Do we really need parallel programming or should we strive for parallelizable programming instead? [*]

João Cachopo
IST / INESC-ID
Joao.Cachopo@ist.utl.pt

Abstract

Should we develop better programming models for concurrent programming and instruct programmers to develop highly concurrent programs, or should we develop managed runtimes that automatically and transparently parallelize the programs that they write?

In this paper, I make the argument that these two approaches may be at odds with each other, and raise the question of whether we should instruct programmers in general to develop concurrent programs at all. I claim that doing so may in fact be hindering the potential of executing the programs in parallel, and argue that, instead, a better approach is to evolve the sequential programming models and the software development practices that we currently have such that they promote the development of **parallelizable programs**—that is, programs that are amenable to automatic parallelization.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Languages, Performance

Keywords Automatic parallelization, Thread-Level Speculation, Software Transactional Memory

1. Introduction

The architectural shift from single-processor single-core machines to multicore machines is touted by many as one of the major challenges that the software industry has ever faced, if not the biggest [14]. The reason is simple: Having more processors on a single-user laptop or desktop computer is useless if an application running in that computer cannot make effective use of those extra processors. The corollary of this seems to be that programmers must now deal explicitly with concurrent programming, something that is feared by most. Worse than that, if the trend of duplicating the number of available cores every other year continues, as expected, it will not take long until we have machines on our laps with tens or hundreds of cores. So, we are not only talking about shifting to concurrent programming; this seems to imply that we have to architect our programs to be highly parallel.

Unfortunately, the current state-of-the-practice programming model for concurrent programming makes this too difficult and error-prone. Thus, an obvious avenue of research to solve this problem is on the development of better programming models that make concurrent programming easier; most of the intense research on Transactional Memories [5, 11] follows this route.

Another approach, however, is to keep the sequential programming model that we are confortable with, and concentrate on parallelizing automatically the sequential programs that we write. Again, this has been the subject of much research, from the initial work on parallelizing compilers (e.g., [2, 15]), to the more recent work on Thread-Level Speculation (TLS) [4, 12, 13, 7].

In this paper, I make the argument that these two approaches may be at odds with each other, and raise the question of whether we should instruct programmers in general to develop concurrent programs at all. I claim that doing so may in fact be hindering the potential of executing the programs in parallel, and argue that, instead, a better approach is to evolve the sequential programming models and the software development practices that we currently have such that they promote the development of **parallelizable programs**—that is, programs that are amenable to automatic parallelization.

In the next Section I present some of the reasons why I believe that automatic parallelization may be a better approach for making effective use of the new multicore architectures. Then, I discuss why promoting concurrent programming in general may be counter-productive. Finally, I briefly present the direction that I think would be most beneficial for the programming languages' community to pursue.

[*] This work was supported by FCT through the RuLAM project (PTDC/EIA-EIA/108240/2008).

2. Why automatic parallelization

The idea of automatically parallelizing sequential programs is enticing: With it programmers do not need to change how they develop their programs to benefit from the new generation of parallel machines. Instead, their programs are executed in parallel by the underlying execution environment in a completely transparent way. Actually, as we all know, that is exactly what the majority of the commodity uniprocessors have been doing over the last few decades, by resorting to instruction-level parallelism to improve the execution speed of sequential programs [10]. But, of course, multiprocessors call for other approaches to extract parallelism from a program, such as the TLS mentioned before.

Even though the results obtained so far with the research on TLS have been disappointing, there is evidence that the limited results may be attributed to only exploring low-level, fine-grained parallelism, and that much better results may be obtained if we explore parallelism at higher levels of the applications [8, 6]. To pursue this idea, I started the RuLAM[1] project, which proposes to develop a managed runtime for TLS that relies on Software Transactional Memory to execute speculatively an unbounded portion of a program (a preliminary prototype of such a system is described in [1]).

Besides the obvious advantage of not having to change the programming model, automatic parallelization is attractive also when compared with the alternative of letting the programmer decide how to parallelize her program. The number of parallel tasks to create depends on several factors that are known only at runtime, such as the available number of processors, the load of the machine, or the data to be processed. Moreover, some of these factors may change over time, meaning that an optimal parallel program needs to be able to adapt continuously the number of parallel tasks to execute. The parallelization logic is, thus, a cross-cutting concern that is best kept separate from the rest of the program.

But, even if we conceive a programming model that allows us to describe which tasks to create and when to create them as a separate aspect of the program, it is hard to imagine how that could possibly be done for a highly parallel and complex program (other than the trivial embarrassingly parallel ones) that is composed of many independent and reusable modules: How to make the parallelization logic composable, and in such a way that all the compositions result in the optimal parallelization strategy of each program?

So, it seems unlikely that, in general, a programmer will be able to parallelize effectively a complex program, much in the same way that, in general, a programmer is not able to deal with the memory management of a complex program.

I believe that, to obtain reasonable parallelism for a complex program, the programmer will, at best, provide only advisory hints about her program that may help in the parallelization. But, if that is the case, then the parallelization is not done by the programmer any longer. Instead, there must be some automatic parallelization runtime in place.

[1] RuLAM stands for "Running Legacy Applications on Multicores."

3. Why automatic parallelization of concurrent programs is not a good idea

To the best of my knowledge, the automatic parallelization of concurrent programs is a largely unaddressed topic. So, if programmers start to develop concurrent programs, either they do it right, performance-wise, or they may be compromising the ability to benefit from automatic parallelization runtimes. That is, unless automatic parallelization research extends to address concurrent programs as well. Yet, doing so introduces some tough challenges.

The most promising approaches for automatic parallelization rely on speculative execution of tasks to explore higher levels of parallelism, which means that they must resort to some form of synchronization among the parallel tasks to prevent invalid executions. Likewise, when programmers explicitly parallelize their programs, they not only create the tasks to execute in parallel, but they also must synchronize those tasks. So, a first challenge for the automatic parallelization of concurrent programs is to ensure that the synchronization mechanisms are compatible. Otherwise, instruction interleaving unforeseen either by the programmer or by the automatic parallelization runtime may ensue. We could argue that this is not much different from current TLS approaches that may have to abort invalid speculative executions, but the semantics of concurrent programming turns this into a much harder problem.

Moreover, even if we assume that the synchronization mechanisms are compatible, parallel tasks spawned by the programmer may delay or abort other tasks created by the parallelization runtime. In fact, unless the parallelization strategies used by the programmer and the runtime are aligned or are completely orthogonal, which is unlikely, the end result will be a less than optimal system, because the parallelization runtime will not be aware of the programmer's intentions.

Finally, even though this is still an open topic of research, it is reasonable to assume that to get a good-performing parallelization runtime, there must be some level of cooperation between the creation of tasks to execute and the scheduling of those tasks. Again, that is something that may suffer interference from unexpected parallelization efforts made by the programmer.

In conclusion, these observations led me to believe that instructing programmers to try to parallelize their programs by using concurrent programming may actually hinder the potential parallelization of those programs. So, instead of advocating such a radical and disruptive shift in the programming model, I argue that everyday programmers should continue to develop their programs using the much simpler sequential programming model and that we should concentrate on giving them a runtime that automatically parallelizes their programs. To do this, I think that we need to make some changes either to the sequential programming model or to the programming idioms and patterns that are currently used, so that programs become parallelizable.

4. Parallelizable programming

Sequential programs are hard to parallelize automatically because they are rippled with data dependencies [16]. Yet, recent experiences of parallelizing existing sequential programs reported good speedups with either minimal changes to, or after some refactoring of, the original program [3, 9].

These experiences provide strong evidence that the sequential programming model may not be that far from being effectively parallelizable and that, after all, we may just need better automatic parallelization runtimes to be able to make use of the new multicore hardware. Also, from these experiences, it seems that many of the data dependencies of a sequential program are either benign or false. Yet, as automatic parallelization approaches are conservative, they abort the speculative execution of parallel tasks when they find a data dependency, regardless of whether they needed to do it to preserve the program semantics.

Thus, even if theoretically an automatic parallelization framework may be made to detect those cases and handle them more gracefully, it seems clear that another way to make the automatic parallelization easier is to make the programs more parallelizable.

How to accomplish that? I expect this to be a question that may be best answered by the research communities working on programming language design and software engineering. New programming idioms, extensions to the existing sequential programming model, or best practices on how to design the API of libraries, are some examples of the contributes that could make a sequential program more amenable to being parallelized automatically.

We surely need more research on how to do concurrent programming, but it is not clear to me that we should rely on the everyday application programmers to do the parallelization of their programs. I believe that we may get better results by slightly changing how sequential programs are written.

References

[1] Ivo Anjo. JaSPEx: Speculative Parallelization on the Java Platform. Master's thesis, Instituto Superior Técnico, November 2009.

[2] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, et al. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, 1994.

[3] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84. IEEE Computer Society Washington, DC, USA, 2007.

[4] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[6] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[7] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, New York, NY, USA, 2006. ACM.

[8] JT Oplinger, DL Heine, and MS Lam. In search of speculative thread-level parallelism. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, 1999.

[9] V. Pankratius, A. Jannesari, and W.F. Tichy. Parallelizing bzip2: A case study in multicore software engineering. *Software, IEEE*, 26(6):70 –77, nov. 2009.

[10] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[11] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[12] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2000. ACM.

[13] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[14] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[15] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[16] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.