Brief Announcement: Serial-Parallel Reciprocity in Dynamic Multithreaded Languages

Kunal Agrawal
Washington University at St Louis
kunal@cse.wustl.edu

I-Ting Angelina Lee
MIT Computer Science and Artificial Intelligence Laboratory
angelee@mit.edu

Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
sukhaj@mit.edu

ABSTRACT

In a dynamically multithreaded platform that employ work stealing, there seems to be a fundamental tradeoff between providing provably good time and space bounds and supporting ***SP-preciprocity***, a property that allows arbitrary calling between parallel and serial code, including legacy serial binaries. Many known dynamically multithreaded platforms either fail to support SP-reciprocity or sacrifice on the provable time or space bounds that an efficient work-stealing scheduler could otherwise guarantee.

We describe PR-Cilk, a design of a runtime system that supports SP-reciprocity in Cilk and provides provably efficient bounds on time and space. In order to maintain the stack space bound, PR-Cilk uses "subtree-restricted work stealing." We show that with subtree-restricted work stealing, PR-Cilk provides the same guarantee on stack space usage as ordinary Cilk. The completion time guaranteed by PR-Cilk is slightly worse than ordinary Cilk. Nevertheless, if the number of times a C function calls a Cilk function is small, or if each of these Cilk functions called by C functions are sufficiently parallel, PR-Cilk still guarantees linear speedup.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming

General Terms: FIX ME! Algorithms, Performance, Theory

Keywords: FIX: Cilk, dynamic multithreading, Intel Threading Building Blocks, scheduling, work stealing

1. INTRODUCTION

Work stealing [3, 5, 6, 4, 7, 9, 10, 11, 12, 13, 15, 17, 18, 23, 27] is fast becoming a standard way to load-balance dynamic multithreaded computations on multicore hardware. Concurrency platforms that support work stealing include Cilk-1 [4], Cilk-5 [12], Cilk++ [22], Fortress [2], Hood [6], Java Fork/Join Framework [19], Task Parallel Library (TPL) [21], Threading Building Blocks (TBB) [24], and X10 [8]. Work stealing admits an efficient implementation that guarantees bounds on both time and stack space [5, 12], but many existing implementations that meet these bounds — including Cilk-1, Cilk-5, and Cilk++ — do not exhibit ***series-parallel reciprocity***, or ***SP-reciprocity***[20] for short, *i.e.*, the property of allowing arbitrary calling between parallel and serial code — including legacy (and third-party) serial binaries. Without SP-reciprocity, it can be difficult to integrate a parallel library into existing legacy code base.

Unfortunately, supporting SP-reciprocity in a concurrency platform that employs work stealing often weakens the bounds on program completion time or stack space comsumption that the platform could otherwise provide. [1] For instance, TBB supports SP-reciprocity and employs a heuristic referred as "depth-restricted work stealing" [26] to limit stack space usage, but does not provide a provable time bound. In [20], the authors propose a modification to the Cilk-5 runtime that provides provable time and space bounds and supports SP-reciprocity, but their system requires additional operating system support. In addition, the space bound of [20] is slightly weaker than ordinary Cilk.

In this work, we present another point in the design space for work-stealing concurrency platforms, referred as ***PR-Cilk***, that employs a heuristic referred as "subtree-restricted work stealing". PR-Cilksupports SP-reciprocity, provides the same space bound as Cilk, and provides a provable but slightly weaker time bound as compared to ordinary Cilk. To be more precise, let T_1 be the ***work*** of a deterministic computation (its serial running time), and let T_∞ be the ***span*** of the computation (its theoretical running time on an infinite number of processors). Let V be the number of Cilk functions which are called from some C function, and let $\widetilde{T}_\infty$ be the "aggregate span" of the computation, where $\widetilde{T}_\infty$ is bounded by the sum over all the spans for each of the V Cilk functions. We prove that PR-Cilk can execute the computation on P processors in expected time $\mathrm{E}[T] = O(T_1/P + \widetilde{T}_\infty + V \lg P)$. We do not present the proof due to space constraints, but to summarize, this bound achieves linear speedup when V is small, or when each of the V Cilk functions has "sufficient parallelism." As for space, PR-Cilk achieves the same space bound as Cilk; if S_1 is the stack space usage of a serial execution, then the stack space S_P consumed during a P-processor execution satisfies $S_P \leq PS_1$. In contrast, to our knowledge, no analogous time bound exists when one uses depth-restricted work-stealing from TBB, an existing scheme which also addresses the problem of supporting SP-reciprocity by making a tradeoff on time bound.

2. DIFFICULTY OF SP-RECIPROCITY

This section describes how a work-stealing scheduler operates, why the "cactus stack" abstraction is necessary, and explains why SP-reciprocity is difficult to obtain. We use Cilk-5 [12] as a model, because it is the system which PR-Cilk is based on.

Cilk's work-stealing scheduler

In Cilk, the programmer denote the logical parallelism of the program by using the keywords such as **spawn** and **sync**.When a function call is preceded by the keyword **spawn**, the ***parent*** function

[1] Although Fortress, Java Fork/Join Framework, TPL, and X10 employ work stealing, they do not suffer from the same problems, because they are byte-code interpreted by a virtual-machine environment.

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.
ACM 978-1-4503-0079-7/10/06.

spawns the ***child*** function, invoking the child without suspending the parent, thereby creating parallelism. The complement of **spawn** is the keyword **sync**, which acts as a local barrier, indicating that the control shall not pass the **sync** statement until all previously spawned functions have returned.

Cilk's work-stealing scheduler load-balances parallel execution across the available ***worker*** threads while respecting the logical parallelism denoted by the programmer. Cilk follows the "lazy task creation" strategy of Kranz, Halstead, and Mohr [17], where the worker suspends the parent when a child is spawned and begins work on the child.Operationally, when the user code running on a worker encounters a **spawn**, it invokes the child function and suspends the parent, just as with an ordinary subroutine call, but it also places the parent frame on the bottom of a ***deque*** (double-ended queue). When the child returns, it pops the bottom of the deque and resumes the parent frame. Pushing and popping frames from the bottom of the deque is the common case, and it mirrors precisely the behavior of C or other Algol-like languages in their use of a stack.

A worker exhibits behavior that differs from ordinary serial stack execution if it runs out of work. This condition can happen due to two cases. First, the worker may stall at a **sync** in a function because some of the function's spawned children have not yet returned. Second, the worker may return from a function and find that its deque is empty (i.e., all its ancestor frames are stolen).[2] When the worker has no work, the worker becomes a ***thief***, and and attempts to steal the topmost frame from a randomly chosen ***victim*** worker. If the steal is ***successful***, the worker resumes the stolen frame. If the victim has no work, the thief picks another worker randomly and attempts to steal again.

Cilk's support for the cactus-stack abstraction

An execution of a serial Algol-like language, such as C [16] or C++ [25], can be viewed as a "walk" of an ***invocation tree***, which dynamically unfolds during execution and relates function instances by the "calls" relation: if function instance `A` calls function instance `B`, then `A` is a ***parent*** of the ***child*** `B` in the invocation tree. Such serial languages use a ***linear-stack representation***: When a function is called, the callee's stack is allocated right underneath the caller's stack by advancing the stack pointer, and when a function returns, the stack pointer is restored to point to the caller's stack. This scheme is space-efficient, because all the children of a given function can use the same region of the stack.

The notion of the invocation tree can be extended to include spawns, as well as calls, but unlike the serial walk of an invocation tree, a parallel execution unfolds the tree more haphazardly and in parallel. Since multiple children of a function may be extant simultaneously (due to spawns), a linear-stack data structure no longer suffices for storing activation frames. Instead, the tree of extant activation frames forms a ***cactus stack*** [14].

Cilk supports the cactus stack abstraction by allocating frames for Cilk functions in noncontiguous space, where each frame is linked to its parent frame. These frames in the noncontiguous memory are referred as ***shadow frames*** to differentiate from the ***activation frames*** in the linear stacks. As a result, the call/return linkage for a Cilk function, referred as the ***Cilk linkage***, differs from the ordinary C linkage: a Cilk function passes parameters and returns value via its shadow frame. That means, if a parent passes a pointer of its local variable to its child, the pointer refers to the location in the shadow frame. Moreover, multiple extant children can share a single view of their parent frame simultaneously.

This shadow stack strategy allows Cilk to provide a provable space bound, but does not allow SP-reciprocity. When a worker's deque is empty, the worker can pop the suspended activation frames in its linear stack (since there is no pointers to variables in the frames elsewhere in the system). Since a worker only steals when its deque (and therefore its stack) is empty, each worker uses no more stack space than the space used by the serial execution of the program. However, this stategy uses the Cilk linkage to spawn, which is incompatible with the ordinary C linakge. A sharp delineation exists between C and Cilk: while a Cilk function may call a C function, a C function may not call back to a Cilk function, unless the C function is also recompiled to use the special Cilk linkage.

Other alternatives

As an alternative to shadow stack, one can preserve SP reciprocity by using linear stacks to implement cactus stacks. For example, if a Cilk function `A` executing on worker p has multiple extant children, other workers executing these extant children may share a single view of `A`'s frame sitting in p's stack space. However, this strategy compromises either the completion time or stack space bound because of the fact that once a frame has been allocated, its location in virtual memory cannot be changed, because there may be a pointer to a variable in the frame elsewhere in the system. Thus, if `A`'s frame is shared among workers, p cannot reuse the stack space where `A` resides until all `A`'s extant children return. Now, if p runs out of work before `A` is ready to return, p has two options. In the first option, p can block and wait for `A`'s children to complete. This alternative causes workers to block and therefore no longer provides the near-optimal completion time guarantee that Cilk provides. In the second option, p can go steal work from some other worker. In this case, p has no choice but to push the stolen work onto its stack below `A`.[3] If `A` is already deep in the stack, and the stolen work is close to the top of the invocation tree, p's stack can grow twice as deep as what it would be in a serial execution. Furthermore, this scenario could occur recursively, consuming impractically large stack space.

TBB operates on linear stacks with ordinary linkage and thus provides SP-reciprocity. In order to avoid this large space consumption, TBB employs ***depth-restricted work stealing***, where a worker is restricted to steal only tasks which are deeper than the worker's deepest blocked task, thereby limiting the space consumption. The fact that a thief can steal from arbitrary part of the invocation tree (as long as the depth restriction is not violated) makes it difficult to prove a non-trivial upper bound on the completion time, however. For a lower bound, [26] exhibits a computation for which TBB runs asymptotically serially because of depth-restricted work stealing, but for which Cilk can achieve linear speedup.

3. PR-Cilk DESIGN

PR-Cilk supports SP-reciprocity and provable time and space bounds by using a strategy called ***subtree-restricted work stealing***. In addition, PR-Cilk uses shadow frames of Cilk functions and the regular C activation frames for C functions. Some modifications to the runtime system and the compiler are required in order to support transitioning between two different types of frames and linkages. Due to space limit, however, we focus our attention on how PR-Cilk supports subtree-restricted work stealing using "parallel regions", a mechanism adapted from HELPER [1].

[2] In this second case, the worker first checks whether the parent is stalled on a **sync** and whether this child is the last child to return. If so, it resumes the parent function passing the stalling **sync**.

[3] We assume the stack grows downward.

To remind ourselves of the problem, suppose a worker p executes a C function `foo` which calls a Cilk function `A`. Since the C function uses the activation frame, the stack space associated with `foo` can not be removed from p's stack until all the descendants of `A` in the invocation tree are completed. If p runs out of the work before all children of `A` finish, then, as we mentioned earlier, it must either block and wait for its extant children to complete (thereby sacrificing the time bound) or steal (potentially consuming excessive space). PR-Cilk solves this problem by using ***subtree-restricted work stealing***, which forces p to steal from only within `A`'s subtree in the invocation tree. Notice that, in any serial execution, the stack depth of any frame within `A`'s subtree is greater than the stack depth of `A`, where p is stalled. Therefore, no processor can use more stack space than the serial execution, and we maintain the Cilk stack space bound. Furthermore, any work p steals is work that must be completed in order for `A` to return, and p is (in some sense) helping to complete its own work.

By default in Cilk a worker is only allowed to steal from the top of a deque; Cilk has no mechanism for limiting work stealing to some subtree of the invocation tree. PR-Cilk augments the Cilk-5 runtime system with ***parallel regions*** in order to allow subtree-restricted work stealing. A parallel region construct for Cilk was originally described in HELPER [1] as a way of supporting nested parallelism in locked critical sections. Here, we use the term ***parallel region*** to refer to a subcomputation with nested parallelism whose root represents a Cilk function called by a C function, but the mechanism for supporting parallel regions is almost identical.

Conceptually, each parallel region $R_{\texttt{A}}$ is an instance of a Cilk function that uses its own ***deque pool*** — a set of deques — for self-contained scheduling. When a worker p starts a parallel region $R_{\texttt{A}}$, the runtime system creates a new deque pool for $R_{\texttt{A}}$, denoted by $\texttt{dqpool}(R_{\texttt{A}})$. The runtime system allocates a deque $q \in \texttt{dqpool}(R_{\texttt{A}})$ to a worker p when p is ***assigned*** to $R_{\texttt{A}}$. In order to support nested parallel regions, each worker p maintains a chain of deques, each for a different region, with the bottom deque in the chain being p's active deque. Whenever a worker p tries to steal, it only steals from deques in the same pool as p's active deque.

We can directly use the design of parallel regions to support subtree-restricted work stealing in PR-Cilk. When worker p calls a Cilk function `A` from a C function `foo`, it implicitly invokes a function called `start_region`. The `start_region` call causes p to start a new region, which involves p creating a new deque pool $\texttt{dqpool}(R_{\texttt{A}})$ and creating a new deque q for itself in $\texttt{dqpool}(R_{\texttt{A}})$. After creating the region, p continues to execute `A`, which may spawn more functions under region $R_{\texttt{A}}$, and the frames associated with these functions are added to q. Other workers may later be assigned to this region and steal from q. Any additional work created by these workers within $R_{\texttt{A}}$ is added to some deque in $\texttt{dqpool}(R_{\texttt{A}})$ as well. If p later stalls on a **sync** in `A`, it can now steal work from any deque in the pool $\texttt{dqpool}(R_{\texttt{A}})$, since such work belongs to the subtree rooted at `A`.

Since PR-Cilk uses the same policy for workers entering and leaving parallel regions as described in [1], the completion time and stack space bounds in [1] can be simplified and applied directly to PR-Cilk. PR-Cilk computations have more structure than HELPER, however. Specifically in PR-Cilk, regions are not associated with locks, so a worker is assigned to a region only by either starting the region or stealing into the region, whereas in HELPER, a worker can also be assigned to a region via acquiring a lock associated with the region. Given this property, we believe that the time bound may be improved. As future work, we plan to improve the time bound, and possibly explore the implications of having different entering and leaving policies for parallel regions, as the completion time may be affected depending on the policy adapted.

Acknowledgments

Thanks to Matteo Frigo of Axis Semiconductor, and the Supertech group members of MIT CSAIL for helpful discussions.

4. REFERENCES

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Helper locks for fork-join parallel programming. In *PPoPP '10*, Jan. 2010.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., Mar. 2008.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129, June 1998.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[6] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[7] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81*, pages 187–194, Oct. 1981.

[8] 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 *OOPSLA '05*, pages 519–538. ACM, 2005.

[9] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *SPAA '94*, pages 94–103, June 1994.

[10] R. Finkel and U. Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, Apr. 1987.

[11] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *OSDI '94*, pages 201–213, Nov. 1994.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[13] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, Oct. 1985.

[14] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[15] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[17] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI '89*, pages 81–90, June 1989.

[18] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, MIT Department of EECS, May 1994.

[19] D. Lea. A Java fork/join framework. In *Java Grande Conference*, pages 36–43, 2000.

[20] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using thread-local memory mapping to support cactus stacks in work-stealing runtime systems. Submitted for publication.

[21] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA '09*, pages 227–242, 2009.

[22] C. E. Leiserson. The Cilk++ concurrency platform. In *46th Design Automation Conference*. ACM, July 2009.

[23] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *LCPC '94*, Aug. 1994.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, third edition, 2000.

[26] J. Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *SPAA '09*, Aug. 2009.

[27] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, Aug. 1988.