Brief Announcement: A Lower Bound for Depth-Restricted Work Stealing

Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
sukhaj@mit.edu

ABSTRACT

Work stealing is a common technique used in the runtime schedulers of parallel languages such as Cilk and parallel libraries such as Intel Threading Building Blocks (TBB). ***Depth-restricted*** work stealing is a restriction of Cilk-like work stealing in which a processor blocked on a task at depth d can only steal tasks from other processors at depth greater than d. To support programs coded in a blocking style, i.e., code without explicit continuations, TBB imposes a depth restriction on work stealing to limit the stack space used by a computation.

We present a lower bound on the completion time of a computation executed using a depth-restricted work-stealing scheduler. In particular, we construct a computation which on P processors runs a factor of $\Omega(P)$ slower with depth-restricted work stealing as compared to unrestricted work stealing. On this pessimal computation, depth-restricted work stealing asymptotically serializes execution while unrestricted work stealing achieves linear speedup.

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

General Terms: Algorithms, Performance, Theory

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

1. INTRODUCTION

Work stealing is a common scheduling technique used in the runtime systems of dynamically multithreaded languages and libraries. In a dynamically multithreaded system, programmers typically specify which tasks in a program have the potential to execute concurrently. Then, when the program executes, a runtime system dynamically schedules the tasks on some number of processors P. With a work-stealing runtime scheduler, whenever a processor runs out of work or waits for a task to complete at a synchronization point, the processor attempts to steal work from another processor. Many dynamically multithreaded systems use a randomized work-stealing algorithm modeled after the work-stealing scheduler in Cilk [3], an efficient parallel programming language. The efficiency of Cilk programs stems in part from Cilk's provably efficient work-stealing scheduler [1].

The efficiency of Cilk also depends, however, on the ability of the Cilk compiler to generate function continuations. Continuations allow a function to begin execution on one processor, but resume execution on another processor (e.g., after a steal occurs). Unfortunately, parallel libraries, such as Intel Threading Building Blocks (TBB) [4], typically can utilize continuations only when they are explicitly provided by the programmer. Thus, such libraries usually support additional, more convenient interfaces which do not require coding explicit continuations.

This research was supported in part by NSF Grants CNS-0615215 and CNS-0540248.

Copyright is held by the author/owner(s).
SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.
ACM 978-1-60558-606-9/09/08.

```
 1 class BTask: public task {
 2  int* sum;
 3  int x, y;
 4  BTask(int* sum_) : sum(sum_)
 5  task* execute() {
 6    XTask& t_x;  YTask& t_y;
 7    t_x = *new(allocate_child()) XTask(&x);
 8    t_y = *new(allocate_child()) YTask(&y);
 9    spawn(t_y);
10    spawn_and_wait_for_all(t_x);
11    *sum = x + y;
12  }
13 }
```

Figure 1: A blocking-style computation using TBB pseudocode. Code for reference counting tasks has been omitted.

For example, TBB allows programmers to write ***blocking-style*** code, as shown in Figure 1. In this code, `BTask` is a task which computes two values, x and y (using two parallel subtasks, `XTask` and `YTask`), and then computes the sum $x+y$. To understand the behavior of blocking-style code, suppose processor p_1 begins executing `BTask`. In line 9, p_1 spawns `YTask` for other processors to potentially steal, and then starts working on `XTask` in line 10. Suppose another processor p_2 steals `YTask` from p_1, and then p_1 finishes `XTask` before p_2 completes `YTask`. Then p_1 stalls at line 10, and p_1 begins trying to randomly steal work. If `BTask` had a continuation, then p_1 could clear the stack space used by `BTask`, since p_2 could resume the continuation of `BTask` after p_2 completes `YTask`. With blocking-style code, however, p_1 cannot clear this stack space, since p_1 must eventually resume `BTask` at line 11.

To avoid a significant growth in stack space due to a processor repeatedly blocking and stealing, TBB uses what we call ***depth-restricted work stealing***, that is, TBB constrains a processor to only steal tasks which are deeper than the processor's deepest blocked task. As the TBB documentation states, this restriction on Cilk-like work-stealing may limit the available parallelism and impact performance [4]; however, no theoretical analysis is presented.

How restrictive can depth-restricted work stealing be, as compared to unrestricted work stealing? We construct a computation which, when executed using depth-restricted work-stealing on P processors, runs $\Omega(P)$ times slower than when executed using unrestricted work-stealing. Thus, there exists a computation which could exhibit linear speedup when run on P processors, but which is asymptotically serialized by depth-restricted work stealing.

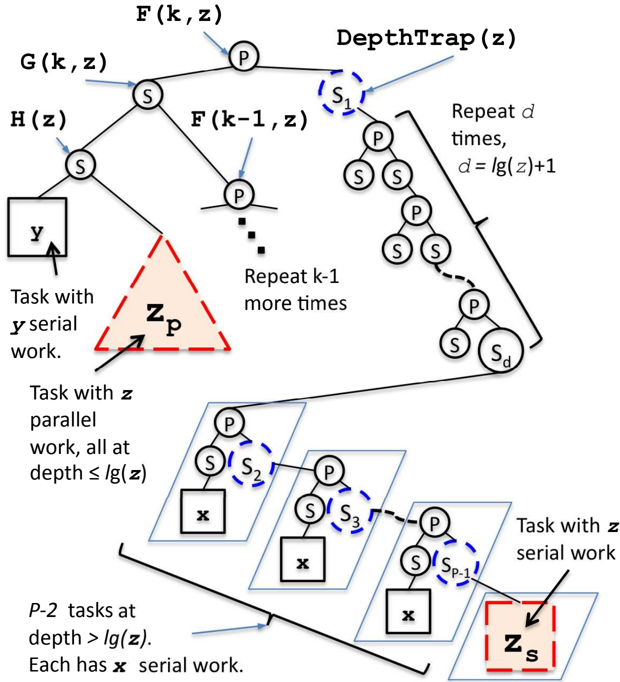

Figure 2: A series-parallel parse tree for F.

2. A PESSIMAL EXAMPLE

We present a parallel computation which exhibits linear speedup on P processors when executed by a runtime with an ordinary, unrestricted work-stealing scheduler, but which achieves only constant speedup when the runtime performs depth-restricted work stealing. First, we outline the general structure of our pessimal example. Then, we analyze the runtime of this computation using both depth-restricted and unrestricted work stealing.

Our pessimal example is generated by a method $\texttt{F}(k,z)$, which conceptually chains together k instances of $\texttt{F}(1,z)$. Figure 2 shows a series-parallel parse tree representation [2] of F. A parallel traversal of a series-parallel parse tree models an execution of a computation on multiple processors. The child subtrees of an ***S-node*** must be traversed serially, from left to right, while the child subtrees of a ***P-node*** can be traversed in any order. In this tree, a P-node corresponds to a blocking spawn. When a processor reaches a P-node, it begins work on the left child subtree. If the right subtree is stolen, and the processor finishes the left subtree, then the processor is blocked on the (S-node) root of the right subtree. We measure the depth of a task as the depth of P-nodes (i.e., nesting depth of spawns) in the tree.[1]

The subroutine $\texttt{F}(1,z)$ forms the core of the example; when executed on P processors using depth-restricted work stealing, $\texttt{F}(1,z)$ runs for at least z time, but completes only about $2z$ work. Intuitively, F spawns two tasks; one task G contains z potentially parallel work (subtask $\texttt{z}_p$), and the other task DepthTrap contains z serial work (subtask $\texttt{z}_s$). Ideally, $P-1$ processors should work on G and one should work on DepthTrap; however, DepthTrap begins with enough parallel work ($P-2$ tasks with serial work x) and G begins with enough serial work (y) so that $P-1$ processors steal work from DepthTrap and only one works on G. Once $P-1$ processors steal from DepthTrap, $P-2$ processors will block waiting for one processor to complete z serial work. Furthermore, since DepthTrap traps these $P-2$ processors at a depth greater than the depth of any work in $\texttt{z}_p$, the processors remain idle, even after G creates additional parallel work.

To form the complete example, we chain k repetitions of $\texttt{F}(1,z)$, arranged so that repetition j can begin only after the $\texttt{z}_p$ task of repetition $j-1$ is complete. F is designed so that with depth-restricted work stealing, DepthTrap finishes before G enables the next repetition of F. Then, the k instances of DepthTrap occur sequentially, and $\texttt{F}(k,z)$ requires at least kz time to execute.

Theorem 1 and Corollary 2 state these properties of F more formally. In F, we set the values of x and y depending on two functions, $X(\varepsilon)$ and $Y(\varepsilon,z)$. Intuitively, x must be larger than the time required for $P-1$ processors to complete $P-1$ successful steals, and $y>x$ must be large enough to ensure that $\texttt{H}(z)$ does not complete before $\texttt{DepthTrap}(z)$. We defer the proof of Theorem 1 until Section 3.

DEFINITION 1. *Let c_s be the maximum time for any steal attempt (successful or unsuccessful). Define $X(\varepsilon)=c_sP\ln\left(\frac{P}{\varepsilon}\right)(1+\ln(P))$. Define $Y(\varepsilon,z)=X(\varepsilon)+(P+c_s)\lg(z)+(P^2+c_s)$.*

THEOREM 1. *For an execution of $\texttt{F}(1,z)$ using depth-restricted work stealing, let T_H and T_D be the completion time of $\texttt{H}(z)$ and $\texttt{DepthTrap}(z)$, respectively. If $x\geq X(\varepsilon)$ and $y\geq Y(\varepsilon,z)$, then $T_H-T_D\geq 0$, i.e., $\texttt{H}(z)$ does not finish before DepthTrap.*

COROLLARY 2. *Let $x=X(\varepsilon/k)$ and $y=Y(\varepsilon/k)$. With probability at least $(1-\varepsilon)$, a depth-restricted work-stealing scheduler using P processors requires at least $\Omega(kz)$ time to execute $\texttt{F}(k,z)$.*

PROOF. By Theorem 1, the execution of $\texttt{F}(1,z)$, H(z) will not finish before $\texttt{DepthTrap}(z)$ with probability at least $(1-\varepsilon/k)$. At least z time is required to execute $\texttt{DepthTrap}(z)$. Thus, using a union bound over k repetitions of F, we know $\texttt{F}(k,z)$ requires at least $\Omega(kz)$ time with probability at least $(1-\varepsilon)$. □

A runtime using unrestricted work stealing can complete $\texttt{F}(k,z)$ quickly, however, because it can complete each z_p quickly (i.e., in $O(z/P)$ time), and overlap the executions of the serial $\texttt{z}_s$ tasks from the k repetitions of F. Lemma 3 states this result more formally, and Theorem 4 compares depth-restricted work stealing and unrestricted work stealing for $\texttt{F}(k,z)$.

LEMMA 3. *Let $x=X(\varepsilon/k)$ and $y=Y(\varepsilon/k)$. With probability at least $(1-\varepsilon)$, an unrestricted work-stealing scheduler using P processors can execute $\texttt{F}(k,z)$ in $O(kz/P+z)$ time, assuming that $z=\omega(kP^2\lg(kP/\varepsilon))$.*

PROOF. The proof follows from the analysis of Cilk [1], which has an unrestricted work-stealing scheduler; for a Cilk computation with work T_1 and span T_∞, the running time on P processors is $O(T_1/P+T_\infty+\lg(P/\varepsilon))$, with probability at least $(1-\varepsilon)$.

The span of F is $T_\infty\leq k(y+\lg(z))+z$. From our choices of x and y, we know $x=O(P^2\lg(kP/\varepsilon))$, and $y=x+O(P\lg(z)+P^2)$. If $z=\omega(kP^2\lg(kP/\varepsilon))$, then one can show that $ky=o(z)$ and $k\lg(z)=o(z)$. Thus, $T_\infty=O(z)$. Similarly, the work of F is $T_1=k((P-2)x+y+2z)+\Theta(k(\lg(z)+P))$. The $2kz$ term asymptotically dominates the other terms, so $T_1=O(kz)$. □

THEOREM 4. *There exists a computation for which the ratio of the runtime using a depth-restricted work stealing scheduler to the runtime using an unrestricted work-stealing scheduler is $\Omega(P)$.*

PROOF. Choose $k=\Omega(P)$ and $z=\omega(kP^2\lg(kP/\varepsilon))$. Then, the computation $\texttt{F}(k,z)$ satisfies Corollary 2 and Lemma 3, and we get a competitive ratio of $\Omega(P)$. □

[1] Our example can be generalized to some other definitions of depth or other work-stealing restrictions, as long as processors are prevented from stealing from $\texttt{z}_p$ once they enter DepthTrap.

3. PROOF DETAILS

In this section, we present the proof of Theorem 1. For completeness, we also give TBB pseudocode (Figure 3) corresponding to the tree in Figure 2.

Theorem 1 requires that we choose values for x and y large enough to make the behavior of random work stealing predictable. The appropriate values for x and y arise from the analysis in Lemma 5.

LEMMA 5. *Consider the execution of* `F`$(1,z)$ *using a depth-restricted work-stealing scheduler. If* $x \geq X(\varepsilon)$*, and* $y \geq Y(\varepsilon, z)$*, then, with probability at least* $(1-\varepsilon)$*,* $P-1$ *processors are stuck in* `DepthTrap`(z) *for at least* z *time.*

PROOF. For $i \in \{1,2,\ldots P-1\}$, let t_i be the time step when some processor begins working on node S_i in Figure 2. Similarly, let t_d be the time when some processor begins work on node S_d, i.e., a processor has reached the bottom of the chain of length $d = \lg(z)+1$ in `DepthTrap`. Intuitively, we prove Lemma 5 by showing that t_{P-1}, the time that a processor starts working z_s, is likely to satisfy $t_{P-1} \leq X(\varepsilon) + c_s d$. Then, since $y \geq Y(\varepsilon,z) > X(\varepsilon) + c_s d$, `H`$(z)$ does not generate any parallel work before time t_{P-1}, and processors must steal only from `DepthTrap`. Furthermore, if we have $t_{P-1} - t_d \leq X(\varepsilon) \leq x$, then, it is impossible for a processor to finish a serial block of work x before t_{P-1}. Then, we know each of the $P-2$ tasks with x serial work and task z_s must be executed by one of $P-1$ distinct processors. Finally, once $P-1$ processors have stolen from `DepthTrap`, they are trapped at a depth larger than the depth of any parallel work generated by `H`, waiting on z_s.

More precisely, to bound t_{P-1}, we construct events A_i which capture the notion that the t_i's are meeting their "likely" deadlines. Let A_1 be the event that $t_d \leq c_s d + c_s P\left(\frac{\ln(P/\varepsilon)}{P-1}\right)$. For $j \in \{2,3,\ldots P-1\}$, let A_j be the event $t_j \leq c_s d + c_s P\left(\sum_{i=1}^{j} \frac{\ln(P/\varepsilon)}{P-i}\right)$. We can show by induction that $\Pr(\bigcap_{i=1}^{j} A_i) \geq (1-\varepsilon/P)^j$. Substituting $j = P-1$ and simplifying the sum gives us $t_{P-1} \leq X(\varepsilon) + c_s d$ with probability at least $(1-\varepsilon)$.

In the base case for induction, we compute $\Pr(A_1)$. Notice that initially one processor begins work on `G`, and $P-1$ other processors attempt to steal S_1. Since processors steal randomly, the probability that S_1 has not been stolen after n steal attempts is at most $\left(1-\frac{1}{P}\right)^n < e^{-n/P}$. Thus, with $P-1$ processors stealing, S_1 is stolen before time $c_s P\left(\frac{\ln(P/\varepsilon)}{P-1}\right)$ with probability at least $(1-\varepsilon/P)$. Once work begins at node S_1, we know some processor must reach S_d in at most $c_s d = c_s(\lg(z)+1)$ time, (i.e., $t_d - t_1 \leq c_s d$), since in the worst case, steals happen for every right S-node on the chain of length d in `DepthTrap`. Thus, $\Pr(A_1) \geq (1-\varepsilon/P)$.

To bound $\Pr(A_1 \cap A_2)$, we first condition on A_1 occurring. Once a processor reaches S_d, it then quickly spawns S_2 and begins working on a block with serial work x. For times $t_d < t < t_2$, at least $P-2$ processors must be idle and trying to steal S_2. Thus, using the same analysis as for t_1, we know that $t_2 - t_d \leq c_s P\left(\frac{\ln(P/\varepsilon)}{P-2}\right)$ with probability at least $(1-\varepsilon/P)$. Thus, $\Pr(A_2|A_1) \geq (1-\varepsilon/P)$, and $\Pr(A_1 \cap A_2) = \Pr(A_2|A_1)\Pr(A_1) \geq (1-\varepsilon/P)^2$.

We complete the induction by repeating this analysis for the remaining A_j. Conditioned on the event that $\bigcap_{i=1}^{j-1} A_i$, we know for times t such that $t_{j-1} < t < t_j$, at least $P-j$ processors trying to steal S_j. Thus, $t_j - t_{j-1} \leq c_s P\left(\frac{\ln(P/\varepsilon)}{P-j}\right)$ with probability at least $(1-\varepsilon/P)$. Thus, $\Pr\left(\bigcap_{i=1}^{j} A_i\right) \geq (1-\varepsilon/P)^j$.

Finally, to show $t_{P-1} - t_d \leq X(\varepsilon)$ with probability at least $(1-\varepsilon)$, note that if we ignore S_1 and compute deadlines for $t_i - t_d$ instead of t_i, the same inductive proof used to bound t_{P-1} applies. □

```
void F(int k, int z) {
  spawn(DepthTrap(lg(z), z));
  spawn_and_wait_for_all(G(k, z));
}
void G(int k, int z) {
  H(z); if (k > 1) { F(k-1, z); }
}
void H(int z) {
  spawn(ParallelWork(z));
  spawn_and_wait_for_all(SerialWork(y));
}
void DepthTrap(int d, int z) {
  if (d == 0) { TrapProcessors(P-1, z); }
  else {
    spawn(DepthTrap(d-1, z));
    spawn_and_wait_for_all(SerialWork(1));
  }
}
void TrapProcessors(int i, int z) {
  if (i <= 1) { SerialWork(z); }
  else {
    spawn(TrapProcessors(i-1, z));
    spawn_and_wait_for_all(SerialWork(x));
  }
}
void ParallelWork(int n) {
  if (n <= 1) { doUnitWork(); }
  else {
    spawn(ParallelWork(n - n/2));
    spawn_and_wait_for_all(ParallelWork(n/2));
  }
}
void SerialWork(int n) {
  for (int i = 0; i < n; i++) { doUnitWork(); }
}
```

Figure 3: TBB pseudocode for `F`(k,z).

By using Lemma 5, we can bound the completion time of `H` and `DepthTrap`(z) and prove Theorem 1.

PROOF OF THEOREM 1. Using Lemma 5, since `DepthTrap` begins executing z_s at time t_{P-1}, and `DepthTrap` requires at most $z+P+\lg(z)$ additional time to finish, we have $T_D \leq t_{P-1} + z + \lg(z) + P$. Also from Lemma 5, with probability at least $(1-\varepsilon)$, we know `DepthTrap` keeps $P-1$ processors occupied and `H`(z) executes serially for at least $t_{P-1}+z$ time. Since `H` has at least $y+z$ work, it cannot finish before time $T_H \geq t_{P-1} + z + (y - t_{P-1})/P$. Then, we know $T_H - T_D \geq (y - t_{P-1})/P - \lg(z) - P$. By substituting $t_{P-1} \leq X(\varepsilon) + c_s(\lg(z)+1)$, we get that $T_H - T_D \geq 0$. □

4. REFERENCES

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

[2] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, Canada, 1998.

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