has an unintended side effect that causes performance to degrade for steps between 100 and 500.

For completeness, we plotted this new batch of runs against our analytical model, this time configured with the data extracted from the new runs. The blue dash-dotted lines represent this. Similarly to the mutex version, the first points, between steps 1 and 10, are slightly off, although the difference is less pronounced. While the busy-waiting based

reimplementation is faster when synchronization is fine-grained, the potential performance gain for the optimal step is negligible. Moreover, the higher energy consumption of busy-waiting is, in this case, not acceptable, as the gain is, as stated, negligible. For these reasons, we kept the standard library's mutex and condition variable in the final implementation.

This shows that our analytical model is quite precise when it comes to finding the low point of the curve. The fact that it is less accurate for low steps is not an issue because the curve is a flattened reversed bell, meaning that even if the first steps do not match perfectly against reality, there is a wide-enough interval of "reasonnable" steps that will also give a good performance.

Note. To create the two "Model" curves, the analytical model was configured using data from a run where both the producer and the consumer were configured with a step of 1 and measurements were done for the entire duration of the execution, rather than only on the first few operations. We obtain more precise values for the parameters of the model than measuring only on the first few operations as done in actual runs. In particular, we have observed that the measures performed during the first half second of execution are less precise. For instance, the average time required to produce an element measured over the first 100 elements (W_p in the model) was 5 μs. The average time required to produce an element measured over all 6000000 elements was 2 μs. Similar results were observed on the time required to acquire and release the mutex protecting the shared buffer (C_s).

Accuracy of reconfiguration and cost of the analysis Table 4.2 presents the time taken to run the microbenchmark with five different configurations.

Configuration "Balanced" acts as a baseline where the producer and the consumer have a balanced amount of work. Configurations "Slow Prod" and "Slow Cons" allow us to evaluate the impact of an unbalanced amount of work between producers and consumers, in one direction and the other. Configurations "Balanced+" and "Balanced++" act as variations on configuration "Balanced" by increasing the amount of work and reducing the amount of synchronization, keeping the total amount of computation constant. They allow us to check that the analytical model gives us adequate steps.

Each configuration was run in three different modes: with a step of 1 and no reconfiguration (column "S = 1"), with a step of 1 and reconfiguration (column "Reconf"), and with the step found by the reconfiguration and no reconfiguration (column "S = "best"). Table 4.3 gives the speedup of the reconfiguration, the speedup of the best step, and the cost of reconfiguration.

On Figure 4.13, the red square dotted line represents the average time it took to run the microbenchmark in the "Balanced' case with reconfiguration enabled when using a mutex to perform the synchronization. The blue dotted line represents the average time when reimplementing mutex and condition variable with an atomic. The two lines are close to each other while atomics are slightly faster, which is also expected.

If we look at the runs of the microbenchmark without measurements nor reconfiguration, the optimal step was located in the interval [512; 1024]. On average, the views

Table 4.2: Runtime in seconds of the microbenchmark with a step of 1 and no reconfiguration, with reconfiguration, and with the step found during reconfiguration

Configuration	Items	Producer	Consumer	S = 1	Reconf	S = "best"
Balanced	6000000	1000	1000	12.14s	11.84s	11.41s (792)
Slow Prod	6000000	4000	1000	58.95s	44.80s	44.46s (2051)
Slow Cons	6000000	1000	4000	44.67s	44.38s	44.27s (462)
Balanced+	600000	10000	10000	11.36s	11.23s	11.17s (79)
Balanced++	60000	100000	100000	11.18s	11.17s	11.30s (5)

Table 4.3: Speedup and cost of the reconfiguration for the measurements of Table 4.2

Configuration	Speedup 1 → Reconf	Speedup 1 → Best	Reconfiguration cost
Balanced	1.025	1.064	0.43s
Slow Prod	1.31	1.32	0.34s
Slow Cons	1.006	1.009	0.11s
Balanced+	1.011	1.017	0.06s
Balanced++	1.001	0.989	-0.13s

were reconfigured with a step of 792, which falls in the correct interval. This seems to indicate that the implementation of our model works properly when calibrated with runtime measurements.

The cost of the analysis is higher than what we expected. Longer runs, as well as runs with more computation than synchronization, seem able to limit the impact of this overhead, but shorter, balanced runs, with a high quantity of synchronization are more impacted. Further experimentations did not allow us to conclude on the origins of this overhead, as the time before the reconfiguration is negligible. Our current hypothesis is that the reconfiguration itself has an adverse effect on cache, but this would need further study.

Summary Through this benchmark, we have seen that our analytical model is quite close to reality, that changing the step improves performance, that the deduced step is not far off from reality, and that the cost of measurement is acceptable.

4.2.5.2 Running the dedup algorithm with FifoPlus

In this section, we check how well our analytical model holds in a scenario that should be more unfavorable: the dedup algorithm. Dedup has two features that we do not take into account in our model:

- dedup is a pipeline. Our analytical model works on a single FIFO. Our strategy is to configure each FIFO independently and assume that the pipeline will get optimized.

- dedup is not regular. The time required to compress a chunk of data or to detect whether it is a duplicate or not may vary wildly. Experimental results show that this time may take any amount of time between one microsecond and several milliseconds. We apply the same strategy as for regular programs, and observe how well our model handles this situation.

Because of our new synchronization pattern, we adapted the algorithm with the following optimization: at the Deduplicate stage, every time either the FifoPlus towards the Compress stage or the FifoPlus towards the Reorder stage receive 100 consecutive items, a `force_push` is performed on the other FifoPlus. This modification prevents Reorder from stalling if the next chunk it expects is blocked in a view that does not have enough items in it to trigger a transfer into the shared buffer.

We run the dedup algorithm on two different files: the "native" input, provided with the PARSEC benchmark suite, that is a 700 MB file which features duplication of data in such a way that the algorithm will detect that roughly half the chunks of data are duplicates; and a second input, "OnlyCompress" that contains (almost) no duplicates (a 1.5 GB MP4 file). The objective of the second file is to have an input such that the algorithm never branches in the pipeline. In practice, this should be a more favorable case as it will make the algorithm behave in a more regular way, and therefore in a more predictable way, which will benefit our algorithm.

For each of these files we provide a graph that displays the results. On each of these graphs, there are three different series of points, labelled "Original" (blue $\times$), "Analytical" (red $+$) and "Reconfiguration" (dotted line).

- The "Original" points (blue crosses) represent the execution time of the algorithm where the buffers used to exchange data are the buffers provided in the original version of the algorithm. Each buffer is dimensioned in such a way that our constraint $N_p * S_p = N_c * S_c$ holds.
- The "FifoPlus" points (red pluses) represent the execution time of the algorithm where the buffers used to exchange data are FifoPlus configured with the given step. We do not perform any reconfiguration nor any time measurements. This merely evaluates that using FifoPlus as a monitor will not incur a time penalty.
- The "FIFO (no optims)" (lime dots) points represent the execution time of the algorithm where the buffers used to exchange data are FifoPlus configured with the given step. We do not perform any reconfiguration nor any time measurements. In addition, the optimization that prevents Reorder from stalling is disabled. This allows us to evaluate the impact of this optimization.
- The "Reconfiguration" line (purple) represents the time it took to run the dedup algorithm when using FifoPlus with an initial step of 1 and reconfiguration enabled.

Table 4.4 presents the total number of chunks processed in each configuration, as well as the percent of duplicates. Figure 4.14 presents the result of running the dedup algorithm on the native input and Figure 4.15 presents the result of running the dedup algorithm on a file containing almost no duplicates. In these two versions, no measurements other than the total runtime were performed, and no reconfiguration happened.

The purpose of the Original and FifoPlus versions is to compare the efficiency of our

Table 4.4: Amount of duplicates in each of the dedup input files

File	Chunks	Duplicates %
Native	369950	54%
OnlyCompress	384363	0.4%

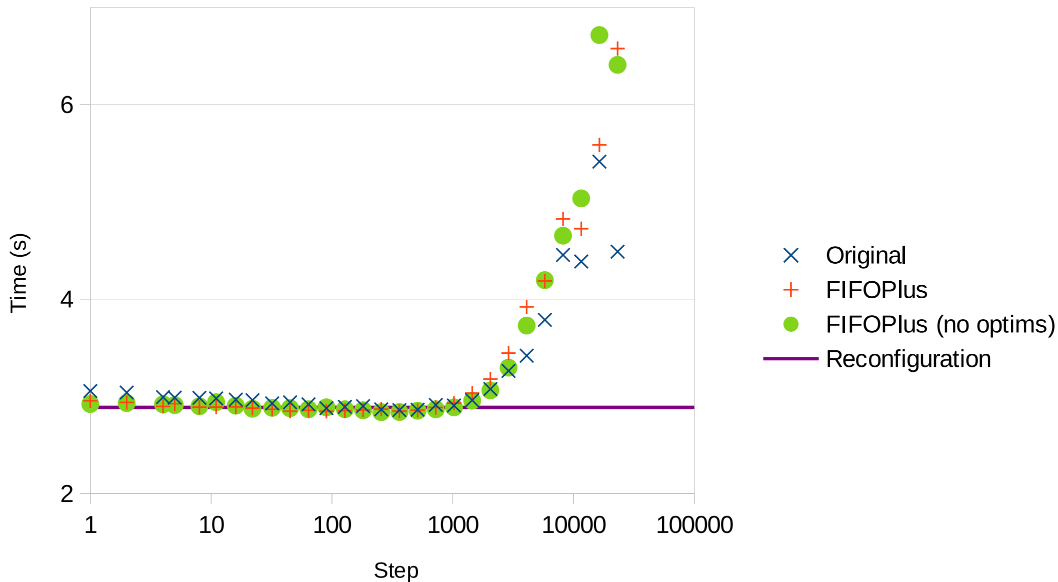


Figure 4.14: Runtime of the dedup algorithm depending on the step (native input)

implementation of the FIFO compared to the original one. In the Original and FifoPlus versions, communication channels were configured according to the step and according to the assumptions of our analytical model. For instance, if the step is 1, then all the steps are 1 except the producer before compress and deduplicate; this one is 5 because there are five consumers for the FIFO; similarly, the reorder stage consumes data with a step of four because there are four producers before reorder. A reminder of the structure of the pipeline can be found in Figure 4.9.

We obtain different results depending on the input.

FIFO efficiency In the "Native" case (Figure 4.14), our FIFO performs similarly to the ringbuffer of the original version for steps lower than 2048. After this point the original version performs better. In the "OnlyCompress" case (Figure 4.15), our FIFO outperforms the ringbuffer by 25% at most (on step 256), and underperforms by 67% at most (on step 32768).

This improved performance in "OnlyCompress" appears due to the optimizations we added to prevent Reorder from stalling. Because the "OnlyCompress" file features

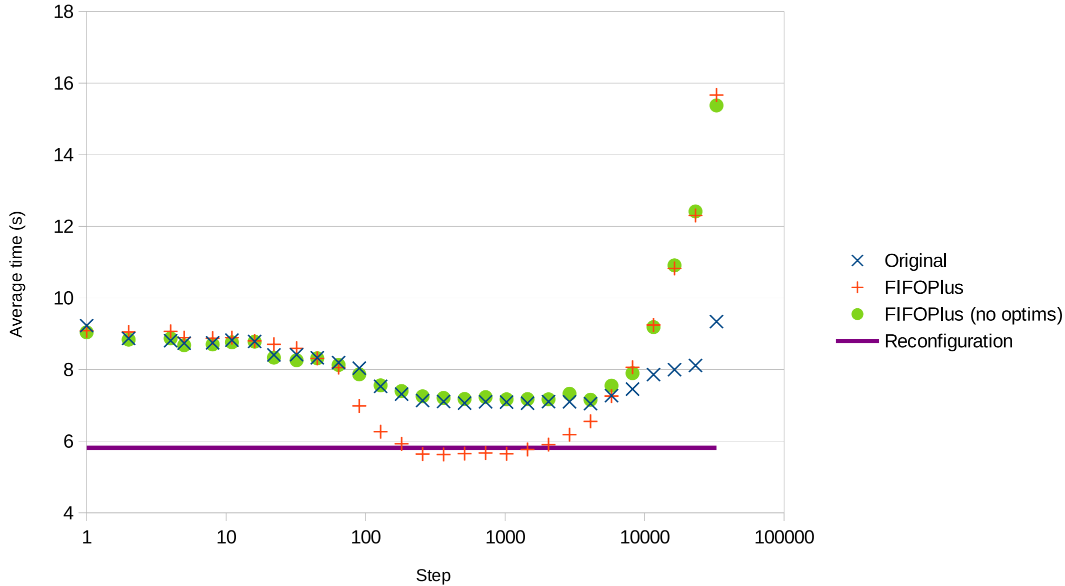


Figure 4.15: Runtime of dedup depending on the step (only compress chunks)

an extremely low, but non-null, amount of duplicated data, if the next chunk expected by Reorder is a duplicate that remains stuck in the FIFOPlus between Deduplicate and Reorder because there is not enough data produced to actually trigger a synchronization, time will be lost. Since FIFOPlus features the `force_push` that forcibly triggers a synchronization, we worked around the problem of stuck chunks by inserting calls to `force_push` if the Deduplicate stage starts favoring one branch of the pipeline over the other. On the other hand, there is no gain obtained from the optimization in the "Native" case. Close inspection of the input file shows that there is not much to be gained by preventing stall, as its structure never causes chunks to remain stuck in a view for a long time.

On both figures, the red + points of "FIFOPlus" are higher than the blue x points of "Original" from step 11585 onwards. We were not able to find an explanation as to why FIFOPlus underperforms when configured with high steps. This result calls for investigation, however in the context of this benchmark it does not have consequences as these steps are far from the optimum.

Performance of the reconfiguration In both cases, the reconfiguration algorithm is able to find a set of steps for the different FIFOs that gives performance similar to the performance of FIFOPlus without reconfiguration in the best case scenario. There is a 5% increase in performance in the "Native" scenario when compared to the original version of dedup configured with a step of 1 and a 25% increase in performance in the "OnlyCompress" when compared to the original version configured with a step of 1.

On the first file, the average time in the reconfiguration scenario is only 1.3% slower

than the best average time without reconfiguration. This is quite positive as this optimum would have to be found by heavy benchmarks and manual configuration. On the second file the reconfiguration scenario is only 3% slower than the best average time without reconfiguration, which is also quite positive.

Finally we should note that we performed additional experiments with a scenario less favourable for us, a larger file with 58% of duplicates. In this case the reconfiguration scenario is slower by 3.1% when compared to the original version with the optimal step. The reconfiguration does not slow down execution either. We believe this is still acceptable.

Overall, we consider the results on the dedup algorithm to be encouraging: even when our model is not perfectly adapted, it can provide satisfactory performance.

4.2.6 Summary

Both these benchmarks show that our analytical model is able to find steps that improve performance. On computations that are regular the deduced step is quite accurate and yields performance close to optimal. In situations that are less favorable, the analytical model is able to compute steps that give local improvements to performance that add up to a global gain. In all cases, the analytical model never found a step that reduced the performance. This points towards more work regarding FifoPlus, in particular a refinement of the analytical model to handle pipelines and eventually support for less regular computations.

4.3 Conclusion – Granularity of Synchronization

In this chapter we presented two new constructs, PromisePlus and FifoPlus that are synchronization abstractions aiming at improving performance of parallel applications that exchange many data. Our focus was the granularity of synchronization and how this granularity needs to be adapted depending on the program in order to improve efficiency of the application.

PromisePlus relies on a static configuration of the granularity: a promise that abstracts the concept of an array and synchronizes on slices of this array. The size of these slices is configured once at runtime, and is specified by the programmer. Our first results are promising even if finding the optimal granularity for synchronization might be difficult.

FifoPlus improves upon the static granularity of synchronization of PromisePlus by embarking an analytical performance model that is able to determine an approximation of the ideal granularity of configuration. This analytical model, while still in its early stages, is able to find a granularity of synchronization that improves performance on applications that are not to far from the conditions of applicability of the approach that we have described. These results are encouraging, and push towards more work on this analytical model to further improve performance on a wider range of applications.

Chapter 5

Positioning

This chapter complements the early positioning of Chapter 2. We compare our contributions with OpenStream, SkePU, StreamIt, the streaming futures of ABS, and distributed futures.

5.1 OpenStream

As previously discussed, OpenStream is an OpenMP extension and thus targets OpenMP programs. On the other hand, PromisePlus and FifoPlus are not tied to a specific framework. One of our examples, using PromisePlus in the NAS-LU algorithm shows that it can be used in the context of OpenMP, although it could also be used in a different context. This can be seen with FifoPlus, which, while different from PromisePlus, is similar enough in its construction, and was successfully used in the `dedup` algorithm.

There are some additional similarities between both approaches. In [53], Pop points out a problem with the termination of OpenStream tasks. OpenStream tasks terminate after executing their last activation. As we have seen before, in order to be activated, an OpenStream task needs to have a certain amount of data available (its burst). If the amount of data given as input for the last activation is less than the burst, then the task will never activate. The solution to this problem is to have other tasks insert fake data into the input stream before terminating. We encountered a similar problem that is solved through the `set_immediate` (resp. `terminate`) methods of PromisePlus (resp. FifoPlus), which, while still leaving the burden on the programmer's shoulders, do not require fake values.

Our work on finding the best step could be reused inside OpenStream, as the framework allows for dynamic horizons and burst. The OpenMP `task` pragma could allow for automation of some measurements, in particular the work time. In FifoPlus, the programmer needs to measure some of these values manually and the lack of annotations to disambiguate which parts of the code belong to production, consumption or simply control makes this much more complicated.

OpenStream also exposes the communications between tasks in a more natural way than what we do with FifoPlus. This is tied to the use of OpenMP tasks. OpenStream

allows OpenMP tasks to have an `input` or `output` tag, which directly provides the information as to whether a task is a producer, a consumer or both. A view on FifoPlus is either a producer or a consumer, but there is no distinction at the type level. Such a distinction could be added to prevent accidental calls to `terminate` on consumer views at compile-time for instance.

5.2 SkePU

SkePU is a library with a focus on algorithmic aspects, while PromisePlus and FifoPlus are focused on synchronization. However, there are two mechanics in SkePU that are closer to our interests: the auto tuning of skeletons and the slicing of tiled computations.

Auto tuning The auto tuning in SkePU is used to determine the best accelerator on which to run a given skeleton. This is performed using an offline machine-learning algorithm, that uses pre-recorded estimates in order to predict the time required to run a skeleton call on a given accelerator. As SkePU is able to record invocation of skeletons and build a graph of calls (the lineage), the tuning algorithm can determine where to run each call to achieve maximum efficiency.

The analytical model in FifoPlus does not yield the optimal accelerator, it yields the optimal granularity of synchronization. Furthermore, the formulas used by our model are hand-written rather than computed by a machine-learning algorithm. As pointed out in Section 4.2.3, not using a machine-learning algorithm makes us less precise, but allows us to understand its output. Our model also does not require pre-calculated results, and instead computes an approximation on-the-fly, using only data measured during execution. While this allows us to have unique runs that perform well, repeated runs suffer from the overhead of the measurements, instead of immediately using the previously computed step.

Slicing SkePU can slice the input of skeletons into tiles, in order to increase parallelism. Tiling is generally used on arrays or matrices to have multiple threads work on different tiles in parallel, assuming there are no dependencies between tiles. This is similar to what motivated PromisePlus, although different. In PromisePlus, we had a fixed number of threads that worked on slices of a matrix, and the step determined the granularity of synchronization. In SkePU, the slicing must ideally result in the highest amount of parallelism possible. The analytical model of FifoPlus could be adapted to be used in SkePU in order to find the optimal tile size.

5.3 StreamIt

StreamIt shares some strong similarities with our work: some of our usecases would be easy to adapt in the context of StreamIt. For instance, each layer in the `dedup` algorithm could be seen as a StreamIt `Filter`: one that fragments an input into big chunks, one

that refines the big chunks into other chunks, one that performs duplicates detection, one that performs the compression and one that performs the reordering of chunks.

To represent something like the fork at the deduplicate layer of `dedup`, the `SplitJoin` composition filter can be used. Additionally, it could be possible to abstract away the explicit calls to `push` and `pop` by representing the whole sequence of `Filter`s as the composition filter `Pipeline` of StreamIt.

Comparison A first point of comparison between StreamIt and our solutions is the difference discussed in Section 2.2.1 between languages and libraries. Indeed, StreamIt, being entirely dedicated to streaming, requires the programmer to learn a specific programming paradigm which makes streaming more easy, mainly through the use of language constructs. Additionally, many of the optimization performed by the compiler are geared towards the execution context of StreamIt: slicing, fusing or partitioning filters are operations that would be considered too specific in a general-purpose language like C or C++.

On the other hand, our tools PromisePlus and FIFOPlus are designed to fit into existing programming models: programmers do not need to learn anything new, besides an API. Additionally, code rewrite is brought down to a minimum, as we have seen with some of the examples in the respective sections of PromisePlus and FIFOPlus, mainly because programmers remain in the same programming model. However, not having a dedicated compiler limits the optimizations on PromisePlus and FIFOPlus to what exists within the compilers of the language in which they are used. When using PromisePlus, what constitutes the code of the “filter” is not as clear-cut when compared to a dedicated `work` method inside a class.

However, FIFOPlus and StreamIt are not mutually exclusive. Communication between filters in StreamIt is performed through FIFOs, and as we pointed out in the initial presentation of StreamIt, the creators of StreamIt originally wanted to remove the static nature of data rates. FIFOPlus could be used to solve this problem while also abstracting away the need for an initial configuration of the data rates. Adding the FIFOPlus reconfiguration algorithm to StreamIt seems to be more interesting than adding StreamIt optimization to other languages: changing a compiler is difficult, even more-so when it is a compiler for a general-purpose language, and besides, these optimization would be too specific and bloat the compiler. Moreover, having the FIFOPlus reconfiguration algorithm in StreamIt could allow the language to benefit from optimization performed by the compiler that could lead to a better deduction of a good data rate.

Intuitively, we think our solution based on an approximation of the execution in order to reconfigure the data rate of a FIFO could be implemented in StreamIt without too much difficulty, however we did not investigate this point further.

5.4 Futures for Streaming Data in ABS

There is not much that we did not say in our first presentation of ABS. The concept of granularity of synchronization could be added to streams in ABS, as it would probably

lead to performance improvement. However, as we do not have knowledge of the inner workings of the ABS language, it is impossible for us to estimate the difficulty in adding this feature.

As is expected from the previous comparisons, our analytical model could fit nicely in ABS. The use of a dedicated keyword to add data into a stream, and the use of a dedicated construct to pull data from the stream is, like in OpenStream, a good way to have clear indications of when to start and stop clocks.

In [53], the creators of streaming futures in ABS provided operational semantics for their new construct[1], which is not the case on our part, although it can be considered future work.

5.5 Distributed futures

The defining idea of Chapter 4 is to slice data into chunks of a given size that work as synchronization entities inside an array (PromisePlus) or an infinite stream of data (FifoPlus). This idea is similar to what happens with distributed futures (see Section 2.5.1). Unlike distributed futures, PromisePlus and FifoPlus allow synchronization on a chunk of the array rather than having to wait for the entire computation to be done. However, there are considerations from distributed futures that could be investigated in the future, in particular on PromisePlus. While PromisePlus allows synchronization on chunks of arrays, it suffers from the same problem as traditional futures: it needs to store the entire array. This problem was identified by Leca as a limitation of futures and resolved in distributed futures by replacing the array with a description of the location of every chunk. In the future, PromisePlus could maybe exploit the same strategy when it is used in a distributed context.

[1]Some criticism could be leveled at the semantics of non-destructive streams with regards to the way they interact with the garbage collector

Chapter 6

Future Work and Conclusion

6.1 Perspectives

Dataflow explicit futures At the end of this thesis, dataflow explicit futures have reached a mature state: their API is intuitive, easy to use, and their performance matches that of control-flow futures . However, there is still work to do in order for them to reach a more mainstream audience: as it stands right now, dataflow explicit futures exist only as a prototype in Encore, an academic language. On the theoretical side, the equivalence between `forward*` and `return` when applied to dataflow explicit futures offers the possibility of new optimization. A future work here could be to design static analyses that could determine whether it would improve performance to compile a `return` as a `forward*`.

Algorithmic skeletons Both PromisePlus and FifoPlus could benefit from higher-level of abstraction, such as algorithmic skeletons. Algorithmic skeletons abstract away the structure of a computation, and let the programmer separate the programming of basic blocks from their composition. As it stands right now, we would need to add checks to the `set` and `set_immediate` functions of PromisePlus to ensure there are not multiple producers. PromisePlus could be transformed into a skeleton that is parameterized by two functions, one for the producer and one for the consumers, and the number of consumers (or, alternatively, a function for each consumer). This way, it would prevent end users from accidentally having multiple producers work on a single PromisePlus: the skeleton would launch a single producer thread that would run the producer function, and the requested number of consumer threads that would each execute the consumer function.

Safer API for FifoPlus There are some improvements that can be made on the FifoPlus to improve safety. As stated in the API of FifoPlus, the `terminate` method should only be called by producers, never by consumers. This is difficult to enforce statically without changing the API. It could be possible to split the views between

consumer views, that do not have access to `terminate` nor to `push` operations, and producer views, that have access to `push` and `terminate`, but not to `pop`.

Better analytical model and measurement performance for FifoPlus Benchmarks provided in Section 4.2.5.2 show room for performance improvement on two aspects.

Measurement performance First, we believe there is a need to understand why enabling measurements and the reconfiguration process induces a degradation of performance of 0.4 s, despite the fact that reconfiguration occurs less than 1 μs after launch of the application. While we suspect it comes from the regularity of the application being broken by the reconfiguration itself, we have not confirmed it using a profiler or some low-level tool. We have also observed that measurements performed at the very beginning of execution are less precise, and it could be beneficial to have FifoPlus automatically discard early measurements in order for the reconfiguration to be more accurate.

Pipelines The benchmarks we provided in Section 4.2.5.2 show that the analytical model we created is able to find steps that optimize a pipeline. We also noted that this depends on the structure of the pipeline: when the flow of data is straightforward, without branching, performance almost reaches a theoretical optimum. On the other hand when the flow of data goes through a split-join structure, performance does not increase as much, and we have experimentally observed an input for the dedup algorithm on which the reconfiguration does not improve performance (without reducing it either). The split-join structure makes the prediction of runtime difficult due to its non statically deterministic nature. For instance, in dedup it is impossible to know before slicing the file into chunks whether there will be a high or low amount of duplicates and where in the file they will be located. This does not allow us to properly configure the FifoPlus shared by the Deduplicate and Compress stages towards the Reorder stage, as the different producers perform computations of different duration, which makes the estimation of W_p difficult. A potential solution to this problem would be to create a different analytical model that is parameterized by the different W_p of each branch, and a user-provided heuristic function that gives an approximation of how much data will flow through each branch. Instances of FifoPlus that are the join of multiple different branches of a split pipeline would need to use this analytical model instead of the one we designed here. Having different analytical models for different data flow scenarios also points towards using higher-level abstractions. Having `FastFlow`- or `StreamIt`-like constructions to represent the stages of the pipeline could abstract away most of the configuration of the different models depending on the situation.

6.2 Conclusion

Throughout this thesis, we investigated several approaches to create safe, simple and efficient abstractions. Dataflow explicit futures are a type-driven approach that leverages typing in order to offer more expressive futures, equipped with a data-driven synchronization. This allows dataflow explicit futures to have all the advantages of explicit and implicit futures. Furthermore, we have proven the Godot conjecture that `forward*` is equivalent to `return`, which paves the way for further optimization. PromisePlus and FifoPlus are library-driven approaches. They aim to improve performance by letting the programmer configure the granularity of synchronization of a data exchange or automatically reconfiguring the granularity based on observations. With FifoPlus we created a simple analytical performance model that can be used to model the performance of regular applications that communicate through a FIFO queue.

On the level of abstraction These three abstraction tools focus on synchronization, but on different levels. Dataflow explicit futures are a high-level construct, where most of their strength comes from the type system. The subtyping rule and the collapsing rule, both of which cannot be expressed solely through libraries, are keys here. The subtyping rule brings more flexibility and expressivity to the language, while the collapsing rule encapsulates the dataflow aspects.

PromisePlus and FifoPlus on the other hand are low-level constructs, and as such enable low-level optimizations. The thread-local pessimistic index of PromisePlus, and the use of atomics to scrap every bit of performance are optimizations that fit nicely with the level of abstraction we targeted. The analytical model of FifoPlus also reflects this lower-level approach, as it directly uses the granularity of synchronization in its estimation of the time taken to run a given program.

Despite the fact that our contributions fall neatly into either “high-level” or “low-level” abstractions, it would be wrong to assume high-level and low-level to be mutually exclusive, or that focus should be only on one or the other. Abstractions are built on top of each other and efficient low-level building blocks are key to build efficient high-level abstractions.

On theory and practice All the contributions we presented mix theoretical foundations and practical implementations. We showed how crucial is the interplay between theory and practice, for example the theoretical background laid down by “Godot” paved the way for our implementation of a safe and efficient dataflow future library. Indeed, “Godot” brought the simplicity of the API and an elegant and intuitive type system that makes working with flows easy. On the other side, only practical considerations led us to an efficient implementation of PromisePlus that relies on efficient low-level synchronizations (and not, e.g. mutexes). In general, the theoretical contribution provides well-thought bases, with safety and potential performance in mind. The practical contribution adapt the theory based on the context in which the implementation is done, with efficiency concerns in mind.

Bibliography

[1] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the benefits of implicit and explicit futures. *Leibniz International Proceedings in Informatics*, 134, 2019.

[2] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 260–267, New York, NY, USA, 1988. Association for Computing Machinery.

[3] Nicolas Chappe, Ludovic Henrio, Amaury Maillé, Matthieu Moy, and Hadrien Renaud. An optimised flow for futures: From theory to practice. *The Art, Science, and Engineering of Programming*, 6, 07 2021.

[4] Amaury Maillé, Ludovic Henrio, and Matthieu Moy. Promise plus: Flexible synchronization for parallel computations on arrays. In *FSEN 2021 - 9th IPM International Conference on Fundamentals of Software Engineering*, pages 190–196, Tehran, Iran, 10 2021.

[5] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[6] Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, page 55–59, New York, NY, USA, 1977. Association for Computing Machinery.

[7] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, oct 1985.

[8] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*, pages 1–56. Springer International Publishing, Cham, 2015.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore

architectures. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[10] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[11] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In Karsten M. Decker and René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218, Basel, 1994. Birkhäuser Basel.

[12] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] Júnior Löff, Renato Hoffmann, Ricardo Pieper, Dalvan Griebler, and Luiz Fernandes. Dsparlib: A c++ template library for distributed stream parallelism. *International Journal of Parallel Programming*, 50:1–32, 10 2022.

[14] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[15] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[16] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[17] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall Englewood Cliffs, 1989.

[18] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017.

[19] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *LNCS*, pages 142–164. PUB-SV, 2011.

[20] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages*, pages 101–120, Cham, 2016. Springer International Publishing.

[21] Elias Castegren. Capability-based type systems for concurrency control. 2018.

[22] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[24] Derek Wyatt. *Akka concurrency*. Artima Incorporation, 2013.

[25] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, 1998.

[26] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP '10, page 5–14, New York, NY, USA, 2010. Association for Computing Machinery.

[27] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[28] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for gpu-based systems. *International Journal of Parallel Programming*, 44:506–530, 06 2016.

[29] August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019. e5003 cpe.5003.

[30] Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 25–32, 2011.

[31] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the skepu skeleton programming library. In Chenggang Wu and Albert Cohen, editors, *Advanced Parallel Processing Technologies*, pages 170–183, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[32] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[33] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, may 1998.

[34] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[35] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[36] Thierry Gautier, João V.F. Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308, 2013.

[37] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. New York, NY, USA, 2005. Association for Computing Machinery.

[38] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 555–566, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[39] Dean Tullsen and Brad Calder. Computing along the critical path. 08 2002.

[40] F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: Online building data flow graph in a parallel language. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, pages 88–95, 1998.

[41] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. pages 15–23, 07 2007.

[42] Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, page 118–131, New York, NY, USA, 2016. Association for Computing Machinery.

[43] Ludovic Henrio and Justine Rochas. Multiactive objects and their applications. *Logical Methods in Computer Science*, Volume 13, Issue 4, November 2017.

[44] Ludovic Henrio. Data-flow Explicit Futures. Research report, I3S, Université Côte d'Azur, April 2018.

[45] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9:1–31, 01 1999.

[46] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 209–220, New York, NY, USA, 1995. Association for Computing Machinery.

[47] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. *SIGPLAN Not.*, 39(1):123–134, jan 2004.

[48] Kiko Fernandez, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. *Forward to a Promising Future*, pages 162–180. 01 2018.

[49] Stephens. R. A survey of stream processing. In *Acta Informatica*, volume 34, pages 491–541, 1997.

[50] Sussman. J Abelson. H, Sussman. G. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[51] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[52] Antoniu Pop and Albert Cohen. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013.

[53] Antoniu Pop. *Leveraging streaming for deterministic parallelization: an integrated language, compiler and runtime approach*. Theses, École Nationale Supérieure des Mines de Paris, September 2011.

[54] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew Lamb, Jeremy Wong, Henry Hoffman, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, Oct 2002.

[55] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*. 03 2014.

[56] Massimo Torquati. Single-producer/single-consumer queues on shared cache multicore systems. 12 2010.

[57] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. volume 7484, pages 662–673, 08 2012.

[58] Reiner Hähnle. *The Abstract Behavioral Specification Language: A Tutorial Introduction*, pages 1–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[59] Keyvan Azadbakht, Nikolaos Bezirgiannis, and Frank S. de Boer. On futures for streaming data in abs. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 67–73, Cham, 2017. Springer International Publishing.

[60] Keyvan Azadbakht, Frank S. de Boer, Nikolaos Bezirgiannis, and Erik de Vink. A formal actor-based model for streaming the future. *Science of Computer Programming*, 186:102341, 2020.

[61] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, jul 1963.

[62] Pierre Leca, Wijnand Suijlen, Ludovic Henrio, and Françoise Baude. Distributed futures for efficient data transfer between parallel processes. pages 1344–1347, 03 2020.

[63] Svend Frølund and Gul Agha. Abstracting interactions based on message sets. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, pages 107–124, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[64] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, aug 1975.

[65] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 372–385, New York, NY, USA, 1996. Association for Computing Machinery.

[66] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages*, pages 135–152, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[67] Aaron J. Turon and Claudio V. Russo. Scalable join patterns. *SIGPLAN Not.*, 46(10):575–594, oct 2011.

[68] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. Compositional Verification of Concurrent Systems by Combining Bisimulations. In *FM 2019 - 23rd International Conference on Formal Methods*, volume 11800 of *Lecture Notes in Computer Science*, pages 196–213, Porto, Portugal, October 2019. Springer Verlag.

[69] Nicolas Chappe, Ludovic Henrio, Amaury Maillé, Matthieu Moy, and Hadrien Renaud. An optimised flow for futures: From theory to practice. *The Art, Science, and Engineering of Programming*, 6, 07 2021.

[70] Pony, an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language, 2021.

[71] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[72] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and efficient bounded fifo queues. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 144–151, 2013.

[73] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. The openmp implementation of nas parallel benchmarks and its performance. 1999.

[74] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[75] Ruslan Nikolaev and Binoy Ravindran. Wcq: A fast wait-free queue with bounded memory usage. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 461–462, New York, NY, USA, 2022. Association for Computing Machinery.