Lock-free Parallel Programming in an Event-driven and Side-effecting World

Alexander Herz

Computer Science
Technische Universität München
Boltzmannstr. 3
85748 Garching
Germany

Date: 12.02.2010
Supervisor: Prof. H. Seidl

Abstract

Parallelization of software is hard because multiple problems must be solved simultaneously to be successful. A sequential program must be decomposed into independent tasks, the granularity of the individual tasks must be sufficiently large and the side-effects inside these tasks must be correctly synchronized in order to preserve the semantics of the program. Pure functional languages like multi-core Haskell [1] trade the complexity of synchronization for fine granularity tasks, the burden of synchronizing side-effects like I/O explicitly and the inability to mix event-driven and pure program parts. While simplifying the development of parallel software, performance is not comparable to well crafted imperative implementations. To resolve these problems the non-deterministic, implicitly parallel language ***FunkyImp*** is introduced. Its semantics shall enable the user to write lock-free programs which's semantics are invariant under parallelization and produce large task grain sizes without undue constraints on the parallel execution of side-effects or the processing of events.

Introduction

Ever since distributed computing became available, researchers have striven to alleviate the problems inherent to the parallelization of programs. Among the most proliferated utilities in the scientific community are the OpenMP [2] library for SMP systems and the MPI [3] library for distributed systems. Both libraries provide portable support for thread coordination and communication. The sole responsibility of partitioning the program into parallel sections and providing correct synchronization is still left with the programmer unless a simple parallel for loop is to be implemented. Algorithmic skeletons, as provided e.g. by Intel's TBB [4], support a richer set of parallel patterns and data structures while abstracting from the low level thread management. Parallelizing non-trivial programs using TBB can be very complicated [5] as the programmer is forced to compose the complete software using the available skeletons. On the other hand, several languages have been introduced to support new parallel programming models. PGAS languages like UPC [6] expose a uniform memory model to the programmer. Although the management of the possibly clustered memory is hidden from the user, the programmer is still responsible for managing memory access latency and synchronization issues. Pure functional languages like Haskel [7] have been modified for parallel programming in order to exploit the absence of side-effects thus completely removing synchronization issues inside pure functions. Haskell's laziness requires the programmer to take into account many subtle evaluation properties of the language [8, 9] in order to achieve any actual speed up. Haskell separates side-effecting expressions from the pure part of the program using monads [10]. This has several awkward consequences for parallelization. The code inside a monad (especially the I/O monad) represents an imperative and possibly side-effecting sub-language which can be parallelized only using traditional methods and explicit synchronization reintroducing all problems attached to this method. Parallel execution of Haskell programs is commonly based on (lazy) parallel graph reduction [11]

which together with the lazy evaluation yields rather small task grain sizes and hence unimpressive performance when compared to imperative implementations [12]. Since CPU clock speeds have ceased to scale and multi-core processors are the norm, the industry has been eager to find new ways to simplify parallel programming. Increased research activity on verification of parallel programs in terms of race detection through abstract interpretation and trace based analysis highlights the need for improved methods [13, 14].

Within the last year we have developed the semantics for the new parallel language ***FunkyImp***. ***FunkyImp***'s single assignment core inherits the side-effect free properties of a pure language while the linear type system [15] allows to interleave side-effecting external functions like I/O with the pure statements. Hence, ***FunkyImp*** allows to perform implicit parallelization of pure and side-effecting statements without explicit synchronization. The strict evaluation semantics of ***FunkyImp*** enables the adaption of the task grain size and the exploitation of high level parallelism without the need to handle possibly diverging lazy terms. The performance of the strict program evaluation becomes more predictable and can be mapped onto standard hardware more easily than a parallel graph reduction. Furthermore, the introduction of event objects, actor like objects with reduced message send and receive guarantees, allows to perform push and pull based event processing inside the language while retaining semantic invariance under reordering (or parallel execution) of functions without data-dependencies. This feature overcomes the need to do functional reactive programming [16] and allows to implement non-deterministic parallel algorithms. Finally, the language was enhanced by functional data-parallel array processing capabilities.

Although there is a lot of literature available, praising the success of pure functional languages, non of these languages has seen wide adoption for parallel programming. Often, these languages were not designed from the ground up to be suitable for parallel execution but modified versions of sequential languages, impaired by their legacy.

Literatur

[1] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*, page 65, Edinburgh, Scotland, 2009.

[2] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 2002.

[3] W. Gropp, E. Lusk, and A. Skjellum. Using MPI: portable parallel programming with the message passing interface. 1999.

[4] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[5] Deepankar Bairagi Liang T. Chen. Developing Parallel Programs — A Discussion of Popular Models.

[6] C. et al. Coarfa. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[7] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *ACM SIGPLAN Notices*, 44(9):65–78, 2009.

[8] T. Uustalu and V. Vene. *Advanced Functional Programming*. Springer-Verlag Berlin/Heidelberg, 2005.

[9] P. W. TRINDER, H.-W. LOIDL, and R. F. POINTON. Parallel and distributed haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.

[10] M.P. Jones and P. Hudak. Implicit and explicit parallel programming in Haskell. *Disponível por FTP em nebula. systemsz. cs. yale. edu/pub/yale-fp/reports/RR-982. ps. Z (julho de 1999)*.

[11] P.W. Trinder, K. Hammond, JS Mattson Jr, AS Partridge, and SL Peyton Jones. GUM: a portable parallel implementation of Haskell. *ACM SIGPLAN Notices*, 31(5):79–88, 1996.

[12] H.-W. et al. Loidl. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[13] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2010.

[14] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155, 2009.

[15] P. Wadler. *Linear types can change the world*, volume 2, page 347–359. 1990.

[16] P. Hudak. Functional Reactive Programming. *Programming Languages and Systems*, pages 67–67, 1999.