How to Achieve Scalable Fork/Join on Many-core Architectures?

Mattias De Wael *
Software Languages Laboratory
Department of Computer Science
Vrije Universiteit Brussel, Belgium
madewael@vub.ac.be

Tom Van Cutsem †
Software Languages Laboratory
Department of Computer Science
Vrije Universiteit Brussel, Belgium
tvcutsem@vub.ac.be

Abstract

Fork/Join is a parallel programming model that implicitly assumes uniform memory access. The transition from multi- to many-core architectures will render this assumption invalid, and consequently it is likely that Fork/Join in its current form will not scale. This research investigates implementations for Fork/Join to allow the transition to many-core.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming, Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features Concurrent programming structures

Keywords Fork/Join, Work Stealing, Many-core

1. Introduction

Contemporary processors become ever more parallel, instead of faster as they used to. This new approach of increasing processor power has put a stop to the virtually free improvement of software performance with every newly released processor. Instead, if we want to continue the production of software that scales in performance with the new multicore processors, then the programs themselves need to become parallel as well. Simultaneous with the rise of parallel processors, the model of uniform memory access starts to break down, amplifying the bottleneck effect of memory latency on performance even more. Therefore, hardware architects do not only increase the number of cores on a chip, but also envision and implement new memory architectures. The latter is what is also referred to as the transition from multicore to many-core architectures. This research wants to prepare the convenient to use parallel programming model Fork/Join for this transition. This text presents the problems we foresee in this transition from multicore to many-core architectures, as well as some preliminary ideas to solve those problems.

* Supported by a doctoral scholarship of IWT-Vlaanderen, Belgium
† Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

2. Fork/Join and Work Stealing

Fork/Join is a convenient to use parallel programming model. Today we see a great variety of Fork/Join implementations. Either as library [9–11] or as language (extension) [3, 5]. The extra cognitive burden of Fork/Join for the programmer is limited to only 2 constructs: *fork* and *join*. Other concerns such as load balancing are taken care of by a scheduler. Fork/Join parallelism forces its users to focus on exposing parallelism by dividing a problem into potentially parallel tasks. These tasks, usually of recursive nature, form at run-time a directed acyclic graph of dependencies. The graph, usually referred to as the computation graph, forms a solid model to reason about the computation. At run-time, the generated tasks are assigned to processing cores without intervention of the programmer. Contemporary Fork/Join implementations all use some variation of work stealing, where idle processors steal work from busy procesors [2, 7].

3. The Shift from Multicore to Many-core

Today, multicore architectures with uniform memory access, up to cache hierarchies, are ubiquitous. However, with the rise of many-core architectures [4, 8, 12], uniform memory access becomes unfeasible. Hardware architects will have to shift to memory architectures with non-uniform access (NUMA). For instance, contemporary hardware exists where a shared memory is supported by an elaborate cache hierarchy, and, furthermore, these caches can be dynamically accessed by neighboring processing cores. Not only caches will play an important role, but also the idea of a network-on-chip that allows fast and direct inter-core communication can already be seen in contemporary hardware [13]. Now, data is not longer close by, or far away, but it can be anywhere in between.

4. Foreseen problems

All the Fork/Join implementations use a (slightly) different scheduling policy. Each flavor of Fork/Join (and its scheduler) behaves differently for different scenarios, and operates with different stack and memory bounds [7].

Today, Fork/Join and work stealing implicitly assume uniform memory access, in the sense that contemporary implementations do not allow access control. Therefore it is likely that in its current form Fork/Join programs will not scale into the many-core era. Since steals are randomized, and the number of potential victims for work stealing increases, the chance of counter productive steals increases due to bad data locality and other NUMA effects.

5. Ideas for possible solutions

To prepare Fork/Join and work stealing for the many-core era, we think changes are needed on three frontiers: at the level of *language* constructs, at the level of *algorithm* design, and at the level of the work stealing *scheduler*. We want to evaluate solutions proposed for multicore work stealing with improved data locality [1], as well as as solutions proposed for distributed work stealing and PGAS-like languages [5], and new solutions that emerge because of the potential of the upcoming hardware.

We will need a richer language to express new phenomena, such a data locality, without adding to much cognitive burden. Currently we think about expressing data locality similar as the efforts done for PGAS languages, such that data and/or tasks can express more elaborate on their data access patterns. On the other hand, we think about type annotations for tasks, added by the programmer or a compiler, to provide more information about the behavior of a task, e.g. tasks that synchronize by their very nature, require less synchronization at the scheduler level. At the algorithmic level, the use of cache oblivious algorithms and data structures [6], can aid programmers in creating scalable Fork/Join programs.

Finally, we want to improve upon the work stealing scheduler itself. We want to evaluate and adapt techniques such as locality guided work stealing [1], and adaptive work stealing [7] on the many-core architectures. A second idea on how to improve Fork/Join's work stealing scheduler is to anticipate on processors becoming idle before they actually do. For instance by sending non-blocking messages to nearby workers to ask for more work when they are about to become idle. Finally, with the extra information about the characteristics of a task (e.g. idempotent, self synchronizing), variants of the local deques can be implemented that require less syntonization and therefore induce less overhead.

6. Conclusion

Since the turn of the century, the need for parallel programs increases. Fork/Join proves to be a convenient to use model to express divide-and-conquer algorithms in a parallel fashion. The model is adopted and implemented by various industry strength vendors. However, with the rise of the many-core architectures this model is unlikely to scale on these new hardware. We propose to adapt Fork/Join on three frontiers to tackle the problems induced by the transition from multicore to many-core architectures.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, 2000.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, September 1999.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[4] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, 2007.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. 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*, 2005.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 1999.

[7] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. *Parallel and Distributed Processing Symposium, International*, January 2010.

[8] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. Technical report, Intel White Paper, 2006.

[9] Intel. Threading Building Blocks. `http://threadingbuildingblocks.org/`, September 2011.

[10] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[11] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, October 2009.

[12] Microsoft. The Manycore Shift. Technical report, Microsoft White Paper, November 2007.

[13] Tilera. TILE-Gx, TilePro, and Tile64 processors. `http://www.tilera.com/products/processors`, 2012.