Fork Join

Pieter van den Hombergh

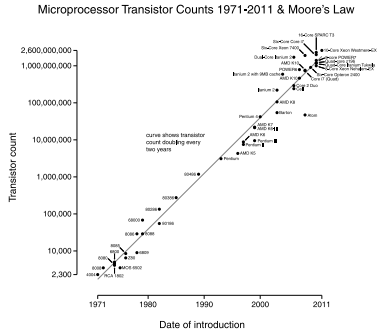
Fontys Hogeschool voor Techniek en Logistiek

April 23, 2018

Trends in hardware

- Moore law (still) works for transistor counts $\rightsquigarrow$ cores, but does not help anymore for clock speeds.
- The brake on clock speed is power consumption of the chip. Not as a matter of environmental care but because of potential chip-death.
- Having more cores, but having them idle for most of the time is a waste of resources.
- Thread per **request response execution model** as in most server applications does under-utilise the CPU. In most cases the CPU-core waits for IO (socket, disk) and cloud do other things if available.
- Finer task granularity will help here.

Moore's Law



Note that the vertical scale is logarithmic. From wikipedia, Oct 2011.

Divide and conquer

- Previous chapters we learned about **Executors** and **Queues**.
- We learned that the relatively high costs of creating and destroying threads can be mitigated by using thread **pools**.
- The Fork-Join framework employs them to good use to provide a *light weight* thread framework that is optimised for a divide an conquer approach to compute intensive (CPU-bound) tasks.

Fork join algorithm outline

Any problem that can be solved after this code template is ideal to apply the FJ-framework.

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

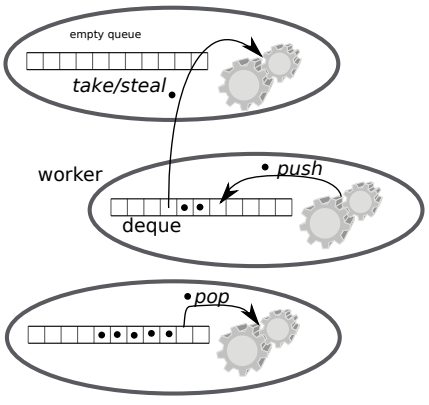
The invoke in parallel part is done by an executor which employs a thread pool.

Stealing work as a strategy

The positive dimension in work is **down** anyway. . . .

To keep worker threads working, it is Okay to let them steal each other's work, as long as they do this unobtrusively.

Work stealing approach in Fork Join



Each worker has its own queue and pushes and pops only to/from it's own queue. It may steal work form *an other's* queue by taking tasks **from the other end** of the deque.

after [?]

Stealing where it really ~~hurts~~ helps.

Any thieve will try to steal as much in one swoop as possible. This behaviour is turned to good advantage, because the big chunks of work land at the bottom of the deque/stack, the spot where the thieve comes in.
Each worker splits up the tasks in halves and pushes these halved tasks onto the top (right) of the deque. So the smaller tasks end up at the top of the stack at the workers side.
The thieves, however, steal at the bottom (left), where the large tasks live. This way, after stealing, the same thieve will not return quickly, for being "satisfied" with his prize for some time.

work at bottom