Doc. No.: N3872
Date: 2014-01-15
Reply to: Arch Robison <arch.robison@intel.com>

A Primer on Scheduling Fork-Join Parallelism with Work Stealing

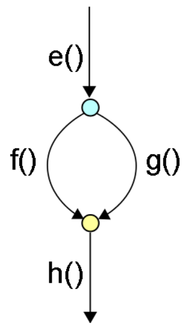
This paper is a primer, not a proposal, on some issues related to implementing fork-join parallelism. It is intended to introduce readers to two key design choices for implementing fork-join parallelism and their impact. Both choices concern how fork-join computations are mapped (scheduled) onto threads. The key point is that the most efficient resolution for these choices is the opposite of what programmers new to parallelism might consider "obvious". This paper also makes clear that the two choices can be made independently of each other.

The separate subject of how to map fork-join call stacks ("cactus stacks") onto memory is not covered. Reference [CilkM] surveys various methods for such.

Notation and Running Example

Cilk-like syntax is used for simplicity, though the fundamental issues apply to most ways to express fork-join programming. Figure 1 shows our first example, as code

```
e();
spawn f();
g();
sync;
h();
```



and a parallel control-flow diagram.

Figure 1: Fork-join code example and diagram of control flow.

The blue circle denotes where control flow forks into two strands of execution that may run in parallel. The yellow circle denotes where they must join before execution can continue afterwards.

Assume that expression `e()` runs on some thread. The questions to be explored are:

- What threads should `f()` and `g()` should run on?
- What thread should `h()` run on?

First Choice: What Thread Executes After a Spawn?

Work-stealing has become the method of choice for scheduling fork-join parallelism. It is used by MIT Cilk, Intel Cilk Plus, Intel TBB, Microsoft PPL, and OpenMP tasking. The basic notion is that threads create pieces of work. When a thread runs out of work, it becomes a *thief* that attempts to steal a piece of work from another thread.

In a work-stealing implementation of fork-join, such as in Figure 1, when control flow forks, the original thread executes one branch of the fork, and leaves the other branch to be stolen by a "thief". If the other branch is not stolen, the original thread eventually executes it after finishing the first branch.

An import design choice is which branch to offer to thief threads. Using Figure 1, the choices are:

- **Child Stealing**: `f()` is made available to thief threads. The thread that executed `e()` executes `g()`.
- **Continuation Stealing**: Also called "parent stealing". The thread that executed `e()` executes `f()`. The continuation (which will next call `g()`) becomes available to thief threads.

TBB and PPL use child stealing. Cilk uses continuation stealing. OpenMP uses child stealing by default, but offers programmers the ability to specify that continuation stealing is permitted.

At first glance the choice might look like a minor detail, particularly given the symmetry on the right side of Figure 1. But the choice can have an irksome semantic impact and serious impact on asymptotic space. Consider the following example, which spawns `n` tasks `f(0)`, `f(1)`, `f(2)`,…`f(n-1)`:

```
for (int i=0; i<n; ++i)
    spawn f(i);
sync;
```

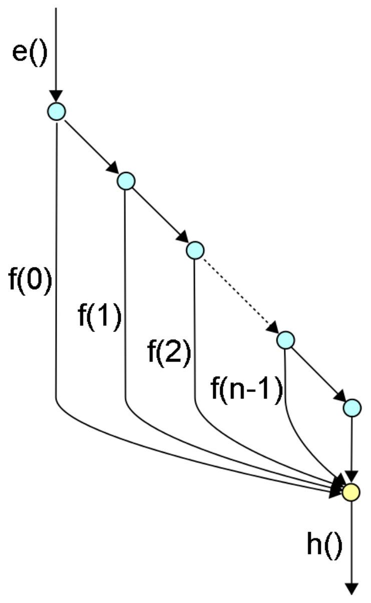


Figure 2: Loop that issues n tasks and diagram of its control flow.

Treat the example as a proxy for trickier loops, such as linked-list traversals, where the trip count cannot be computed in advance of executing the iterations.

The asymptotic space difference between child-stealing and continuation stealing is:

- With child stealing, the loop spawns *all* `n` tasks before executing any of them, if no thieves are available. From the perspective of Figure 2, execution walks across the blue circles, and then executes the tasks. The code requires space proportional to `n` to hold the unexecuted tasks.
- With continuation stealing, the original thread executes `f(i)` before advancing the loop. If a thief becomes available, it advances the loop by one iteration, spawns a task, and leaves the loop continuation ready to be stolen and advanced by the next thief. Thus the number of extant tasks is no more than *P*, the actual number of threads executing the loop in parallel.

The advantage of continuation stealing is that the space consumption blows up by no more than a factor of *P*, whereas with child stealing the blowup is unbounded. Of course throttling heuristics have been suggested, and implemented in some systems, but these can err on the side of not exposing enough parallelism and consequently hindering speedup.

The blowup is not always easy to see. The author wrote a semi-recursive parallel quicksort, and only years later discovered that it could blow up with child stealing. When sorting N items, the blowup could cause it to consume O(N*)* temporary space, despite recursing only O(lg *N*) levels deep. The space-efficient solution required coding contortions that imitated continuation-stealing in a child-stealing environment.

The previously mentioned irksome semantic difference between child-stealing and continuation-stealing is most evident when no thief is available. With continuation stealing, the order of execution becomes identical to as if the spawn and sync annotations were ignored. With child stealing, the loop executes to completion, except no `f(i)` executes, and then each `f(i)` executes, possibly in reverse order, depending upon how tasks are buffered. The net effect is that execution order for a single thread becomes very different than if the parallel annotations were ignored.

Continuation stealing also has an advantage in that it makes "hyperobjects" [Hyper] semantically simple, and efficient to implement.

Despite its asymptotic inefficiency and semantic drawbacks, child-steal has a few merits:

- It is easier to implement as "just a library" without compiler support. This was the motivation for using it in TBB.

- It may gain a small constant factor improvement in situations where the cost of saving/restoring a continuation is high, such as on a machine with a very large register set. Saving a continuation requires saving the "caller-save" registers *and* the "callee-save" registers. In a child-steal system, only the "caller-save" registers have to be recorded in a spawned task.

- It may be more intuitive to novices with respect to handling of classic thread-local storage, though there is a strong case for avoiding classic thread-local storage, and using other abstractions, in highly parallel fork-join programs.

Nonetheless, throwing away an asymptotic bound should be a point of concern.

Second Choice: What Thread Executes After a Join?

The other design choice is deciding which thread continues execution after a join point. The choices are:

- **Stalling:** The same thread that initiated the fork. In Figure 1, the thread that executed `e()` would execute `h()`.

- **Greedy:** The *last* thread to reach the join point would continue execution. In Figure 1, the thread that executes `h()` would be either the thread that executed `f()` or the thread that executed `g()`.

The first choice is called "stalling" because it sometimes requires that a thread stall and wait for other threads to finish their work before continuing. Some systems, such as TBB and PPL, try to find other work that the thread can do in the meantime. However, finding such work that does not entail risk of causing deadlock is not always possible. In contrast, in a greedy scheduler, no thread idles if there is work to do.

Greedy scheduling is critical to an asymptotic bound, known as "Brent's Lemma", which is widely used in analyzing parallel algorithms. Let T_P denote the execution time with P hardware workers on an ideal machine (Parallel Random Access Machine). The bound is:

$$T_P \leq T_1/P + T_\infty$$

The formula says that with greedy scheduling, the execution time is no worse than the total work (T_1) parallelized across the processors, plus the time to execute the critical path (T_∞). Stalling schedulers prevent rigorous application of this theory.

Despite this drawback, stalling schedulers are sometimes preferred because of issues related to thread identity. Greedy scheduling can cause a function to return on a different thread that it was called on, causing havoc in codes that use thread-local storage and possibly breaking *some* mutex implementations. As with the choice of child-steal vs. continuation steal, it's a case of common practice conflicting with the asymptotic ideal.

Intel has experimented with a compromise choice that provides physically greedy scheduling while yielding logical thread mappings like a stalling scheduler. This compromise uses more than *P* software threads, but only *P* are awake at any instant. When a software thread needs to stall, it sleeps and wakes up another thread that becomes a thief. In effect, the hardware thread is not stalling, but merely changing its name. The scheduler *looks* like a stalling scheduler to software, but is a greedy scheduler from the hardware viewpoint. The combination appears to deliver acceptable performance.

2x2 Taxonomy

The two decisions presented can be made independently. As the following table shows, different designers have made different choices.

	Stalling	**Greedy**
Child Stealing	TBB PPL OpenMP tied tasks	unpublished scheduler OpenMP untied tasks
Continuation Stealing	experimental Cilk run-time [logical stalling] OpenMP untied tasks	Cilk OpenMP untied tasks

Note how OpenMP occupies all four table combinations. It pushes some of the decision onto the programmer on a task-by-task basis ("tied" versus "untied" tasks), which lets programmers decide whether thread identities or asymptotic issues are most important. However, it makes no guarantee of continuation-stealing or gready scheduling. An OpenMP implementation is allowed to treat untied tasks as tied, and thus incur asymptotic penalties. Thus an OpenMP programmer cannot rely upon the asymptotic guarantees outlined in this paper.

Summary

There are two orthogonal decisions for this scheduling, one pertaining to forks, and the other pertaining to joins. The intuitively obvious way to schedule fork-join task graphs onto threads incurs asymptotic penalties.

References

[CilkM] Lee, Boyd-Wickizer, Huang, and Leiserson. "Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems", PACT' 10.

[Hyper] Frigo, Halpern, Leiserson, and Lewin-Berlin. "Reducers and other Cilk++ hyperobjects", SPAA '09.