A Survey on Parallelism and Determinism

Laure Gonnord, Ludovic Henrio, Lionel Morel, Gabriel Radanne

▶ **To cite this version:**

Laure Gonnord, Ludovic Henrio, Lionel Morel, Gabriel Radanne. A Survey on Parallelism and Determinism. ACM Computing Surveys, 2022, 10.1145/3564529 . hal-03828497

HAL Id: hal-03828497
https://inria.hal.science/hal-03828497

Submitted on 25 Oct 2022

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.

A Survey on Parallelism and Determinism

LAURE GONNORD, LCIS (UGA, Grenoble INP), France and LIP (EnsL, UCBL, CNRS, Inria), France
LUDOVIC HENRIO, CNRS, EnsL, UCBL, Inria, LIP, France
LIONEL MOREL, Univ Lyon INSA de Lyon, CITI Lab, France
GABRIEL RADANNE, Inria, EnsL, UCBL, CNRS, LIP, France

Parallelism is often required for performance. In these situations an excess of non-determinism is harmful as it means the program can have several different behaviours or even different results. Even in domains such as high-performance computing where parallelism is crucial for performance, the computed value should be deterministic. Unfortunately, non-determinism in programs also allows dynamic scheduling of tasks, reacting to the first task that succeeds, cancelling tasks that cannot lead to a result, etc. Non-determinism is thus both a desired asset or an undesired property depending on the situation. In practice, it is often necessary to limit non-determinism and to identify precisely the sources of non-determinism in order to control what parts of a program are deterministic or not.

This survey takes the perspective of programming languages, and studies how programming models can ensure the determinism of parallel programs. This survey studies not only deterministic languages but also programming models that prevent one particularly demanding source of non-determinism: data races.

Our objective is to compare existing solutions to the following questions: How programming languages can help programmers write programs that run in a parallel manner without visible non-determinism? What programming paradigms ensure this kind of properties? We study these questions and discuss the merits and limitations of different approaches.

CCS Concepts: • **Theory of computation → Parallel computing models**; *Program schemes*; *Program verification*; • **Computing methodologies → Parallel programming languages**.

Additional Key Words and Phrases: Parallelism, Determinism, Data-Races, Programming Language Paradigms

ACM Reference Format:
Laure Gonnord, Ludovic Henrio, Lionel Morel, and Gabriel Radanne. 2022. A Survey on Parallelism and Determinism. *ACM Comput. Surv.* 55, 10, Article 210 (October 2022), 29 pages. https://doi.org/10.1145/3564529

Partially funded by French ANR CODAS project (ANR-17-CE23-0004-01).
Authors' addresses: Laure Gonnord, LCIS (UGA, Grenoble INP), 26902, VALENCE Cedex 09, France , LIP (EnsL, UCBL, CNRS, Inria), F-69342, LYON Cedex 07, France, laure.gonnord@esisar.grenoble-inp.fr; Ludovic Henrio, CNRS, EnsL, UCBL, Inria, LIP, F-69342, LYON Cedex 07, France, ludovic.henrio@cnrs.fr; Lionel Morel, Univ Lyon, INSA de Lyon, CITI Lab, EA3720, Villeurbanne, France, lionel.morel@insa-lyon.fr; Gabriel Radanne, Inria, EnsL, UCBL, CNRS, LIP, F-69342, LYON Cedex 07, France, gabriel.radanne@inria.fr.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Computing Surveys*, https://doi.org/10.1145/3564529.

1 INTRODUCTION AND SCOPE OF THE SURVEY

1.1 Parallelism: why and how?

In computer science, parallelism relates to the simultaneous execution of different tasks contributing to the same application. Parallelism can either be expressed in programs' source code or be introduced during their compilation or their execution.

An application may be parallel by nature because its functionality needs to be performed by several distinct computing units, for example if it gathers information originating from different geographical locations. Other applications are perfectly described in a sequential manner but parallelism can be introduced in the program, its compilation, or its execution in order to increase its overall efficiency or responsiveness. Programs can be split into different tasks that can run in parallel; these tasks can either fulfil different roles, e.g. computation vs. input/output, or handle different parts of the work to be performed.

Two very common abstractions proposed are *task-parallelism* and *data-parallelism* (see Definition 1)[1]. Task-parallelism consists in splitting the application's work into several tasks to be performed. These tasks can execute in parallel and well-specified events can be used to coordinate their execution. This programming paradigm is particularly adapted when the tasks are different and relatively complex. Data-parallelism comes from an identical treatment (e.g. the exact same function is used) being applied to different pieces of the input data. In that case, communication and synchronisation are used to ensure coherence of treatment over the different parts of the data treated in parallel. While the two models are expressive enough to express all kinds of parallel programs, programming a task-parallel problem as a data-parallel pattern or the contrary is cumbersome and inefficient.

1.2 Parallelism and Determinism

When tasks run in parallel, their effect may lead to a conflict, such as two threads trying to write data at the same time to the same memory location, or two threads trying to be scheduled concurrently. Such conflicting behaviours create non-determinism that can lead to different executions with different behaviours, and thus make the programming more error-prone and difficult to debug. Additionally, determinism properties can improve both execution support and debugging. For example, partial determinism properties of active objects have been successfully used to design recovery protocols in the context of fault-tolerance [BCDH07b], and to design deterministic replay techniques in the context of debugging support [TJS20a].

Let us consider the different behaviours of a program. According to the rewriting theory or the foundations of programming language semantics, a program (or a semantics) is deterministic if there is a single way to execute it and confluent if it can be run in different manners but all

[1]The fundamental terminology for understanding the survey has been highlighted and defined in pink boxes. Most of the definitions are "common knowledge" in the programming language community but some alternative definitions exist, this is why it is important to clearly state the definitions we use.

Definition 1. Task parallelism and data parallelism

Task parallelism: A parallelisation paradigm that consists in splitting the application into several tasks to be executed. Parallelism comes from the simultaneous execution of tasks.
Data parallelism: A parallelisation paradigm that consists in providing a single task to be performed on a set of (homogeneous) data. Parallelism comes from the fact that the data is split and different instances of the task treat the different data parts.

Definition 2. Races

Race-conditions: A *race-condition* occurs when two actions are possible and conflict in the sense that they lead to different global executions.
data races: A *data race* is a particular case of race-condition where the conflicting actions are the writing or reading of data at the same memory location.

executions (scheduled differently) globally have the same result. Monitoring a confluent program leads to different traces, which is not the case for a deterministic program. If the execution is also finite confluence means that all executions lead to the same final state. If the execution is not finite, confluence means that, from any two intermediate states reached by two partial executions, there is a way to schedule the rest of the executions so that they both reach a common (intermediate) state [BN98].

Because most of the works we study here use the words determinism and confluence interchangeably, in the following we use the term "**determinism**" for both concepts of rewriting theory: determinism and confluence. This terminology comes from the observation that both deterministic and confluent rewriting systems produce a deterministic result. Thus the difference between the two concepts is not useful for the programmer.

Non-deterministic programs exhibit *race-conditions*, see Definition 2. To tame the difficulty to write and analyse non-deterministic programs several concurrent programming languages limit the possibility to have races. The absence of data races is an interesting property because it ensures that non-determinism only originates from conflicting communications or synchronisations. Data-race freedom and determinism properties are often ensured at a language level, e.g. all programs using actors are data race free or all dataflow synchronous programs are deterministic.

We call *partial determinism* any property that identifies the conditions under which the execution of a program leads to a deterministic result. For example, based on the absence of data-races, some programming models are able to ensure partial determinism. One instance of such a partial determinism property has been shown in the context of ASP (Asynchronous Sequential Processes) [CHS04]: in an actor model with a FIFO service policy and futures with blocking reads, non-determinism only originates from two conflicting communications toward the same mailbox. More generally, several static analyses, compilation approaches, or runtime frameworks also ensure partial determinism properties.

In this paper we classify the approaches based on the "programming model" they rely on. We use programming model to designate the concepts manipulated by the programmer and their semantics. Because the programmer's main entry point to development is the programming model itself, we have chosen to focus our presentation on this criterion. Rather than a *language* vision, we consider *programming models* as a distinguishing factor in order to highlight the fact that important criteria for the choice of a language should not relate on syntactic details of a real programming language but more on the programming concepts that are exposed.

This paper is written with two types of readers in mind: *application programmers* that desire to pick the right approach with respect to their needs; and *language designers*, that wish to provide better safety guarantees or more parallelism in their languages. As opposed to previous surveys [ST98, Kes07] that mainly focus on cost models and performance, we focus on programming approaches that provide partial determinism properties. In other words, the focus of this survey is more on safety than on performance.

The rest of the paper is organised as follows. In Section 2, we describe the scope of our study: we focus on programming models and their safety properties regarding parallelism. The next two sections explore various parallel programming models of the literature, classified in two families: the deterministic ones in Section 3 and the data race free ones in Section 4. Finally, we summarise our study in Section 5 where we compare the different programming models, exhibiting how they can be decomposed in complementary building blocks, each of which serves more specific purposes leading to generally-speaking deterministic programs. 0

2 POSITIONING: LANGUAGE AND PROGRAM ANALYSIS FOR DETERMINISM

Deterministic parallelism can be addressed at different steps of the software life-cycle. These different steps are briefly described below, we also explain how each step can be used to produce efficient and correct parallel applications.

2.1 The Different Phases of the Software Life-cycle

Programming. Many programming languages or libraries let the programmer declare the intrinsic or potential parallelism of his/her application. Some primitives in the programming language are sometimes intended for the programmer to express this potentiality. It is then the responsibility of the programmer to state which parts of the program can be parallelised safely and efficiently.

Typing and Static Analysis. Type systems and static analyses serve two purposes. First, they reject programs for which there is no well-defined behaviour. Second they are used to extract properties about programs. These analyses can be used to check that the parallelism declared by the programmer is correct, or to extract potential parallelism, e.g. by inferring that two parts of the program can be separated into independent entities running in parallel.

Compilation. Modern hardware has so many characteristics that fine-grain optimisations are often intractable for the programmer. Compilers are often used to explore possible transformations and generate code that a human cannot imagine by herself. In particular, they deal with programming issues which should remain hidden to the programmer. In the field of domain specific languages, even more details need to be kept hidden from the programmer, either because she is not a developer or because we want her to focus on the modeling of specific artefacts such as the external environment (in reactive systems) or temporal properties (in critical systems).

Compilers are complex pieces of software that rely on a large set of static techniques to generate code. The efficiency of these analyses and transformation techniques is linked to the programming language and the static analyses, especially when it comes to parallelism extraction. For example, a language with strong parallel constructions needs less analysis effort to be compiled into optimised parallel code.

Runtime. Runtimes play a role whenever a decision such as code placement, code scheduling, or data distribution cannot be made statically or can only be performed with substantial over-approximations. For instance, these decisions may highly depend on dynamic data, or on hardware components shared with other programs running on the system. In this case, the runtime (be it the operating system or a language-specific interpreter or runtime, e.g. openMP's runtime) can implement such decisions.

Hardware. Modern hardware often itself performs clever optimisations like branch prediction. Every such optimisation is by essence more efficient than the corresponding software version. However, the complexity of modern architectures makes predicting the exact impact of these optimisations difficult, and making a clever usage of hardware characteristics is still a challenge for all the steps above.

Also, many hardware designs were proposed that could be naturally mapped to some parallelism-oriented programming models. In particular, dataflow processors [ŠRU99] are still extensively explored as a way to efficiently execute applications following this programming models. We believe these hardware designs represent a relatively small niche and we rather concentrate on programming models that are executed on the more classical Von Neumann-style processors.

2.2 Language, Compilation, and Program Analysis Approaches

Programming languages are not only designed to allow the programmer to express computations or programs efficiently. They also allow safety verification of programs: they let the programmer provide the right information to a static analysis and to the compiler to enable efficient and safe execution. In particular, high level programming abstractions give a high level point of view to the programmer; the important global information on programs is directly expressed by the programmer and does not have to be inferred through static analysis of lower-level constructs.

There are different approaches to express parallelisation in a program: classical full-fledged languages, domain specific languages, parallelisation libraries, pragmas, and annotations. These approaches have similar characteristics but with different guarantees, tools, and development costs.

A full-fledged language is a very flexible approach where the designer of the language can introduce rich constructs with powerful constraints on the program, ensuring by construction strong guarantees on the executed application. The drawback is that developing and maintaining a language costs a lot of time and energy. A frequently used solution to mitigate this cost is to compile to a low-level language, typically C [BCC+15a] or sometimes to richer languages (e.g., ABS features different backends including Haskell, Maude, or Erlang [JHS+11]), for which an efficient compiler already exists.

Domain specific languages (DSL) are languages that target a specific domain and generally compromise expressiveness for dedication through specific operators or programming structures. The DSL approach generally relies on a similar approach as full-fledged languages but allows the language developers to spend most of their energy in the efficient compilation of constructs that are specific to the targeted domain.

The previous approaches rely on the implementation of a compiler (implementing the semantics defined by the language, generating machine code that fits this semantics); in the case of parallelism, compilers can also reschedule, distribute, or merge parallel code. To avoid having to develop and maintain a specific compiler, one alternative is to rely on an existing language and extend it.

Developing libraries is the least intrusive way to do this. What can be expressed as a library is easier to maintain and a library may provide enough constructs to allow the programmer to express rich parallel programs. Writing complex libraries is easier when the host language provides enough information on the programs themselves. Indeed complex libraries often rely on meta-programming and reflection [Wya13, CDdCL06] (i.e. the ability to manipulate some information about the program itself). The main drawback of the library approach is that it often relies on "good practices" because the library itself is not able to enforce strong properties on the program, especially if the programmer uses features of the original language that conflict with the purpose of the library. For example, [TDJ13] analysed several Scala projects where programmers mixed the actor model with other types of concurrency in a generally unsafe manner, threatening the absence of data-races and the (partial) determinism properties of actors.

Finally, **pragmas and annotations** allow the programmer to attach information to an existing program. Some of this information can be used to parallelise the program, sometimes changing the program behaviour because of the concurrency introduced. Annotations can also be viewed as a way to control the amount of concurrency introduced. Annotations are the core principle of

the standard openMP approach [DM98a]. They have also been introduced, and particularly well integrated with the concepts of the language, in Jac, a Java framework for concurrency [HL06]. Pragmas differ from libraries in terms of expressiveness: the language expressed inside the annotations can be adapted to the target domain compared to the general-purpose host language, but the connection between elements of the host language and elements of the annotations might be complicated. Additionally, what can be expressed inside the annotation language needs to be designed carefully to reach the right level of expressiveness.

Complementarily to these different views on programming model approaches, static analyses can be designed in order to infer some properties of parallel programs such as absence of deadlock or race conditions [EA03, Min15, GHLM16]. These analyses can be performed on any type of language, from general purpose languages with explicit threads and mutexes, to languages that propose high-level constructions like futures.

All these approaches might be combined in many ways to provide the desired language features. In this survey, we focus on programming paradigms that provide at least data race freedom, the next section details what is inside the scope of this survey and what are the related works outside the strict scope of this survey.

2.3 Scope of the survey

In this survey, we take a programming language approach on the design of deterministic or data race free programs. Before getting into the details of partial determinism in the context of programming models, we summarise in this section alternatives approaches and surveys on related topics. This section thus focuses mostly on approaches that either do not ensure the absence of data races, or do not rely on the definition of a programming model.

The choice to take a programming model perspective on data race freedom is particularly relevant because, on the programming model side, there is a clear distinction between data race free and non-data race free languages. The focus of our survey is on programming models where data race freedom is part of the language philosophy. This means that the "obvious" code using these languages should be data race free, even if there might exist some out-of-the-way escape hatches or workaround to create data races. Similarly, we include in our survey a section on linear types (Section 4.5) because it is a language construct that by nature ensures data race freedom, even if there are languages with linear types that are not data race free. On the contrary, we do not cover languages where data race freedom is only achieved through alternative tools such as static analyses. We briefly describe these out-of-scope programming paradigms in the next paragraph.

A few classical parallel programming models. Historically, numerous parallel libraries have been designed to enhance the parallel and distributed capacities of C programs in addition to classic threads. First, the classical OpenMP library [DM98b] is composed of liberal constructs that do not prevent data races, even if `critical` and `atomic` directives might help to design safe programs. Nevertheless some attempts do exist to detect data races in OpenMP programs [HKTJ12]. The Cilk language [FHLLB09], a multithreaded parallel C-like language, similarly has data races which can be avoided using specially designed data structures called `hyperobjects`. X10 [CGS+05] is a similar Java-based language where static analysis is necessary to detect tasks that can occur in parallel and cause data races [ABSS07]. The MPI library [CGH94] aims at providing communication and distribution primitives for programs to be executed on large clusters. Its message-passing based approach does indeed helps the programmer to design data race free programs. We however exclude it since, compared to the languages we study here, data race freedom relies more on the coding practises than on the language itself. For instance, MPI communications can be made efficient by

using Remote Memory Access, which mandates the use of static analysis to ensure the absence of data races [ASS+21].

Transactional Memory. Transactional Memory is a *runtime* approach to provide optimistic lock-free constructs to access and modify memory simultaneously. Each agent wanting to access the shared memory locations will attempt to execute and commit a transaction. One will succeed and apply its operations, while the others will abort and may retry. Transactions were originally intended to use dedicated hardware support [MHM93], but were soon implemented in software [ST95] and now support a wide range of techniques and properties.

Transactional memory approaches provide limited control over interleaving, by ensuring transactions behave similarly to critical sections, while still allowing some runtime concurrency for efficiency. This relies on some relaxation of determinism properties. For instance, these approaches can provide various properties [HW90] such as "obstruction-freedom" (absence of deadlock but allow livelocks) or "lock-freedom" (absence of both deadlocks and livelocks). Concretely, they provide neither full determinism nor data race freedom for the whole language, but rather rely on *opacity* [GK08] which means that from an external view point, different transactions should behave as if they had been executed sequentially and always have a consistent view of the memory. As a consequence, if the programmer encapsulates every critical section inside transactions, no data-races are possible.

Related surveys. A first related survey can be found in [CZG+15] concerning deterministic replay, which is very useful for debugging or fault-tolerance. Replay techniques enforce one execution of a program to be identical to a previously recorded execution. These techniques thus mostly apply to programming languages that are not by nature deterministic, potentially with data races and other sources of non-determinism. On the contrary, we focus here on approaches that allow the programmer to write (almost) deterministic programs directly, thus easing their understanding and debugging. Note that partial determinism properties of programming models can be used to design efficient deterministic replay techniques as less information needs to be logged to characterise an execution [CH05]. In addition, a significant part of the survey [CZG+15] is placed on lower-level aspects (operating systems, hardware, etc.) which are significantly different from the language point of view we adopt. For instance it studies deterministic replay in presence of weak memory models or data races, which is clearly out of our scope here.

Similarly, thread-based speculation techniques [ELGE16], allow running originally sequential piece of code in parallel. Dependence violations are detected at runtime (either with hardware of software mechanisms) and roll-back mechanisms are used to replay the sequential code if required. These techniques can thus be viewed as a way to increase parallelism for deterministic programs without introducing non-determinism. Even if they can be complemented by language mechanisms that enables dependencies to be efficiently computed, we consider these approaches out of scope.

Finally, other surveys about programming parallel machines have been published in the past. The present article can be viewed as a follow-up to the survey [Kes07] about models of computations: our "programming model" focus benefits from more recent contributions that have transformed the "general parallel programming methodologies" described in the 2007's paper into what we call "programming models/languages features". More recently, [DMCN12] present a rather thorough survey on programming languages covering pure parallel models (OpenMP and MPI), heterogeneous models (CUDA, OpenCL), Partitioned Global Address Space models, as well as models mixing several of these approaches. Compared to [DMCN12], our survey has a narrower focus, especially on guarantees brought by programming models about determinism.

2.4 How Programming Models Limit Non-determinism

From the study of several deterministic and partially deterministic languages, we identified the main solutions that exist to limit the non-determinism of parallel programs. The main part of this survey will present the languages that we considered as particularly interesting in this context. As stated above, in many cases, non-determinism is an undesired property, and consequently several languages provide abstractions restricting or forbidding non-determinism behaviours. In the sequel, we call "action" every single possible transition of our programs (a "step" in the terminology of operational semantics, or an "event" in the distributed system terminology). Actions can be for example a data read or write, a communication through a message, a thread creation. Two actions are *racy* if they are both enabled at the same instant of execution, and the observable remaining behaviour of the system is different depending on which of the two actions is executed first. The fact that we are interested in the observable differences introduces a notion of equivalence where we only consider the visible result of the execution. A language is called *deterministic* if there cannot be any racy action. A racy action can be formally identified by studying, for example, the operational semantics of the language.

As we are interested in solutions for parallelising an application while producing a deterministic result, these solutions should thus avoid racy actions. These races between actions can be characterised as:

Data races Two actions that read or write to the same memory location can be racy if one of the two is a write. Programming safely with data races is difficult and data races are avoided in most of the partially deterministic programming models. Additionally, in the presence of weak memory models [AG96], the impact of data races is often more difficult to predict. All the programming models presented in this survey will at least strongly limit data races thanks to additional constraints such as single assignment (Definition 3), blocking read (Definition 4), or partition of the memory. Because data access is well-protected, the programming models described in this survey are not sensitive to the reordering of memory accesses. This is why weak memory models [AG96] are almost absent in the state-of-the-art of the models presented here.

Synchronisation and communication races When two actions that synchronise two threads can be executed at the same time and are exclusive, like a lock release or the execution of a critical section by two entities, a race occurs.
Similarly, if the same channel can be used by two different message sending actions, or two different message reception actions then these two actions constitute a racy condition. Communication races can be arbitrarily complex depending on the complexity of the communication patterns. Communication races can be limited by providing constraints on the type of communication or their schedule, e.g. by imposing FIFO communication over channels (Definition 5).

The categories cited above are not disjoint, in particular some actions can be considered both as memory access and communication, or both as synchronisation and memory access. This is the case for example to shared memory accesses in distributed systems; whose concurrent reads may be considered as data or communication races.

The languages we study in this survey limit the potential races in different ways. These solutions can be classified as follows:

- *Limiting the semantics* so that no conflicting actions can be observed as in synchronous languages or in purely functional languages;
- *Limiting the interactions between computing entities* to prevent races as in actors, or BSP (Bulk Synchronous Parallel) [Val90] which prevents races between computation and communication (in two isolated phases);

Definition 3. Single Assignment

One way to get rid of write/write conflicts is to forbid multiple writes to the same (memory) location: such a pattern is called "Single Assignment" [RWZ88, CFR+91].

Definition 4. Blocking Read

A read to a location is said to be *blocking* if the reading process is suspended until this part of memory actually contains (updated) data.

Definition 5. FIFO channel

A *FIFO channel* (also refered to as *FIFO queue*) is a data buffer with exactly *one* producer and *one* consumer, where data tokens are forwarded to the consumer in the exact same order they have been emitted by the producer. A FIFO channel can hold a varying number of elements, and these elements are given exactly *once* to the consumer. Reading a FIFO is *blocking* (see Definition 4): when the consumer tries to read an empty FIFO, it is suspended until at least one data element is emitted by the producer. FIFO channels are usually *single assignment* (see Definition 3) while writing (they forbid multiple rights to the same cell). Depending on the programming model, FIFO buffers can be considered to hold an unlimited number of data tokens. Emitting on a FIFO channel is most often a non-blocking action.
Note that an *actor* (see Section 4.2) may be equipped with FIFO *mailbox*es. Such a mailbox is not a channel *per se* because it supports several producers.

- *Providing communication and synchronisation tools* that are *by nature data race-free* like futures or Single Assignment;
- *Restricting communication scheduling* to limit the set of potential behaviours and prevent some communication races (FIFO channels / mailboxes, see Definition 5);

3 PROGRAMMING MODELS WITH DETERMINISM GUARANTEES BY CONSTRUCTION

Several programming models can somehow reach the goal of combining parallelism and determinism (with some restrictions on the form of parallelism), they are presented in this section. Many of the programming models presented in this section are based on synchronous or dataflow programming models.

3.1 Kahn Process Networks

The first dataflow programming models (see Definition 6) were proposed in the seminal work of G. Kahn [Kah74]. Those constructs were later called KPNs, for Kahn Process Networks. At the heart of KPNs is the idea to decompose the application as a set of independent processes or *agents* that communicate solely through First-In-First-Out (FIFO) channels.

When a KPN agent is scheduled for execution, there is essentially no way to know how many data elements it will read/write from/to its input/output channels. As a consequence, buffer sizes as well as scheduling of agent's activations cannot be determined statically. Also, no guarantee can be given that a KPN program can be executed with finite memory. It is therefore the task of a dynamic scheduler to attend to these questions. As an example, consider a video encoder that

Definition 6. Dataflow

A *dataflow* programming model [KM66, Kah74] is a model where a program is decomposed into actors that solely communicate through FIFO channels.

processes constant-size blocks of raw images and produces compressed encodings corresponding to these blocks. Statically, there is no way to know the size of blocks produced by the encoder as it essentially depends on the content of each block in the input raw image. While the global correctness is easy to establish, the size of the communication channels cannot be predicted statically. In some contexts, like critical control systems, this is unacceptable, as static guarantees on the necessary memory are needed to ensure the system's safety. However, in many contexts, KPNs offer the right level of expressiveness because of the simple and deterministic parallelism they provide.

To address the question of static predictability of boundedness, many variant models and languages have been studied. They focus on the ability to determine at compile-time how many data tokens are exchanged by agents over FIFO channels. These *static dataflow programming* models are quickly reviewed now.

Strong Points for KPN:

+ Very fertile ideas that have been used as inspiration for many parallel languages.
+ Simple semantics, and simple syntactic requirements to ensure determinism.
− Restricted programming patterns and parallelism.
− Unpredictable size of channels.

3.2 Static Dataflow Programming

In Static DataFlow Programming (SDF in short) [LM87], agents communicate through channels. The programmer gives, for each agent, the (static) number of tokens it reads (resp. writes) on each of its input (resp. output) channels, each time it is activated.

Static dataflow programming models focus on properties necessary to ensure boundedness of memory usage as well as static schedulability. In fact, the problem of statically scheduling agents is shown to be decidable, for sequential as well as for parallel (homogeneous) processors (with maximum parallelism).

The ΣC language [DLSD12, DLC14], which implements the Cyclo-Static Dataflow model [BELP96] where data rates of agents are known statically and can change periodically, has also the same property. Depending on the target architecture, FIFO communication channels are compiled into efficient shared-memory implementations or into distributed communication mechanisms. The ΣC runtime is then in charge of allocating memory regions and scheduling the activities corresponding to agents, either relying on an underlying operating system or through dedicated scheduling policies.

Parallelisation of Static Dataflow programs has been well studied for the StreamIT language [Thi09, Gor10]. The StreamIT compiler and runtime manages both data- and task- as well as *pipeline*-parallelism. Also, StreamIT distinguishes explicitly between stateless agents, which can be trivially duplicated for data-parallelisation and stateful actors, that embed a state, which makes parallelisation possible in certain cases but much trickier [SHGW12].

Definition 7. Synchronous Language

A *synchronous language* [HCRP91, LTL03, Ber00] is a language dedicated to programming *reactive systems*.
It enforces a logical vision of time where on each tick of a *clock*, the program receives new values for all or part of its inputs, processes these values and generates new values for all or part of its outputs. These output values are made available before the next tick of the clock.

Strong Points for SDF:
+ Determinism inherited from the KPN model.
+ Static scheduling and statically bounded channels.
− Requires static information on communication rates given by the programmer.
− Restricted parallelism patterns (similar to KPN).

3.3 Synchronous Languages

Synchronous languages (See Definition 7) like Lustre [Hal05] have been proposed in the early 80s for the reliable development of safety critical embedded systems. In Lustre, a program is described as a composition of components that communicate through data flows. Each component is a function over streams of data tokens. Computations and communications are decoupled from one another: whenever a component is "activated" (i.e. scheduled), all its inputs are available. It then produces all its outputs immediately to its output environment.

From the global system's point of view, the relations between components through their communication channels yield a partial occurrence ordering of all observed events: while focusing on a node, all its local events (used to compute its outputs) are totally ordered with respect to its abstract clock. The language is deterministic, and the sequential compilation process of synchronous language [HCRP91] generates code that is a correct sequentialisation of every computation of every node.

Parallel execution of Lustre programs. Overall, while Lustre expresses deterministic execution as a set of independent entities, the efficient parallel execution of Lustre programs is still a challenging task. Indeed, as stated in the survey [Gir05], the problem is not to exhibit the greatest possible parallelism in the synchronous source program, but rather to deploy it onto a given (distributed/parallel) architecture, with some fixed level of parallelism. Approaches that address this problem include replication of the control and/or partitioning algorithms to distribute tasks [CG95, BS00, GM02]. All these algorithms and approaches, even the most sophisticated ones, still guarantee the preservation of dependencies and the determinism of the overall application. In the context of hard-real time critical systems, the work presented in [CSST08] proposes a completely safe multitask implementation of Lustre parallel programs that is compatible with well known real-time scheduling algorithms like fixed-priority or earliest-deadline first: the authors of [CSST08] prove the preservation of synchronous semantics by implementing communication channels as buffers with concurrent accesses.

Introducing asynchrony into synchronous languages . Alternatively to the previous approaches whose common point is to preserve synchronicity as much as possible, the authors of [CGP12] suggest to extend Lustre with futures (see Definition 8), providing more asynchronism without

Definition 8. Future

A *future* [BJH77, Hal85, NSS06] is an entity representing the result of an ongoing computation. It is used to launch a sub-task in parallel with the current task and later retrieve the result computed by the sub-task.
By nature, futures are single-assignment entities that can support multiple readers. Several semantics for synchronising on the future resolution exist but if only blocking read is used then futures do not introduce non-determinism [CHS04].

losing determinism. This allows easier expression of non regular synchronisation, or fork/join programming models that are tedious in standard Lustre, albeit essential idioms for elegantly writing certain programs. This is performed by the combination of a keyword *asynch* that triggers asynchronous computations, and explicit blocking synchronisation on the future that corresponds to such asynchronous function executions. The communication between the task that launches the *asynch* and the *asynch*ronously called one is done by a bounded FIFO channel with a statically predefined size. Each of these entities can also be multi-threaded to have several processes perform the same task in parallel, but only if the task is stateless (see Section 3.2). The approach is implemented as a fork of Heptagon [DRM13], a language very similar to Lustre. It is interesting to notice that the notion of future is also massively used in actors (see Section 4.2) and that the runtime used to execute Lustre programs with futures is similar to an actor system.

Some other variants of Lustre have proposed extensions or restrictions of clocks (e.g., the integer clocks of [GGPP12]) that enable efficient and proven determinism at runtime. Prelude [CBF+11], one of the most mature of these variants, has a compilation and scheduling process that guarantees the determinism of computations that are eventually executed in their runtime. These solutions mostly emerged in the context of HPC where the strict synchronous vision is too restrictive, as many applications need relaxed forms of synchronisation.

Strong Points for Synchronous Languages:

+ The programming model is a good fit for certain applications, notably embedded reactive systems.
+ Static inference of scheduling and communications.
± Tailored for task parallelism.
− Parallel efficiency is difficult to obtain.

3.4 Polyhedral Model

The principle of the polyhedral model is to rely on dependence analysis in order to parallelise the execution of a (sequential) program that is automatically scheduled in a correct manner (from a data dependency point of view). The programmer thus writes a sequential program.

Because it features no parallelism, sequential programming is the obvious deterministic programming model, and the polyhedral "model" (framework) is one of the most elegant approach to execute a sequential program in parallel. The ideas of the framework date back in the late 70s, when the need for more performance in physical simulations (weather forecast, physics simulations for various applications) has crucially pushed for efforts to take advantage of hardware parallelism. The contribution on computing dependencies, by Karp, Miller and Winograd [KMW67],

the efforts on systolic arrays [BK84], as well as the work on loop transformations initiated by Lamport [Lam74] made the foundations necessary to the emergence of the whole framework in the early 90s.

With the polyhedral model, iterations of compute-intensive loop kernels are abstracted away as integral points satisfying affine constraints, namely, *polyhedra*. This framework [Fea92a] provides exact dependence analysis information where statement instances (i.e., statements executed at different loop iterations) and array elements are distinguished. The exact dependence information and the use of linear programming techniques to explore the space of legal schedules [Fea92b] is what constitutes the basis of the polyhedral model for loop transformations.

The traditional use of polyhedral techniques in optimising compilers typically focuses on loop transformations of *polyhedral kernels*. PLuTo [BHRS08] is now widely used as a push-button tool for automatically parallelising polyhedral loop nests. It tries to optimise locality in addition to parallelisation. Polyhedral techniques for loop transformations are now adopted by many production level compilers, such as GCC, IBM XL, and LLVM. There is also significant work in data layout optimisation for polyhedral programs where analyses are performed to minimise the program's memory requirement [DSV05]. Recently, more modular approaches [DTZD08, ZNS11, AP21] have emerged that promote the generation of intermediate dataflow models. With these approaches, the extracted parallelism is represented as *process networks* that communicate through FIFOs. This enables further optimisations to be performed in a modular manner.

As for properties, all polyhedral techniques have the same *guarantees by construction*: the programs obtained after transformation compute the same result (i.e. have the same semantics) than the one from the input sequential program. Only valid programs are generated, and the dependences of computation are preserved. However, to be effective, this theoretical result strongly relies on safe implementation of communication patterns [LGCP13, AP21]. Finally, recent works such as [BYR+11, YFRS13] propose a safety analysis of transformed parallel programs that guarantees the absence of races. Beyond the *all-static* approach of parallelising compilers, with a particular focus on new applications such as neural networks [BRR+19], some hybrid approaches have also been proposed to enhance the expressive power of the polyhedral model. Among these recent works we can cite the sparse polyhedral model [MYC+19] and loop speculation [LMJC20].

Strong Points for Polyhedral Model:

+ Programming model is a good fit for HPC kernels.
+ Efficient execution obtained quasi-automatically.
± Tailored for data parallelism.
± The programmer has many parameters to play with to control optimisations.
− Only applicable to well-formed computation kernels.

In this section we have seen several programming models that achieve determinism while allowing for parallel execution. To achieve this, the only solution is to prevent concurrency between conflicting effects. We have seen several techniques that allow parallelism without non-determinism: specific communication patterns like FIFO channels with blocking read on channels; specific semantics that deals with time in particular ways, like synchronous semantics; compilation techniques that generate parallelism for sequential programs, like the polyhedral model.

4 DATA RACE FREE PROGRAMMING MODELS

The programming models of the preceding section had strong properties but had a restricted applicability in the sense that they require a specific semantics that is sometimes far from the one of the

mainstream programming languages. When aiming at a broader field of application, one solution is to rely on languages that limits the sources of non-determinism. The most adopted solution that allows non-determinism in a controlled way is to only allow races concerning communications or synchronisations but to prevent data races. In general, data races can lead to complex bugs and it is a good programming practice to avoid them. We review here the programming models that natively prevent them. In these languages, first the programmer does not have to worry about potential data races; and second some partial determinism properties can often be identified because non-determinism can only originate from some well-identified races between synchronisations or communications.

4.1 Dataflow Programming Models that are not Fully Deterministic: DPN and CAL

Dataflow Processes Networks (DPNs) were introduced by Dennis [Den74]. The CAL Actor Language [EJ03] is an implementation of the DPN model. It proposes a structure of actors that perform a sequence of steps, consuming tokens from input ports, computing, and then producing tokens onto output ports. Actors can be parameterised, which simplifies the design of programs while avoiding to duplicate code. A set of CAL actors is compiled to a set of sequential functions which are scheduled dynamically by a dedicated runtime.

CAL is not a deterministic language and according to the authors of the Caltrop specification [EJ01], an actor is specified by *firing rules* that possibly involve non-deterministic choice among the possible reactions. Some of these ambiguities are solved by the compiler while others are left to the scheduler's choice. For example, a programmer can build non-deterministic CAL actors by using overlapping conditions in case/switch statements. However, the language was designed to enable tools to identify possible sources of non-determinism and warn the programmer about them. Conflicting firing rules can be identified and dataflow between actors can be analysed to ensure progress and/or determinism. If non-determinism is required by the programmer, the language provides a non-deterministic choice, whose resolution is left to the responsibility of the programmer. Finally, non-determinism as introduced by the use of a dynamic scheduler can also be reduced by identifying deterministic schedules at compile time, using a model-checker as in [ERJ+11].

Lohstroh et al. [LL19] recently proposed a deterministic actor language. The key ingredient for determinism is the *logical timing* of messages based on a protocol which combines physical and logical timing to ensure determinism. The solution relies on the tagging of messages with a timestamp and determination of an arbitrary (deterministic) order for identical timestamps. To the best of our knowledge, there is no proof of correctness of the scheduling protocol.

There is a significant adoption of dataflow programming models in the industry. One of the most prominent example is TensorFlow [ABC+16], an efficient and flexible framework for machine learning based on stateful dataflow graphs. Despite the dataflow inspiration, there is no determinism result for TensorFlow and more generally no high-level mechanism to restrict race-conditions; programming artefacts equivalent to mutexes have to be used to restrict non-determinism [AIM17].

Strong Points for DPN:

+ Generic programming model which fits many use-cases (ML, Signal Processing, HPC, ...).
+ Deterministic variants of DPN can be designed (e.g. [LL19] that relies on timestamps and arbitrary deterministic ordering to provide deterministic reactors).
± Tailored for task parallelism.
− Non-determinism needs to be skimmed away by programmer.
− Static analysis needed to guarantee determinism, deadlock-freeness, etc.

In the setting of CAL, each computing entity is sometimes called an actor. However, actors were already introduced back in the eighties [Agh86, Wya13] for a relatively similar notion. The main difference is that the actors in CAL are associated with a dataflow semantics, while more traditional actors, presented in the following section, are based on a sequential semantics.

4.2 Actors and Active objects

An actor [Agh86, Wya13] is a mono-threaded entity that communicates with others by asynchronous message sending. In particular two actors cannot access the same memory location, which prevents data races. Active objects [BSH+17] unify the notions of actor and object (in the sense used in object-oriented programming), as they give to each actor an object type and replace message passing by *asynchronous method invocation*: an active object communicates by calling methods on other active objects, asynchronously.

In active object languages, the result of an asynchronous method call is a future [TMY94] (see Definition 8). An actor program is generally designed quite similarly to a sequential program except message sending that triggers asynchronous computations. When a reply to a message is expected, either the programmer must implement an explicit callback or rely on a future if possible. The internal behaviour of an actor is deterministic, and the only source of non-determinism is either the concurrent sending of messages to the same destination, or a non-deterministic scheduling of message treatment (e.g. if the mailbox is not FIFO). The absence of multi-threading inside an actor and the fact that each data is handled by a single actor prevents data races.

ProActive is one of the first full-fledged active object languages. It uses active objects to implement a Java library made of distributed objects that communicate asynchronously. The ASP (Asynchronous Sequential Processes) calculus [CHS04, CH05] formalises the ProActive framework. It has been used to prove partial determinism properties that can be summarised as follows 1) *If the dependence graph between active objects is a tree then the execution is deterministic*; 2) *The result of a program execution is entirely determined by the ordered list of the identifiers of the caller in the request queue of each active object.* The work on ASP also relates deterministic active objects to Kahn process networks [CH05]. The partial determinism of ASP comes from the actor model but also the FIFO service of requests in the model. Interestingly, this result somehow show that futures do not compromise the determinism properties of programs provided they are accessed in a blocking manner; somehow justifying the fact that similar futures have been successfully introduced in Lustre without loosing determinism. The determinism properties of ProActive have been used to design a fault-tolerance protocol for active objects [BCDH07a]. More recently [HJP20], these determinism properties have been revisited in a more general setting, enabling a restricted form of cooperative scheduling, a type system ensuring determinism for active objects have also been designed.

AmbientTalk [DCMM06] was created based on the E Programming Language [MTS05] which implements an actor model with a communicating event-loop. It targets embedded systems and

uses asynchronous future access (see below). Asynchronous reaction on future resolution and more generally on events is a prominent source of non-determinism in AmbientTalk.

Creol [JOA03, JO07] and the languages that inherit from it, JCobox [SPH10], ABS [JHS+11], and Encore [BCC+15a], rely on *cooperative scheduling* allowing the single execution thread of the active object to interrupt the service of one request and start (or recover) another one, but only at program points specified by the programmer. This approach clearly introduces more non-determinism than ASP for two reasons: 1) Cooperative scheduling introduces interruption points, generally depending on external events like the resolution of a future, but they are placed by the programmer who can control the sources of non-determinism. 2) There is no predefined order on the scheduling of pending tasks and the scheduler is thus by nature non-deterministic. More recently, partial determinism properties of ABS have been exploited to reproduce a given execution [TJS20b]; the approach is based on the recording of non-deterministic events that include request service and local scheduling. The fact that in active objects the set of non-deterministic decision in the scheduling of an active object is very small makes this approach feasible and scalable.

In a more industrial setting, Akka [HO09, Wya13] is a scalable library for actors on top of Java and Scala. The Akka documentation encourages programmers to use asynchronous reaction on futures. This introduces a significant source of non-determinism, suggesting that (partial) determinism is not one of the objectives of the language. Software transactional memory has also been combined with actors in the context of Akka to perform speculative computation and improve performance while ensuring the absence of data races in such actors [HSH+13].

Strong Points for Actors and Active Objects:

+ Mesh particularly well with object-oriented languages.
+ Easy to use, and distributed as library in mainstream languages.
± Tailored for task parallelism.
± Can scale well with sufficient engineering efforts.
− Race conditions exist, in particular due to communications.

4.3 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) [Val90] targets data-parallelism where parallelism appears when the same task is performed on different data. BSP is a parallel execution model that defines algorithms as a sequence of supersteps, each made of three phases: computation, communication, and synchronisation. BSP is adapted to the programming of data-parallel applications and requires the strong synchronisation of all computing entities. A BSP program is typically made of a piece of code that is a step of computation and terminates by a synchronisation operation that coordinates all processes. Interactions between BSP processes occur through communication primitives sending messages or performing one-sided Remote Direct Memory Access (RDMA) operations.

While task parallelism, addressed for example by actors, is more general because it is always possible to split data and then launch different tasks, data-parallel programming models are more specialised and can run programs much more efficiently provided the application is inherently data-parallel. Typically, BSP allows the programmer to easily write a program that performs a heavy computation on a huge matrix and run it efficiently on high-performance computers, provided the computation can be split into independent treatments on parts of the matrix.

BSP features predictable performance and absence of deadlocks under simple hypotheses. The fact that computation is separated into independent entities makes it very easy to design deterministic algorithms in BSP. Only concurrent writes on the same memory address, in the implementations of BSP that support it, can be a source of non-determinism. Recently, [HHLS18] integrated BSP and actors, leading to a model that features both task and data parallelism with limited non-determinism. Additionally, it is possible to design a static analysis that checks deadlock-freedom for BSP programs [JDB+17, Dab18].

Compared to the races that can appear in active objects, the way concurrent communication is handled in BSP is very different: in active objects there is no data race but communication races are inherent to the model and intertwined with computations. In BSP, the computation part is completely race-free and races that appear during communication steps can always be made deterministic, e.g. by prioritising a process. As the synchronisation step involves no computation and is bounded, a reordering of communications is always safe (but might induce delays in the synchronisation step).

There is a quite wide adoption of the BSP programming model in the industry. For example, Pregel is a graph processing framework adapted to computation over large graphs inspired by BSP and developed by Google [MAB+10].

Strong Points for BSP:

+ Simple to learn for sequential programmers.
+ Easy to ensure deadlock freedom and other properties, by analysis or by restricting the language.
+ Has a cost model.
± Tailored for coarse-grained data parallelism.
− Limited programming model.

4.4 Algorithmic Skeletons

Algorithmic Skeletons [GVL10] are a high-level parallel programming model introduced by Cole [Col91]. Skeletons express parallel composition patterns through a composition language that assembles basic (sequential) blocks. A pattern is a high-level construct that composes some operations and may introduce parallelism. The pattern can provide data-parallelism when using a farm pattern that instantiates several instances of the same skeleton to work on several independent pieces of data, or task-parallelism when instantiating a pipeline where each stage runs in parallel (on a different input). Composition of skeletons is realised by using a set of classical parallel programming patterns proposed to the programmer, typically *map*, *reduce*, *pipeline*, *farm*, *divide and conquer*, etc.

The expressive power of algorithmic skeletons comes from the variety of patterns proposed to the programmer, while their efficiency is obtained by specific scheduling techniques used by the execution environment. The Map-reduce pattern [BDL10] is one of the most massively used instances of algorithmic skeletons, it is dedicated to data-intensive computations and is successful because of the simplicity of its usage and the efficiency of its implementations. Dynamic and speculative versions of polyhedral-model based optimisation use dynamic information to chose between skeleton patterns for final code generation [SR15].

While there is no focus on determinism properties, most instances of algorithmic skeleton programs are by nature deterministic because each sequential skeleton is independent and in general the composition operation preserves determinism. Basically non-determinism could appear during

some composition operations that gather results coming from different computations. For example the reduce phase of a Map-reduce computation could be dependent on the order of production of the gathered results. But very often reduction phases are programmed so that they are not sensitive to the order of computation and skeleton programs are often deterministic in practice. Overall, ensuring determinism only requires to ensure that some operations are commutative. To the best of our knowledge no formal specification of determinism properties or no deterministic skeleton runtime has been proposed yet.

Calcium [CL07] is an algorithmic skeleton library implemented over the active object library ProActive, enabling a large-scale distributed execution of a variety of skeletons. The separation into actors provided by the active object pattern naturally fits well with the notion of independent, hierarchically organised skeletons. No property of determinism has been identified in this context despite the use of the ProActive library that features partial determinism properties

Strong Points for Algorithmic Skeletons:

+ Excellent composition.
+ High-level/good optimisation opportunities.
– Very limited programming model.

4.5 Linear Types

Linear types are a language feature enhancing the capabilities of the underlying language through a richer type system. Linear types provide efficient low level control over copy and aliases [Wal02] and data race freedom *by construction*. For this purpose, linear types enforce "linear" uses of resources, where resources must always be used exactly once. These resources can therefore not be shared or copied by other functions, restricting concurrent accesses and data races, and preventing various programming errors such as use-after-free. For instance, if a database handle is used linearly, it cannot be shared between different functions, and thus cannot be accessed concurrently. Since it cannot be shared, when the database handle is closed it cannot be used any more.

This initial idea has been expanded to numerous contexts. In functional programming, it birthed two families of type systems, linearity-by-functions [Wal02, BBN+18] where functions must themselves use their argument linearly and linearity-by-kinds [MZZ10, TP11, Mor16] where resources are given a usage mode which might be linear. Finally, unique types [BS95] enforce that resources have a single reference to them which enables aggressive optimisations for purely functional parallel languages [HSE+17] while preserving data race freedom. Several systems provide formal proofs of non-concurrent accesses for resources used linearly [Wal02, RST20].

To handle imperative programming with mutable states, linear types have been extended with complementary notions such as regions [LG88, JAD+09] and ownership [BR05]. Regions allow to delimit an area of memory that functions are allowed to use. Parallelism can then be limited to operations on distinct regions, thus preventing aliasing. Ownership relaxes this notion by allowing a function to temporarily "borrow" a resource, which can be used in limited ways (for instance, it cannot free it). This concept has been widely used in Rust [MKI14], along with other imperative and functional languages [GMJ+02, RST20]. [WPMA19, JJKD18] provides formal proofs that Rust ensures data race freedom for linear resources.

Finally, capabilities [BNR01] and typestates [DF01, ASSS09] combine linearity with object-oriented programming to precisely control what objects can or can't do, such as "this channel can send messages now". They have notably been used in parallel languages with actors such as Encore [BCC+15b] to prevent data races.

In all these cases, linear types serve as support for the underlying programming model by encoding additional checks in the type system. This allows to statically check numerous properties for safety (absence of data races or race-conditions), efficiency (additional no-aliasing guarantees), or to adapt to different use-cases, e.g. for data-parallelism [Teab], task-parallelism [Teaa], or lock-free data-structures [CW17]. The flip slide is that such checks need to be encoded in the discipline of type systems, which requires great care by the programming language designers, and occasionally exposes programmers to concepts difficult to understand.

Strong Points for Linear Types:

+ Enable as much control as C with POSIX Threads, with additional safety.
+ Feedback on potential races through typing errors.
+ Composes well with other paradigms and multipurposes languages.
− Steep learning curve.
− Safety properties requires careful design of the compilers.

5 DISCUSSION

5.1 Summary: How to Limit Non-determinism?

This section provides a general discussion over the models we have presented earlier, and identifies how they help dealing with races and limit non-determinism. We identify the following mechanisms used to obtain determinism, which can be formulated as questions of the form: "Does (and if yes, how?) the approach ...:

(1) Separate computation from communication;
(2) Rely on sequential computation as much as possible;
(3) Limit the potential interleaving of communications (or synchronisations);
(4) Protect or prevent access to shared data."

The goal of this section is neither to compare the various techniques nor the programming models, but rather to present the landscape of approaches in broad strokes and to hint at gaps in the existing ecosystems. It also aims to provide a guide to adequately choose among programming constructs and models with respect to the desired guarantees.

The rest of this section summarises the approaches we studied in this survey and identifies the techniques they use to address the four questions. Table 1 provides a "bird's eye view" of this summary, organised by properties and approaches.

5.1.1 Separate Computation from Communication. The first mechanism we identify to limit non-determinism is the ability the programmer has to specify the interplay between computations and communications.

A first possibility is for the programmer herself to describe in a *fine-grained* manner how communications are interleaved with computations. The most extreme examples are manual use of threads, as well as linear types. KPN, DPN, and actor models all allow the programmer to mix computations and communications in a fine-grained fashion, albeit differently from one another. In KPN, reads are blocking and all communication channels are point-to-point, which is a combination of properties that guarantees determinism effectively. In DPN and actor models, communication, performed through multiple writer mailboxes do not interrupt computations, which provides partial determinism. The races between different messages is a source of non-determinism but it also brings flexibility and reactivity.

Other approaches use *coarse-grained* communications, typically at the frontier of intersteps. In these approaches, no communication can occur while the program is performing computations on data, and all data are supposedly available when the computation starts. This is typically the case for BSP, where the communications are written down by the programmer. This is also the case for synchronous dataflow languages, where the programmer only describes data dependencies which are then translated by the compiler into supersteps of the form "get input data / compute / output data".

Finally, some approaches derive communication *automatically*, without explicit knowledge provided by the programmer. In the polyhedral model and in algorithmic skeletons, no information about data exchanged by threads is ever given by the programmer. All communications and synchronisations are inferred from the program's data dependencies.

5.1.2 Rely on Sequential Computation as Much as Possible. The second identified mechanism to limit non-determinism is to actually have the programmer write as much of her program's behaviour in a sequential manner. Each programming model can be characterised by a *sequential quantum* which varies depending on the desired property of the programming model.

First, the quantum can span the full program, like in polyhedral model, letting the compiler parallelise it, with the help of annotations.

In many other approaches, the quantum is identified by programming constructs. For instance, BSP programs are segmented into "supersteps", each of which contains sequential code. In algorithmic skeletons, sequential code is contained in each computation function (also called muscles). In actor models, DPN or KPN, sequential code is contained in entities whose granularity is left to the programmer. Such programming constructs can also be implicit, such as regions for linear types.

Finally, in synchronous dataflow languages, the actions described inside a block are individually sequential. Any of them can be fired concurrently during the execution of a block. Non-determinism is handled by sequentializing the description at compile-time.

When "sequential quantum" are identified, non-determinism can be limited by controlling the way these quantum interact and can be interleaved. We first review how to control interleaving of messages below before focusing on sharing data in the following section.

5.1.3 Limit Interleaving of Communications. The more communications are constrained, the less variability is possible in terms of scheduling, hence the less non-determinism will appear. At one extreme, in algorithmic skeletons, as well as polyhedral or synchronous dataflow programs, the programmer has no control over communications and the scheduling is entirely *statically computed*. The exact method of communication depends on the approach. This can be slightly relaxed: for instance, explicit communications in Rust can still be safe thanks to the usage of linear types.

In BSP, all the outputs of a sequential quantum are communicated "at once" using a *barrier* when their new values have all been computed. All communications can be seen as deterministically performed at the inter-superstep level.

In DPN and KPN, communications are mixed with computations. This is reinforced by the blocking-read semantics of *channels*: when no data is available on an input port, the actor is blocked, and no other form of communication can happen. Unlike KPN, DPN's communication is not always deterministic.

Actors also feature blocking-reads, but on a mailbox that can be filled by several other actors. This can lead to non-determinism due to races when sending messages to the mailbox. There is no consensus on the use of a partial ordering on communications and some models rely on asynchronous communications while some others enforce a causal ordering of messages.

	Separation of computation and communication Section 5.1.1	Span of sequential computations Section 5.1.2	Means to limit interleaving of communications Section 5.1.3	Potential ways to protect or prevent access to shared data Section 5.1.4
Seq C	NA	Full program	NA	NA
C+POSIX Threads	Manual	Manual	Mutex (Manual)	Atomic (Manual, Semaphore) Future (Manual)
KPN Section 3.1	Explicit and fine-grained	"Process"	Dynamic channels	FIFO channels
StaticDF Section 3.2	Explicit and fine-grained	"Process"	Compiled channels	FIFO channels
SyncDF Section 3.3	Automatic	Block	No control, Compiled communications	FIFO channels Future ([CGP12])
Polyhedral model Section 3.4	Automatic	Full program	No control, Compiled communications	Single assignment with blocking reads Linear types ([HSE+17])
DPN Section 4.1	Fine-grained	Actor	Dynamic channels	FIFO channels
Actors Section 4.2	Explicit and fine-grained	Actor	Mailboxes	Future Linear types ([BCC+15b])
BSP Section 4.3	Explicit and coarse-grained	Superstep	Barriers	Atomic (Barriers)
Algorithmic skeletons Section 4.4	Automatic	Computation function	No control, Compiled communications	Many approaches
Linear types Section 4.5	Fine-grained	Region	Control by typing	Single assignment (Types) Futures ([Teac])

Table 1. Summary of our classification of parallel programming models. The four columns describe the various techniques used to enable determinism, described in Section 5.1.

5.1.4 Protect or Prevent Access to Shared Data. Sharing data is an efficient way to exchange information between processes but it is a frequent source of non-determinism and bugs. One solution to prevent races is the control of the access to this shared data.

The first solution is to only do atomic operations on the shared data. For instance, Software Transactional Memory-based approaches limit the non-determinism introduced by data races by ensuring that transactions are executed atomically, without conflicting access to data. This is very efficient thanks to the lock-free approach but the order of the transactions is still in general non-deterministic. By extension, we can consider BSP as atomic, as communications are done in an atomic step with respect to computations thanks to barriers.

The second solution is to rely on single assignment with blocking reads (Definition 3), notably Futures (Definition 8) and FIFO channels (Definition 5). Indeed the cells of the FIFO channels used in synchronous approaches, as well as DPN and KPN, can be seen as memory cells accessed through single assignment with blocking reads. More generally, linear types also ensure determinism and safety for the access to shared data. Futures and linear types do not prevent race conditions between threads but they prevent all forms of data races.

All approaches described in the survey propose in essence a trade-off over these three major properties. In order to enhance their possibilities, future work may combine the mechanisms described in Table 1. Programming models that are less constrained rely on a clever combination of the four mechanisms identified above: combining two mechanisms provides a more relaxed programming discipline than the strict application of one mechanism. For instance, actor languages

use futures to compensate for the strict separation of data. Each actor is sequential but global execution relies on asynchronous messages that provide task parallelism, and causal ordering can be used for message communications to make the model more deterministic.

Similarly, in the case of synchronous languages, several properties are applied with the additional constraint brought by the synchronous hypothesis that requires every computation step to fit inside a given timeslot. From this point, full determinism is guaranteed because data do not need to be protected as shared data is automatically copied. No synchronisation is necessary because the synchronous hypothesis ensures the availability of data. Finally, communications are implicit and occur between steps and are thus separated from computations.

While we attempted to map out the various mechanisms for determinism in this section, most of the possible combination remain unexplored.

5.2 Strengths of the different Approaches presented in this Survey

In addition to Table 1, we now identify characteristics that we deem fundamental when coming to choosing a language or modeling approach. These are based on our experience as programmers and language designers. For each characteristic, we give some approach examples that particularly seem to address them:

- **Efficiency** is the ability to execute programs with the best performances, according to a set of benchmarks. We believe that efficiency is only meaningful if crucial properties, like determinism or absence of data races, are still guaranteed. To achieve efficiency, different approaches are appropriate for different applications. **The polyhedral model** (Section 3.4) achieves great efficiency for fine grain parallelism with regular dependencies though automatic parallelisation. **Linear types** (Section 4.5) provide "zero-cost" abstraction, which allows programmers to write the most efficient code while ensuring determinism without additional runtime check. Finally **Bulk Synchronous Parallel** (Section 4.3) come with a cost model allowing programmers to evaluate the efficiency of the parallelisation pattern used.
- **Safety** is the ability of the programming model to enable additional safety properties (worst case execution time, non aliasing, absence of use after free, ...). **Synchronous dataflow** (Section 3.3) enables precise dependence computation, which provides worst-case estimations and static sizes for buffers. **Linear types** (Section 4.5) guarantee determinism as a side-effect of the strict typing discipline which also guarantees non-aliasing and absence of use after-free, among others safety properties.
- **Ease of use** is the "entry cost" for programmers used to mainstream languages. In this category, **Actors** (Section 4.2) are of notable interest as they provide concurrent semantics over a simple extension of Object-Oriented programming, making it familiar to all programmers used to this paradigm. **Algorithmic skeletons** (Section 4.4) propose high-level operators which are easy to compose and usable within any language.

6 CONCLUSION

In this survey we proposed a comprehensive review of data race free and deterministic existing parallel programming models. We proposed a classification based on different choices used by each programming model to guarantee determinism. All these criteria are summarised in Table 1. We also discuss additional efficiency, safety and usability characteristics of these approaches, based on our own experience. More important than the classifying table, Section 5.1 analyses and classifies the approaches used by the different languages to limit non-determinism, providing insight on the strength of each language and the way the existing approaches can be combined. We believe our classification can be used to chose among programming models, both as a programmer choosing

an existing programming language, or as a language designer creating a safe and efficient parallel programming language. The bird-eye view provided by the summarising table can also be used to provide ideas for new combinations of models and/or approaches.

This survey also allows us to study precisely different programming methodologies that are often considered by different communities and thus not directly compared in details. Indeed the approaches presented here belong to object-oriented programming, synchronous languages, compilation techniques, functional programming, or high-performance computing, which are rarely if ever discussed together.

Finally, we would like to highlight a possible research direction that can be identified as a missing combination according to our review: the combination of futures or linear types with techniques providing structural constraints on programs, such as dataflow approaches. Indeed, futures and linear types would provide flexibility in these strongly structured languages, without threatening the powerful safety properties provided by the underlying dataflow structure.

REFERENCES

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[ABSS07] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 183–193, New York, NY, USA, 2007. Association for Computing Machinery.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[AIM17] Martín Abadi, Michael Isard, and Derek G. Murray. A Computational Model for TensorFlow: An Introduction. In *International Workshop on Machine Learning and Programming Languages (MAPL)*, MAPL 2017, page 1–7, New York, NY, USA, 2017. Association for Computing Machinery.

[AP21] Christophe Alias and Alexandru Plesco. Data-Aware Process Networks. In Aaron Smith, Delphine Demange, and Rajiv Gupta, editors, *ACM SIGPLAN International Conference on Compiler Construction (CC)*, pages 1–11. ACM, 2021.

[ASS+21] Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, and Guillaume Papauré. Dynamic data race detection for mpi-rma programs. In *EuroMPI 2021 - European MPI Users's Group Meeting*, Munich, Germany, September 2021.

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Shail Arora and Gary T. Leavens, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 1015–1022. ACM, 2009.

[BBN+18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):5:1–5:29, 2018.

[BCC+15a] 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. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. 2015.

[BCC+15b] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Kiko Fernandez-Reyes, Einar Broch Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for the Design of Computer, Communication, and Software Systems, (SFM)*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015.

[BCDH07a] F. Baude, D. Caromel, C. Delbé, and L. Henrio. Promised messages: Recovering from inconsistent global states. In *ACM SIGOPS Conference Principles and Practice of Parallel Programming (PPoPP). Poster*, 2007.

[BCDH07b] Françoise Baudé, Denis Caromel, Christian Delbé, and Ludovic Henrio. Promised messages: Recovering from inconsistent global states. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 154–155, New York, NY, USA, 2007. Association for Computing

Machinery.

[BDL10] D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010. ICCS 2010.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996.

[Ber00] Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[BHRS08] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI '08, pages 101–113, 2008.

[BJH77] Henry. G. Baker Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial Intelligence and Programming Languages*, pages 55–59. New York, NY, USA, 1977.

[BK84] Richard Brent and Hsiang-Tsung Kung. Systolic VLSI Arrays for Polynomial GCD Computation. *IEEE Transactions on Computers*, pages 731–736, 1984.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In Jørgen Lindskov Knudsen, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2001.

[BR05] John Tang Boyland and William Retert. Connecting Effects and Uniqueness with Adoption. In Jens Palsberg and Martín Abadi, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, pages 283–295. ACM, 2005.

[BRR+19] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.

[BS95] Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP)*, volume 982 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.

[BS00] Gérard Berry and Ellen M. Sentovich. An Implementation of Constructive Synchronous Programs in POLIS. *Formal Methods of System Design (FMSD)*, 17(2):135–161, October 2000.

[BSH+17] 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.

[BYR+11] Vamshi Basupalli, Tomofumi Yuki, Sanjay Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and Dave Wonnacott. ompVerify: Polyhedral analysis for the OpenMP programmer. In *International Workshop on OpenMP (IWOMP)*, IWOMP '11, pages 37–53, June 2011.

[CBF+11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In *International Conference on Real-Time and Network Systems (RTNS'2011)*, Nantes, France, September 2011. Irccyn.

[CDdCL06] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, oct 1991.

[CG95] Paul Caspi and Alain Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *1st International Conference on Parallel Processing, EURO-PAR'95*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, August 1995. Springer-Verlag.

[CGH94] 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.

[CGP12] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in Lustre. In *ACM International Conference on Embedded Software (EMSOFT'12)*, Tampere, Finland, October 7-12 2012. ACM. Best paper award.

[CGS+05] 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. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 519–538, New York, NY, USA, 2005. Association for Computing

Machinery.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2005.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[CL07] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, pages 72–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Col91] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[CSST08] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computer Systems*, 7(2):15:1–15:40, 2008.

[CW17] Elias Castegren and Tobias Wrigstad. Relaxed Linear References for Lock-free Data Structures. In Peter Müller, editor, *European Conference on Object-Oriented Programming, (ECOOP)*, volume 74 of *LIPIcs*, pages 6:1–6:32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[CZG+15] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Surveys*, 48(2), sep 2015.

[Dab18] Frédéric Dabrowski. Textual alignment in SPMD programs. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Annual ACM Symposium on Applied Computing, (SAC)*, pages 1046–1053. ACM, 2018.

[DCMM06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of 20th European Conference on Object-oriented Programming (ECOOP)*. Springer, 2006.

[Den74] Jack B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 362–376, 1974.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69. ACM, 2001.

[DLC14] Xuan Khanh Do, Stéphane Louise, and Albert Cohen. Comparing the StreamIt and ΣC Languages for Many-core Processors. In *Fourth International workshop on Data-Flow Models for extreme scale computing (DFM 2014, associated with PACT)*, Edmonton, Canada, 2014.

[DLSD12] Paul Dubrulle, Stéphane Louise, Renaud Sirdey, and Vincent David. A low-overhead dedicated execution support for stream applications on shared-memory cmp. In *ACM International Conference on Embedded Software (EMSOFT*, EMSOFT '12, pages 143–152, New York, NY, USA, 2012. ACM.

[DM98a] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[DM98b] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[DMCN12] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[DRM13] Gwenaël Delaval, Eric Rutten, and Hervé Marchand. Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.

[DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[DTZD08] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, and Ed F. Deprettere. Deriving efficient control in Process Networks with Compaan/Laura. *International Journal of Embedded Systems*, 2008.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. volume 37, pages 237–252, 01 2003.

[EJ01] Johan Eker and Jörn W. Janneck. *An introduction to the Caltrop actor language*. University of California at Berkeley, September 2001.

[EJ03] Johan Eker and Jörn W. Janneck. *CAL Language Report: Specification of the CAL Actor Language*. ERL Technical Memo UCB/ERL M03/48 University of California at Berkeley, December 2003.

[ELGE16] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Surveys*, 49(2), jun 2016.

[ERJ+11] Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow programs with model checking. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 37–42. IEEE, 2011.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[FHLLB09] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 79–90, New York, NY, USA, 2009. Association for Computing Machinery.

[GGPP12] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 51–60, Beijing, China, June 2012. ACM.

[GHLM16] Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *PPDP 2016 18th International Symposium on Principles and Practice of Declarative Programming* , Edinburgh, United Kingdom, September 2016.

[Gir05] Alain Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science, New-York.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. Association for Computing Machinery.

[GM02] Alain Girault and Clément Ménier. Automatic production of globally asynchronous locally synchronous systems. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, pages 266–281, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[GMJ+02] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In Jens Knoop and Laurie J. Hendren, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM, 2002.

[Gor10] Michael I Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2010.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[Hal05] Nicolas Halbwachs. A synchronous language at work: the story of Lustre. In *3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'05)*, Verona, Italy, july 2005.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[HHLS18] Gaétan Hains, Ludovic Henrio, Pierre Leca, and Wijnand Suijlen. Active objects for coordinating BSP computations (short paper). In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018*, LNCS. IFIP International Federation for Information Processing, Springer, June 2018.

[HJP20] Ludovic Henrio, Einar Broch Johnsen, and Violet Ka I Pun. Active objects with deterministic behaviour. In Brijesh Dongol and Elena Troubitsyna, editors, *International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2020.

[HKTJ12] Ok-Kyoon Ha, In-Bon Kuh, Guy Martin Tchamgoue, and Yong-Kee Jun. On-the-fly detection of data races in openmp programs. PADTAD 2012, New York, NY, USA, 2012. Association for Computing Machinery.

[HL06] M. Haustein and K.P. Lohr. Jac: declarative java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[HO09] Phillip Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[HSE+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In Albert Cohen and Martin T. Vechev, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, pages 556–571. ACM, 2017.

[HSH+13] Yaroslav Hayduk, Anita Sobe, Derin Harmanci, Patrick Marlier, and Pascal Felber. Speculative concurrent processing with transactional memory in the actor model. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems*, pages 160–175, Cham, 2013. Springer International Publishing.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[JAD+09] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In Shail Arora and Gary T. Leavens, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 97–116. ACM, 2009.

[JDB+17] Arvid Jakobsson, Frédéric Dabrowski, Wadoud Bousdira, Frédéric Loulergue, and Gaetan Hains. Replicated synchronization for imperative bsp programs. *Procedia Computer Science*, 108:535 – 544, 2017. International Conference on Computational Science, (ICCS).

[JHS+11] 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 *Lecture Notes in Computer Science*, pages 142–164. 2011.

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *ACM Principles of Programming Languages*, 2(POPL):66:1–66:34, 2018.

[JO07] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Softw. Syst. Model.*, 6(1):39–58, 2007.

[JOA03] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In Dag Langmyhr, editor, *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, November 2003.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information processing*. North-Holland, 1974.

[Kes07] Kessler, Christoph and Keller, Jörg. Models for parallel computing: Review and perspectives. 24:13–29, 12 2007.

[KM66] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[KMW67] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 47–57. ACM Press, 1988.

[LGCP13] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and Efficient Bounded FIFO Queues. Research Report RR-8365, INRIA, September 2013.

[LL19] Marten Lohstroh and Edward A. Lee. Deterministic actors. In Tom J. Kazmierski, Reinhard von Hanxleden, and Terrence S. T. Mak, editors, *2019 Forum for Specification and Design Languages, FDL 2019, Southampton, United Kingdom, September 2-4, 2019*, pages 1–8. IEEE, 2019.

[LM87] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

[LMJC20] Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, and Philippe Clauss. Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *29th International Conference on Compiler Construction (CC'2020)*, San Diego, United States, February 2020.

[LTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *International conference on Management of data*, pages 135–146, New York, NY, USA, 2010.

[MHM93] Maurice Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In Alan Jay Smith, editor, *International Symposium on Computer Architecture*, pages 289–300. ACM, 1993.

[Min15] Antoine Miné. AstréeA: A Static Analyzer for Large Embedded Multi-Task Software. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931 of *Lecture Notes in Computer Science*, page 3, Mumbai, India, January 2015. Springer.

[MKI14] Nicholas D. Matsakis and Felix S. Klock II. The Rust language. In Michael Feldman and S. Tucker Taft, editors, *ACM SIGAda annual conference on High integrity language technology (HILT)*, pages 103–104. ACM, 2014.

[Mor16] J. Garrett Morris. The best of both worlds: Linear functional programming without compromise. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *ACM SIGPLAN International Conference on Functional Programming, (ICFP'2016)*, pages 448–461. ACM, 2016.

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. 2005.

[MYC+19] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. Sparse computation data dependences simplification for efficient compiler-generated inspectors. In *Programming Languages Design and Implementation (PLDI)*, 2019.

[MZZ10] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In Andrew Kennedy and Nick Benton, editors, *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 77–88. ACM, 2010.

[NSS06] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[RST20] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. Kindly bent to free us. *Proc. ACM Program. Lang.*, 4(ICFP):103:1–103:29, 2020.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[SHGW12] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *International conference on Parallel architectures and compilation techniques (PACT)*, pages 53–64. ACM, 2012.

[SPH10] Jan Schafer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

[SR15] Aravind Sukumaran-Rajam. *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation*. Theses, Université de Strasbourg, November 2015.

[ŠRU99] Jurij Šilc, Borut Robič, and Theo Ungerer. *Dataflow Processors*, pages 55–97. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. In James H. Anderson, editor, *ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[ST98] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Surveys*, 30(2):123–169, June 1998.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *European Conference on Object-Oriented Programming (ECOOP)*, ECOOP'13, pages 302–326, 2013.

[Teaa] Actix Team. Actix: Actor framework for Rust.

[Teab] Rayon Team. Rayon: Data parallelism for Rust.

[Teac] Tokio Team. Tokio: Asynchronous Programming for Rust.

[Thi09] William Thies. *Language and compiler support for stream programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[TJS20a] Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. Global reproducibility through local control for distributed active objects. In Heike Wehrheim and Jordi Cabot, editors, *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2020.

[TJS20b] Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. Global reproducibility through local control for distributed active objects. In Heike Wehrheim and Jordi Cabot, editors, *Fundamental Approaches to Software Engineering*, pages 140–160, Cham, 2020. Springer International Publishing.

[TMY94] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.

[TP11] Jesse A. Tov and Riccardo Pucella. Practical affine types. In Thomas Ball and Mooly Sagiv, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, pages 447–458. ACM, 2011.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *CACM*, 33(8):103, Aug 1990.

[Wal02] David Walker. Substructural type systems. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*. 2002.

[WPMA19] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019.

[Wya13] Derek Wyatt. *Akka Concurrency*. Artima, 2013.

[YFRS13] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral X10 programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, PPoPP '13, pages 23–34, February 2013.

[ZNS11] J. T. Zhai, H. Nikolov, and T. Stefanov. Modeling adaptive streaming applications with parameterized polyhedral process networks. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 116–121, June 2011.