15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Scheduling **Date:** 2007 November 1st
Scribe: Matthew Danish

20.1 Scheduling fine-grained threads

For a Directed Acyclic Graph of size W and depth D we know that any greedy schedule will run in $\Theta(W/P + D)$ where P is the number of processors. Assuming a collection of ready threads, Q, scheduling algorithms tend to differ primarily on the ordering of work performed in Q. The operation of forking is simply a matter of pushing a new task onto Q.

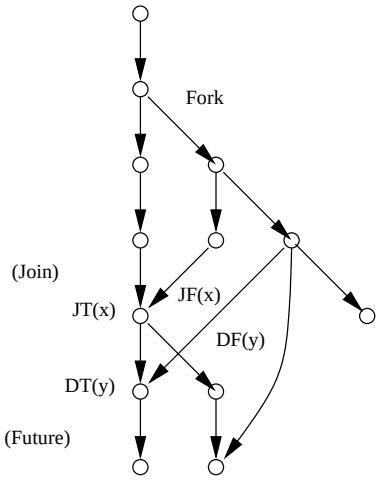

Figure 20.1.1: DAG example

Figure 20.1.1 demonstrates a possible DAG with a fork/join and also a simple usage of futures (or synchronous variables). The data dependencies are intended to illustrate that while fork/join implies only a single data dependency link back, futures allow any number of reads.

- $JT(x)$ is a join edge in the original thread.
- $JF(x)$ is a join edge in the forked thread.
- $DT(y)$ is a future read edge.
- $DF(y)$ is a future finished edge.

The meaning of join will be considered first, in this pseudo-code:

```
JT(x) = if x.state = 0
          then continue
```

```
          else set x.continuation to current continuation
               decrement x.state
               yield

JF(x) = if x.state = 0
          then push continuation of original thread onto Q
               yield
          else decrement x.state
```

`x.state` is initialized to the number of forked threads. As each forked thread completes, it decrements `x.state` until 0 has been reached. At this point, the join is complete and it is now okay to proceed with the computation in the original thread. Therefore, the meta-variable x needs to contain this information which is shared across all threads.

$$x = \langle state, continuation \rangle$$

In addition, it is important that x be updated atomically.

Futures require some more work because there are multiple possible continuations after the parallel computation has finished. Therefore a different meta-variable, y, is used:

$$y = \langle state, value, continuation\ list \rangle$$

where *state* is either $Unfinished$ or $Finished$.

```
DT(y) = if state = Finished
          then continue
          else push current continuation onto x.continuations
               yield

DF(y) = set state to Finished
        for each T in x.continuations
          push T onto Q
        continue
```

Threads waiting for a future to finish will invoke $DT(y)$ in order to be placed on the waiting list. When the future finishes, it invokes $DF(y)$ to queue up the waiting tasks. Some form of atomicity should be used to prevent contention over y. While futures are simple in pseudo code, the details can be quite complex which is why they are not widely available.

20.1.1 Real APIs

20.1.1.1 pthreads

- fork: `pthread_create`
- join: `pthread_join`

20.1.1.2 OpenMP

- fork: parallel sections
- join: barriers

20.1.1.3 Cilk

- fork: `spawn`
- join: `sync`

20.1.1.4 X10

- fork: `async`
- join: `finish`

20.1.1.5 Concurrent ML

- fork: `spawn`
- join: `joinEvt` with `sync`
- synchronous variables: `iVar`, `iPut`, `iGet` (write-once) `mVar`, `mPut`, `mGet` (mutable)

20.1.1.6 Parallel Haskell

- fork: `par`, `parMap`, `forkIO` and other combinators
- futures: implicit
- synchronous variables: `newTMVar`, `putTMVar`, `takeTMVar` (transactional memory library)

20.2 Schedulers

There are a number of strategies for dealing with Q which can produce different performance characteristics.

Breadth-First Use Q as a FIFO queue thus in effect traversing the DAG in breadth-first fashion. This is very simple, so it is often used, but has some serious deficiencies. It is possible for the Q to become very large. Consider Matrix Multiplication: at some point the DAG will be $\mathcal{O}\left(n^3\right)$ in breadth and therefore the queue can grow cubically![1]

[1]This is why most pthreads implementations cannot be used for fine-grained threading.

Parallel Depth-First Assign a priority to nodes in the DAG by preordering them. Then when considering between choices of nodes to run next, choose that with best priority. If you only have fork/join, this produces the same effect as treating Q as a stack. This is better than Breadth-First because $|Q| \in \mathcal{O}(p \log n)$.

Work Stealing With this algorithm, Q is broken down into per-processor queues which are double-ended. From one end, a push or a pop can be performed by the local processor. From the other end, only a pop can be performed, and that is called a *steal*. The basic idea behind the algorithm is that each processor will check its local queue for work, and if there is nothing, then it will randomly steal some work from another processor.