The Random Traversal Technique for Parallel Evaluation of Functional Programs

Arthur Peters
Portland State University
Portland, Oregon, USA
`amp4@cs.pdx.edu`

Many functional programs contain significant amounts of natural parallelism, but taking advantage of this parallelism in a scalable way is difficult. Previous work on parallel evaluation of functional programs has focused on spawning evaluations in other threads. The threads must synchronize to notify other threads of work and perform load balancing. This synchronization is generally slow so the evaluations must be large enough to make the synchronization worth the effort.

This paper describes the *random traversal* technique for parallel evaluation of graph rewrite systems. This technique uses randomized traversals to allow multiple threads to work on the same evaluation with little or no synchronization. However, because the traversals are random the threads may end up doing the same work. In essence the random traversal technique trades some duplicated computation for a reduction in the amount of synchronization. However computation is faster than synchronization on modern hardware so this results in a net performance improvement. The random traversal technique does not require any special hardware support and can be implemented on current hardware. Currently only deterministic inductively sequential graph rewrite systems are supported but this technique could be extended to support non-deterministic graph rewrite systems using the *basic scheme* [Antoy and Peters, 2012]. This would allow it to be used to evaluate functional logic programs.

1 The Random Traversal Technique

The random traversal technique uses a needed rewrite strategy that works by traversing the expression based on definitional trees [Antoy, 1992]. Any subexpression that is encountered during this traversal is *needed* (must be evaluated to complete the overall evaluation). Parallelism can easily be extracted whenever more than one subexpression is needed. In the random traversal technique each thread randomly chooses an order in which to evaluate the needed subexpressions. Each thread will, on average, make a different choice so that multiple threads are unlikely to do the same work. Therefore, if multiple threads start evaluating the same expression then they will diverge and rewrite different subexpressions, exploiting parallelism.

When an expression e is evaluated, each thread stores the replacement back to e as soon as possible so that if another thread reaches e later it will find the rewritten version instead of duplicating work that has already been done. However, if two threads encounter e at the same time, they will both begin rewriting it and then try to store the replacement. This results in duplicated work, but cannot produce incorrect results because both threads are rewriting e using the same deterministic rules and will produce the same replacement. No synchronization is needed because it does not matter if one thread overwrites the rewrite performed by another thread. This lack of synchronization is the primary performance advantage of the random traversal technique.

As an example, if the expression `let x = f 10, y = f 11 in (x, y)` is evaluated by two threads, A and B, one possible evaluation is: thread A evaluates the left subexpression (`x`) first, and thread B evaluates the right subexpression (`y`) first. Thread A finishes its work and

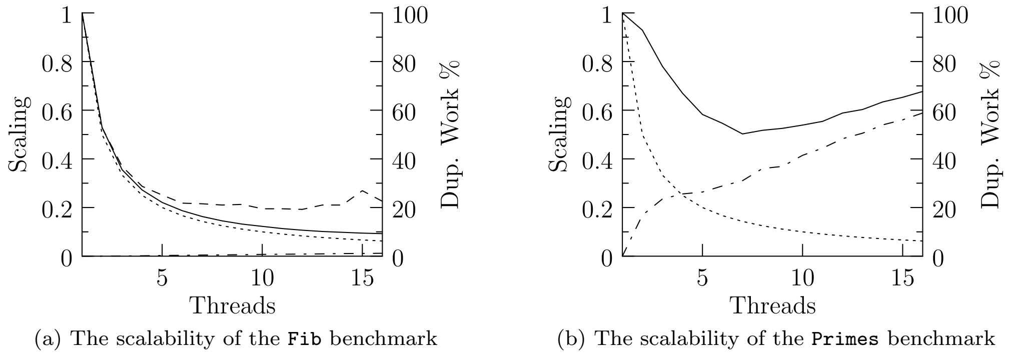

Figure 1: Scalability Benchmarks for `Fib` and `Primes`. The solid line (——) shows the scaling of the random traversal technique. The dashed line (- - -) shows the scaling of a Haskell implementation of the `Fib` benchmark running on GHC. The dotted line (······) shows ideal scaling. The dash-dotted line (– · –) shows the amount of work that was duplicated during the evaluation. All scaling numbers are computed as the execution-time of the specific number of threads divided by the single-thread execution-time of the same implementation.

examines `y`, but thread B has nearly finished that evaluation, so both threads perform the final steps duplicating a small amount of work. The threads have evaluated x and y in parallel.

To allow multiple threads to perform rewrites concurrently, all rewrites must be performed atomically, so that other threads can never see a node in an invalid state. Rewrites are allowed to happen concurrently with reads on the same node. No locking is used. Instead the system uses RCU primitives [Desnoyers et al., 2012] to publish the rewrite in a single atomic step. This publish still allows threads to see unrelated rewrites performed by other threads in different orders. However, this is safe because threads will always see the graph in some valid state.

The prototype implementation is written in C and runs on x86-64 processors. The test programs were written by hand in C following the random traversal technique instead of being automatically generated from a functional language. However it would not be difficult to automatically generate this code as the process is very mechanical. The current implementation uses a simple but inefficient rewriting technique where the expression being evaluated is represented as data and rewritten in-place.

The benchmarks shown in Figure 1 were performed on a 16-core Intel Xeon using two test programs: `Fib` is a non-memoizing Fibonacci benchmark and `Primes` iterates a list of numbers searching for primes. `Fib` scales very well (better than a Haskell implementation of the same benchmark) with a very small amount of duplicated work. `Primes` does not scale as well as expected. It should scale well because every number can be checked in parallel. The performance difference between the two benchmarks could result from various properties of the algorithm and implementation including the properties of the random number generator used for the traversal and errors in the implementation of `Primes`. This will be investigated.

The preliminary results are promising and show that randomization of evaluation order can allow for parallel evaluation without the need for synchronization between threads. The current implementation is inefficient, but it may be possible to develop a virtual machine that performs this kind of randomized traversal at some level of granularity. This research could provide a useful alternative to traditional task scheduling techniques, that is more efficient on modern hardware where computation is faster than synchronization.

References

S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, Kobe, Japan, May 2012. Springer LNCS 7294. To appear.

M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, Feb. 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2011.159.