Synchronization Property Checking and Inference in a Lock-Step Synchronous Parallel Replica Language

Jari-Matti Mäkelä[1], Ville Leppänen[1] and Martti Forsell[2]

[1] University of Turku
Dept. of Information Technology
jmjmak@utu.fi, ville.leppanen@utu.fi
[2] VTT
Oulu, Finland
Martti.Forsell@VTT.Fi

As more and more computer platforms adopt the model of multiple processing cores to get further speedup from parallelism, programming languages also need to adapt. Contemporary hardware solutions include heterogeneous systems with units optimized for different tasks, distributed memory clusters, and shared memory systems with different consistency models (e.g. on one end constrained platforms such as the GPGPU and on the other end more general purpose systems such as NUMA x86). As chip multi-processors provide new potential for organizing data sharing, we only consider languages targeting these systems.

We argue that in the case of general purpose algorithms, both the hardware and software solutions approach the issue with suboptimal abstractions, effectively preventing maximal utilization of the computational power. Parallelism is typically orchestrated with explicit locks, which leads to difficult problems with performance scaling and correctness [2]. Some platforms provide sequential consistency for safer programming with locks, but even more relaxed models [1] are in use in performance oriented languages such as C++.

We assume a totally different model for computation with strong synchronization guarantees. The SB-PRAM [6], TOTAL ECLIPSE [4], and REPLICA [5, 7] platforms conceptually follow the PRAM (*parallel random access memory*) model of computing. That is, each thread's computation proceeds in a globally synchronous lock-step at instruction level. Previously the model was considered too inefficient to be practical compared to more relaxed execution models. However, modern techniques of parallel *multi-operations, latency hiding* and *wave synchronization* alleviate the issues with the approach.

In this model, instructions have a unit time amortized cost in terms of cycles in relation to other threads, which makes it is possible to reason about the time cost of a sequence of instructions assuming no branching happens. This opens up a possibility for a different programming style where threads can be grouped so that the group property holds through a sequence of code and abstractions can make assumptions of the synchronicity of a group of threads. However, the hardware provided synchronicity on instruction level does not implicitly extend to higher level abstractions because the control flow structures may diverge the flow until explicitly synchronized with a barrier.

The *Fork* [6] language adopts a similar approach for maintaining synchronicity on block and function level, but does it explicitly with user annotated regions (*async, sync, straight*) that are statically checked. The major disadvantages of Fork's approach are 1) the need for tedious bookkeeping when switching between asynchronous and synchronous modes both at call site and in function signatures and 2) the limitation of reusing code in different synchronicity context. However, Fork does prevent erroneous use of synchronicity from compiling and provides a way to structure code into parallel and sequential sections.

Our approach originally presented in [8] attempts to automate the chore of tagging code with synchronization metadata like in Fork. In the previous work, each syntactic language entity (expression, statement, function, higher order abstraction) is associated with *synchronicity pre- and post-conditions* and intertwining constraints. The mechanism not only covers the case of *async/sync/straight* blocks of code like in Fork, but supports arbitrary conditions assuming the checking algorithm is provided with constraints involved additional conditions. As an example, we consider the synchronization token attribute that can be used to accelerate multi-operations on architectures such as REPLICA. New constraints could be provided as part of the compiled program, but in the case of Replica language, for simplicity we only use a static constraint set defined in the compiler.

A short summary of the checking algorithms is given next. For each syntactic language entity, the properties are represented by a pair of in- and outgoing sets of attributes ($\langle F_1, F_2 \rangle$) and the related constraint is a logical boolean predicate C representing a set invariant that covers both F_1 and F_2. On the implementation side, the compiler has an additional control flow analysis pass that carries the synchronicity condition state throughout the program in one pass and for each entity checks that the conditions are satisfied. A constraint rule is required for each language entity. The previous work also discussed so called "implicit conversion" rules between the states when transitioning between different sections with respect to synchronicity.

The problem of generating compatible conditions that satisfies the constraints for all parts of the program – that is, an inference algorithm – was not fully addressed in the previous work although it was mentioned that the constraint rules give rise to a similar inference technique as with e.g. standard typed lambda calculus. In the case of rules with no state dependencies, its state can be easily inferred, but there is no single solution in the case when states form complex dependencies between several syntactic nodes. While the condition checking against the constraints can be performed without backtracking or lookup in a linear fashion with respect to the syntax tree traversal, the inference algorithm requires arbitrary lookup for determining the need and position for code generation.

The main contribution of this work is to provide a way to automatically tag the test the correctness of the tagging is given for a minimal core of the Replica language. We propose a solution that handles basic functions and blocks with multiple exit points. We also revisit the idea of duality between rules for constraint checking and inference as discussed in the previous preliminary paper and elaborate how the mechanism extends to user defined abstractions.

In addition to the treatise of the inference and checking algorithm, we show the performance implications of the machine checked and generated code versus explicitly stated synchronization directives in computational kernels. The performance effect of the synchronization token is also measured with a task parallel runtime stub library that spawns a single task for evaluation purposes. Further applications for the synchronicity conditions in faster branching control flow constructs (so called "fast operations" in the E language [3]) are also considered. The goal of the benchmarks is to show the performance advantage of the approach with inferred synchronization properties versus a simpler, more generic algorithm that does not assume a strictly synchronous mode as a default nor does have hardware acceleration support for fast operations or special task tokens.

References

[1] Sarita V Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *computer*, 29(12):66–76, 1996.

[2] M. Duranton, D. Black-Schaffer, S. Yehia, and K. De Bosschere. Computing systems: Reseach challenges ahead – the hipeac vision 2011/2012, 2012.

[3] M. Forsell. E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. *WSEAS Trans. on Computers*, 3(3):807–812, jul 2004.

[4] M. Forsell. TOTAL ECLIPSE – An Efficient Architectural Realization of The Parallel Random Access Machine. *Parallel and Distributed Comput., Ed. A. Ros, IN-TECH, Wien*, pages 39–64, 2010.

[5] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. *Int. Journal of Networking and Computing*, 1(1):21–35, 2011.

[6] C.W. Kessler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of parallel programming*, 1997.

[7] J-M. Mäkelä, E. Hansson, M. Forsell, C. Kessler, and V. Leppänen. Design Principles of the Programming Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In *Proceedings of 4th Swedish Workshop on Multi-Core Computing*, page 136. Linköping University, 2011.

[8] J-M. Mäkelä, V. Leppänen, and M. Forsell. Composable Hierarchical Synchronization Support for REPLICA. In Kiss kos, editor, *13th Symposium on Programming Languages and Software Tools*, pages 230–244. University of Szeged, 2013.