High-level lock-less programming for multi-core

Fabio Tordini*, Marco Aldinucci*[1],
Massimo Torquati†[2]

* *University of Torino, corso Svizzera 185, 10124 Torino, Italy*
† *University of Pisa, largo Pontecorvo 3, 56127 Pisa, Italy*

ABSTRACT

Modern computers are built upon multi-core architectures. Achieving peak performance on these architectures is hard and may require a substantial programming effort. The synchronisation of many processes racing to access a common resource (the shared memory) has been a fundamental problem on parallel computing for years, and many solutions have been proposed to address this issue. Non-blocking synchronisation and transactional primitives have been envisioned as a way to reduce memory wall problem. Despite sometimes effective (and exhibiting a great momentum in the research community), they are only one facet of the problem, as their exploitation still requires non-trivial programming skills.

With non-blocking philosophy in mind, we propose high-level programming patterns that will relieve the programmer from worrying about low-level details such as synchronisation of racing processes as well as those fine tunings needed to improve the overall performance, like proper (distributed) dynamic memory allocation and effective exploitation of the memory hierarchy.

KEYWORDS: Concurrency; multiprocessors; non-blocking synchronisation; skeleton programming

1 Introduction

The whole hardware industry has been moving to multi-cores, which nowadays equip the large majority of the computing platforms. The rapid shift toward multi-core technology has many drivers that are likely to sustain this trend for several years now. Software technology is moving under this pressure [ABD+09]. Certainly, in the long term writing parallel programs ought to be as efficient, portable, and correct as it has been to write programs for sequential computers. To date, however, the parallel programming drill does not embrace much more than low-level communication and synchronisation libraries. In the hierarchy of abstractions, it is only slightly above toggling absolute binary into the front panel of the machine. We believe that, among many, one of the reasons of such failure is due to the fact

[1]E-mail: {tordini, aldinuc}@di.unito.it
[2]E-mail: torquati@di.unipi.it

that programming multi-core is still perceived as a branch of high-performance computing and the consequent exclusive focus on absolute performance measures. By definition, the raison d'être for high-performance computing is high performance, but MIPS, FLOPS and speed-up need not be the only measure. Human productivity, total cost and time to solution are equally, if not more important [Ree09]. While a big methodological change will be required to design applications that are likely to be designed as parallel programs, this shift is needed to be graceful in the short term: existing applications should be ported to multi-cores with moderate effort (despite they could be redesigned with a larger effort and larger performance gain).

Current commodity multi-core and many-core platforms exhibit a cache-coherent shared memory, which can effectively reduce the programming complexity of parallel programs (whereas different architectures, such as IBM Cell, have exhibited their major limits in programming complexity). Cache coherency is not for free, however. It largely affects synchronisation costs (e.g. mutual exclusion) and may require expensive performance tuning. The scenario is even more convoluted considering that almost the totality of commodity multi-core architectures exhibits a *cache-coherent non-uniform memory access* (cc-NUMA) and weak memory consistency models (e.g. Total Store Ordering and Weak Ordering) [AG95].

In cc-NUMA systems, both the non-optimized access (e.g. randomized) to memory space and cache re-conciliation protocols can be a real issue, especially in fine-grain parallelism exploitation. The former issue is still partially unaddressed and it is one of the topics this proposal is focusing on. In respect of the latter issue, optimistic concurrency control approaches, inter-alia, non-blocking (lock-free) algorithms equipped with concurrent data structures, transactional primitives (both hardware and software) are promising (yet well-known) technologies to ameliorate cache re-conciliation costs by way of mutual exclusion avoidance.

Unfortunately, non-blocking synchronisation algorithms exhibit a natural complexity and a delicate implementation under the weak memory consistency models adopted by almost all multi-core CPUs. As an example, concurrent containers with lock-free methods can be hardly composed and proved correct, and eventually, their usage requires particularly skilled programmers.

This is both an opportunity and a challenge for parallel programming framework designers, since a properly designed framework should support the application with easy exploitation of parallelism (either design from scratch or porting from sequential code) and high-performance. In this respect, while the *direct usage* of nonblocking algorithms are not likely to be considered a viable for mass software production, they still represent a viable solution to build higher level coding abstraction, e.g. algorithmic skeletons and parallel patterns [Col89, GVL10, EK10, ADKT12, ABD$^+$09].

As an example, starting from lock-free CAS-free FIFO queues, FastFlow implements a set of C++ parallel programming patterns providing the programmer with efficient, high-level and easy to use programming abstractions. Building on this experience, we aim to extend both FastFlow design and implementation.

2 The FastFlow parallel programming framework

FastFlow is a programming environment specifically targeting cache-coherent shared-memory multi-cores. FastFlow is implemented as a C++ template library built on top of a lock-free synchronisation mechanism. Conceptually, it is a stack of layers that progressively abstract

the shared memory parallelism at the level of cores up to the definition of useful programming constructs and patterns [fas09, ADKT12].

FastFlow provides mechanisms to define simple streaming networks whose *run-time support* is implemented through efficient lock-free Single-Producer/Single-Consumer (SPSC) queues. These queues constitute a solid ground for a low-latency synchronisation mechanism for multi-core (e.g. 10 nS latency core-to-core on Intel E7@2Ghz). SPSC queues, whether composed with mediator threads, can be used to build memory fence-free Multiple-Producer/Multiple-Consumer (MPMC) queues. These queues, or alternatively with CAS-based implementations of MPMC queues, make it possible to build arbitrary networks of activities behaving as streaming networks [Kah74].

On the top of streaming networks, FastFlow provides a programming framework based on parallelism exploitation patterns (a.k.a. *skeletons*) [Col89]. They capture common parallel programming paradigms (e.g. MapReduce, ForAll, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics.

Parallel patterns are usually categorized in three main classes: Task, Data, and Stream Parallelism. FastFlow specifically focuses on Stream Parallelism, and in particular provides: *farm*, *farm-with-feedback* (i.e. Divide&Conquer), *pipeline*, and their arbitrary nesting and composition. The set of skeletons provided by FastFlow could be further extended by building new C++ templates or even further abstracted to derive problem specific skeletons.

FastFlow has been used in a number of applications; they typically exhibit a better or similar performance compared with their counterparts developed using state-of-the-art programming frameworks (e.g. Intel TBB, Cilk, OpenMP) [AMT10, fas09].

3 Future Work

Within the framework of the ParaPhrase FP7 EC-STREP project, we aim to explore the usage of parallel patterns as a main vehicle to provide the programmer with portable and efficient programming abstractions. As mentioned, the work will focus mostly on data movement and so-called *memory affinity*, which — we believe — represent two of the cornerstones of efficiency and programmability in the code production for multi-cores. In particular, the work will unroll along two main directions:

Memory affinity. Memory affinity addresses both performance and performance portability of programming frameworks. The idea underpinning memory affinity is similar (in spirit) to parallel patterns but it applies to data structures and their allocation: under cc-NUMA assumption, the data structures allocation in memory (and their mapping onto physical memory) depends on their usage, i.e. on how they are *placed* and *moved* across the system. This task is typically too complex to be studied in the most general case, as it depends on many parameters, such as memory access patterns, memory allocation strategies, thread-to-core pinning, etc. We believe that the problem becomes tractable in a pattern-based system, as the data-paths are really implicit in the patterns themselves. We aim at describing a number of data structures that can be effectively coupled with parallel patterns. In order to accomplish this task, the full control of allocation strategy will be possible by way of the lock-free memory allocator already implemented in the FastFlow framework.

Extended lock-free run-time support. Memory affinity described in a high-level programming model (e.g. concurrent data structures), should be supported by an extend set of synchronisation mechanisms at the run-time level. On the ground of existing work, they will be of two main classes: queues of various kind (single and multiple producer-consumer) supporting producer-consumer data movement, and transactional primitives mainly aiming to support (optimistic) true concurrency on data.

References

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[ADKT12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2012.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[AMT10] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with fastflow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, February 2010. IEEE.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[EK10] Johan Enmyren and Christoph W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proc. of the 4th Intl. workshop on High-level parallel programming and applications (HLPP)*, pages 5–14, New York, NY, USA, 2010. ACM.

[fas09] *FastFlow website*, 2009. `http://mc-fastflow.sourceforge.net/`.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, 1974. North Holland, Amsterdam.

[Ree09] Daniel Reed. *High-Performance Computing: Where'd The Abstractions Go?* BLOG@CACM, May 2009.