Défi en compilation et langages: du parallélisme oui, mais du parallélisme efficace et sûr!

Caroline Collange[*], Laure Gonnord[†], Ludovic Henrio[†]

Signataires Gabriel Radanne (Inria Paris), Emmanuel Chailloux (LIP6, Sorbonne Université), Julien Tesson and Mathias Bourgoin (Nomadic Labs), Christophe Alias and Matthieu Moy (Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP), Sid Touati (Université Côte d'Azur), Erven Rohou (Inria Rennes), Emmanuelle Saillard (Inria Bordeaux), Kevin Martin (Université Bretagne Sud).

1 Context

The advent of parallelism in supercomputers and in more classical end-user computers increases the need for high-level code optimization, advanced programming languages, and improved compilers. New architectures such as multi-core processors, Graphics Processing Units (GPUs), many-core and FPGA accelerators, and soon, quantum computers, are introduced, resulting into complex heterogeneous platforms. In particular, FPGAs are now a credible solution for energy-efficient HPC. The multiplicity of hardwares and languages for parallelism raises constant challenges for compilers. We review below existing solutions and raise the necessity to gather the results brought by different communities that could contribute to the correct compilation and execution of parallel programs.

Parallelism and concurrency There might be several reasons to introduce parallelism when executing a program. First, the problem to solve can be by nature parallel, for example because it needs to be performed by several distinct computing units. A typical example of such a scenario is the gathering of information originating different geographical locations.

However, in many cases, and especially in high-performance computing, parallelism is not a desired feature but the only way to achieve the desired performance by spreading a computational task over multiple cores/machines/servers …At a smaller scale, to be responsive, a program often needs to implement local parallelism, in order for example to take into account incoming information while performing a long computation.

The tasks that run in parallel might have conflicting effects, this is why some parallel languages are rather called "concurrent languages" : the concurrency between the different tasks running in parallel might have an effect or not.

Research statement We propose to study and enhance the handling of parallelism in programs, from languages to runtimes. We believe that the two following challenges are equally important : 1 - understanding the different forms of parallelism and 2 - benefiting from the different ways to execute programs while ensuring strong properties about them.

Programming without errors is a difficult task because of the complexity of the algorithms, and because of the complexity due to the interaction between program entities; this problem is at the

[*] Inria, Univ Rennes, CNRS, IRISA

[†] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP

center of many research directions inside the GdR GPL group. However in case of parallel programs the task is even more difficult because the interaction between two program entities can occur at any moment. The programmer has to take into account additionally the concurrency between the different tasks of the program which makes the verification of the program correctness much more difficult. *Race condition* is the notion related to such concurrency : it describes a situation where two tasks perform a conflicting action, and depending on the task that acts first the result can be different.

In particular, parallel programs can raise two categories of bugs that are difficult to find and very annoying : deadlocks (several tasks blocked because each of them needs an action of another task to progress) and data-races (two tasks trying to access the same data in a conflicting manner). Data-races are a special case of race condition where the race concerns the access and modification to some data, this is the lower level race condition and the most undesirable. The design of programming languages for parallelism should find ways to reduce the risk of having such bugs while allowing the programmer to write complex and efficient programs.

When a program is deterministic, it has no race condition at all, and it either systematically deadlocks or never deadlocks but limiting parallel programming to deterministic languages is a bit too restrictive in terms of expressiveness in general.

Finally, the latest kind of hardware accelerators, quantum computers, offers a completely different and much less mature programming model. Designing compilers for these targets demands a radical departure from the traditional ways of classical computer architecture and compilers.

2 Research agenda : parallelism everywhere

In this section we propose to make a tour of different approaches to achieve the goal of safe yet efficient parallel programs. In particular we identify challenging questions for the GDR GPL groups.

2.1 Language-based approaches for parallelism

The goal of language-based approach is two-fold : enable the use of modern software development techniques such as modular programming, and improve the safety of programs by preventing some errors by providing programming abstractions that forbid undesired behaviours. As stated above, in many cases non-determinism is an undesired behaviour, and consequently several languages provide abstractions restricting or forbidding non-deterministic behaviours. We distinguish three approaches :

— Introduce new programming constructs which, by construction, will never exhibit bad properties regardless how they are used. Such constructs must be supported by appropriate runtime ensuring that the semantics never exhibit any non-determinism, regardless of how programs are composed. This is the case, for instance, of Kahn process networks [Kah74], or of Actors [Agh86] that by nature ensure determinism and consequently the absence of any kind of race-condition. But Kahn process networks are a bit restrictive in terms of allowed parallelism; many programming models allow more parallelism while allowing the programmer to easily control race-conditions, and often preventing data-races [BSH$^+$17].

— Augment an existing language which rich type systems (and type check or type inference) to statically prevent bad behavior such as data races. This is the case of Rust, which does not require any runtime support and support very low level access to the system, but uses a rich type system to rule out several forms of data races. More generally linear and ownership types is one solution frequently used to prevent data-races from the type system point of view [BCC$^+$15], while keeping a rich, efficient and expressive programming language. More generally, it is crucial to design static analysis and type systems that complement adequately the languages designed above, guaranteeing the correct behaviour of programs that adopt the programming model specified in the previous approach.

- Design analyses on top of existing languages in order to a posteriori prove the absence of deadlock or race conditions. These analyses can be performed on any type of language, from general purpose languages with explicit threads and mutexes to languages that already enable to express coarse grain construction like premises. It is thus necessary to design scalable analysers that can analyse parallel code, in the spirit of AstreeA [Min15].

Naturally, in practice these approaches form a continuum, however, in the current structuration of the GDR they are addressed by different subgroups : Compilation, LAMHA, and LTP.

In the "language approach" we also classify all the approaches that are more attached to the study or a programming model, i.e. a way to program parallelism, without being tied to a programming language. Here are the challenges we identify in this approach :

1. Design programming models that are convenient to program, can lead to efficient execution of programs, and help the programmer to write programs without bugs, deadlocks, or data-races.
2. Design static abstractions, e.g. type systems, that increase the reliability of the programs while keeping the programming language expressive enough.
3. Design proper abstractions to get precise analysis; and analyses that benefit from high level constructions (or guarantee by construction) of the language.

2.2 Compilation approaches

The goal of compilation-based approach is two-fold : let the user declare the *optimisation potentiality* for her program while still remaining readable and independant from the final usage (architecture, level of parallelism, ...). We can distinguish two approaches :

- Many variants of C have been proposed, the most famous is OpenMP (`https://www.openmp.org/` that enables the programmer to declare independant tasks through pragmas. It is up to the compiler to parallelise these tasks effectively (and generate communications).The expected behavior parallelisation pragmas, such as the ones of OpenMP, is generally unformally described in the norm, and some of them are only user declarations.
- Express the computation itself and let the compiler generate and optimise code. For instance in the polyhedral model framework [FL11], DSLs such as Alpha [RGK11] are used to express and then are aggressively compiled into equivalent tiled/pipelined sequential code or parallel code (with explicit communications).

These compilation approaches are currently studied in the Compilation subgroup of the GDR. Scientific research questions :

1. How to formally specify the behaviour of parallelisation pragmas ? How to ensure that the compiler does not introduce bugs ? How to ensure statically or at runtime that the asumptions made by the developer are made explicit ?
2. Polyhedral aggressive compilation and loop rescheduling often produce complex code with a non trivial structure : how to be sure that these transformations are safe ? This is a non trivial application for *certified compilation*.

2.3 Runtime approaches

As for runtime the objective is to guarantee that the execution of the parallel programs will follow the properties proven. This is generally ensured by checking that the runtime allows exactly the executions specified by the language semantics. This part can be ensured more or less formally :

- Ideally, there would be a complete formal proof of the runtime platforms according to the parallel language semantics. However the the complexity of the runtime platform and of the parallel languages make the complete formal proof of correctness often not realisable. We believe that static properties could still be verified by allowing coarse grain abstractions.
- By runtime verification [BFFR18, AHO19, EHF18] : instrumenting the generated code to at least warn during executions if one of the semantics assumptions or properties are not required or statically proven.

It is necessary that the developers of the runtime environment, the person that specifies the behaviour of the language, and the developers of the static analyses for the parallel languages agree on the semantics of the program, at least in an informal way, and make sure that all the developments respect this semantics. Scientific questions [1] :

1. How to design runtimes and prove that they execute the semantics of the code they are given ? Even in the simpliest case where runtimes make no decision choices for scheduling it is not trivial to guarantee a correct instruction scheduling at runtime (and thus propagate the properties proven on the program, like e.g. absence of race conditions).
2. Correct proofs of complex runtimes, such as for example StarPU [ATNW11], seems to be untractable for formal proofs with provers, however some of its parts such as the task sheduler is by itself interesting in order to understand relationships between operational semantics and its execution support. This emphasizes the need to collaborate with other groups that currently do not belong to the GDR GPL.

2.4 Complementary approaches

Of course, parallel programs have also other characteristics that deserve to be studied :

- These programs need to be designed in productive ecosystems : from specification to runtime, verification and experimental evaluation, all software engineering techniques in general have to be rethought, notably in terms of user-friendliness, scale, and heterogeneity of code. These activities are also part of the GDR topics (MFDL, RIMEL, GLACE).
- The intrinsic sequential part of parallel programs should also be studied. Code experts are able to invoke clever sequence of compiler optimisations, none of them being used by default. We also advocate in favor of designing expert benchmarks that demonstrate state-of-the art optimisation potential.
- Parallel programs execute themselves on physical machines, and runtimes are usually designed to fit these particular machines, that also contain increasing hardware-based solutions : branch prediction, vectorisation. A proper study of safe concurrency should also not forget to take these architecture into account, the main challenge being to properly describe their (complex) behavior. This is part of the activity of the Compilation Group in the SOC^2 GDR.

[1] In the GDR GPL, some of these questions are studied by the LAMHA group

3 Compilers for quantum computing

While multi-core CPUs, GPUs and even FPGAs fall under the same category of parallel processors and may benefit from the aforementioned approaches, quantum computers are currently a category on their own that comes with its own new set of challenges.

- As the first quantum computers will remain too limited to accommodate error correction for a while, quantum program compilers will have to deal with noise. Effective compiler optimization are thus critical, not just for the execution time, but more importantly for noise sensitivity. Noise and its characteristics should be modeled and taken into account at compile-time.
- Quantum compilers need the equivalent to instruction scheduling and register allocation, that is quantum gate scheduling and qubit allocation. The quantum context gives new constraints : the no-cloning theorem forbids the simple duplication of information, and the connectivity across qubits is typically limited. The proper level(s) of abstraction for quantum gate selection and scheduling in particular is still an open research question.
- Quantum compilation currently lacks an equivalent of the successful SSA form of classical compilers. An intermediate representation that enables both powerful analyses and efficient code generation is still to be found.
- Compilers and runtimes also need to manage interactions between the quantum and classical world. Integration issues for a quantum co-processor within a classical system and interplay between the quantum and classical parts will need taking into account. Compilers will have to to support dynamic quantum-driven classical control-flow.

4 Concluding remarks

Different possible classifications

The challenges raised in this document can be additionally classified according to two dimensions : the dimension of the target running hardware and the dimension of the expressed parallelism.

On one side side platforms for running programs range from sequential processors to distributed systems made of independent computers spread over the intermediate. Between these two extremes, we find multi-cores, many-cores, GPUs, and FPGAs. Depending on the hardware, two problems arise : memory consistency and scheduling of different computing entities. Without aiming at a full review here, it is obvious that the most efficient way to coordinate two cores with cache consistency issues cannot be the same as the coordination of two computers not sharing their memory.

On the other side, the same kind of classification could be made from programming models, ranging from simple threads that can be efficient but difficult to coordinate safely, to actors considering each computing unit as independent entity communicating by message passing. Intermediate solutions include bulk-synchronous parallel model adapted to data-parallelism ; algorithmic skeleton library (like MapReduce) providing more abstraction and more automation, but less expressive ; standard lower-level parallel libraries like MPI and OpenMP ; etc.

One could think that one programming model is better adapted to one kind of hardware or running platform, but, experience has shown that there is no obvious correspondence in practice. For example the actor model enforces a strong separation of data where each memory location can be manipulated by a single thread. This seems better adapted to distributed systems and actors indeed shine naturally in such setting. However in the last 20 years huge efforts have been made to make actors efficient on single machines, perhaps because language designers were convinced that they provide a good programming abstraction. First efforts have been made to implement actor runtimes that can instantiate thousands of logical threads on the same machine, then different approaches have been proposed to avoid systematic copy of data upon actor communication, or to make actors multi-threaded.

The different efforts to make a programming paradigm work in different settings rely on a combination of language, compilation and runtime solutions, according to the organisation proposed above.

A working group on a language and compilation approach for safe parallelism

As mentioned in the presentation of the challenge, several combination of approaches already exist and several distinct communities address the challenge of safely and efficiently compiling parallelism. Our challenging objective is to gather the ideas and results originating from these different communities into a single working group, providing coherent solutions that combine modern state of the art results in expressive programming languages, powerful analysis and compilation techniques, and efficient runtime environments. We additionally want to stress that we are aiming at an approach that is well specified and provides guarantees of correct behaviour and efficiency to the programmer.

We believe that the complexity of the task highlights the need for a research group that would gather researchers that are experts in parallel architectures, programming languages, high-performance computing, typing and static analysis, compilation, and runtimes.

This workgroup could take the following forms :

1. Slightly extend the scientific outlines of the Compilation group toward parallel languages, quantum computing and semantics, and organise regular joint meetings with other groups like LAHMA and LTP.
2. A more ambitious solution is to merge the Compilation group and the LAMHA group, including also quantum computing in the research agenda. As the group would get bigger, subgroups meetings would be organised in addition to the meetings of the research group.

Références

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

[AHO19] Wolfgang Ahrendt, Ludovic Henrio, and Wytse Oortwijn. Who is to blame ? runtime verification of distributed objects with active monitors. *Electronic Proceedings in Theoretical Computer Science*, 302 :32–46, Aug 2019. Post-proceedings of VORTEX 2018.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[BCC$^+$15] 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.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification. Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, February 2018.

[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.

[EHF18] Antoine El-Hokayem and Yliès Falcone. Can We Monitor All Multithreaded Programs ? In *RV 2018 - 18th International Conference on Runtime Verification*, pages 1–24, Limassol, Cyprus, November 2018.

[FL11] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

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

[Min15] Antoine Miné. AstréeA : A Static Analyzer for Large Embedded Multi-Task Software. In *16th 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.

[RGK11] Sanjay Rajopadhye, Samik Gupta, and Dae-Gon Kim. Alphabets : An extended polyhedral equational language. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 0 :656–664, 2011.