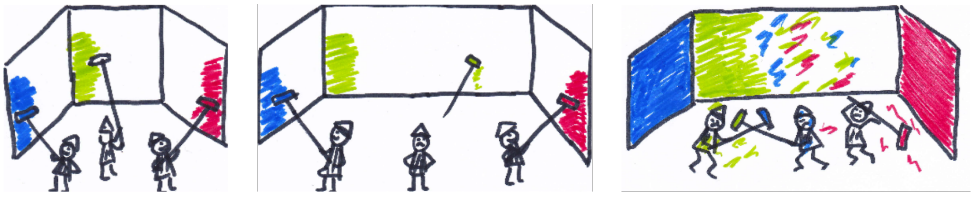
Highly Concurrent Data Structures// Joint project Telecom ParisTech/EPFL

Goals: Design of highly-concurrent data structures.

Tools: Logic, algorithmic reasoning, programming.

Prerequisites: Some maturity in math and algorithms, (optionally) basic concurrent programming skills.

Synchronization involves resolving *data races*, informally, resolving the conflicts about who goes first in accessing a shared resource, which may not scale with the number of processes. As an extreme example, imagine a computational task that has a "parallelizable part" (captured by a fraction p) that can be split at an arbitrary number of parallel sub-tasks and a "sequential part" (fraction $1-p$) that can only be solved by a single process. The resulting *speedup* capturing how much faster n processes will solve the task compared to one process is then $1/(1-p+p/n)$. Thus, no matter how many processes we can use, the speedup will be upper-bounded by $1/(1-p)$.



(a) (b) (c)

Figure 1: Amdahl's law in action. How can three people paint three walls? No problem if the walls are of equal size and the workers are of equal skills (1(a)), but quite problematic when the work is not equally distributed (1(b)) and synchronization among them is required (1(c)).

This observation, known as 'Amdhal's law' (after Gene Amdhal), suggests that to efficiently exploit the parallel-processing abilities of modern computing systems, it is crucial to minimize the synchronization costs, even in presence of failures and asynchrony (unavoidable in real systems).

Designing highly concurrent *data structures*, such as lists, sets, directories, etc., is therefore believed to be a very important challenge that is, however, not easy to meet.

In this work we aim at designing data structures that provide *optimal* concurrency. What does it mean? Informally, a data structure that accepts *every* concurrent schedule, i.e., every possible interleaving of memory accesses of concurrent threads is called concurrency-optimal. Interestingly, the criterion of *concurrency-optimality* [1] is not exactly orthogonal to the choice of synchronization techniques. We observe, for example, that *pessimistic* (conservative) lock-based synchronization is inherently sub-optimal, as it must sometimes conservatively grab locks on the elements of a data

structure in order to avoid inconsistencies in the future. Similarly, strongly consistent transactional synchronization aims at *serializabiliy* of high-level concurrent operations (such as updates or lookups on a dictionary) even when certain operations do not conflict and, thus, may execute correctly in parallel.

As the first step, we consider *linked-list*-based implementations of a `set` object, a convenient abstraction in concurrent programming. We observe that know to be the most efficient implementations to date [2–4] are not concurrency-optimal, as they reject certain potentially correct schedules. Would we be able to devise a more concurrent implementation? And, if yes, would not the intrinsic synchronization overhead be harmful for the performance gains?

The project is maintained in collaboration with the Distributed Programming Lab, EPFL (Prof. R. Guerraoui, `http://lpd.epfl.ch`).

Contact

Prof. Petr Kuznetsov
`http://www.infres.enst.fr/~kuznetso/`
`petr.kuznetsov@telecom-paristech.fr`
INFRES, Télécom ParisTech
Office C213-2, 46 Rue Barrault

References

[1] V. Gramoli, P. Kuznetsov, and S. Ravi. Optimism for boosting concurrency. *CoRR*, abs/1203.4751, 2012. http://arxiv.org/abs/1203.4751.

[2] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[3] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.

[4] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.