Work-Stealing without the Baggage

V. Kumar, D. Frampton, S. Blackburn, D. Grove, O. Tardieu

Presentation by Roman Schmocker
Work stealing: Idea

- Work queue for each thread
  - No dependencies allowed between work jobs

- Idle threads steal jobs from busy threads

<table>
<thead>
<tr>
<th>sqrt(42)</th>
<th>Thread a</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(3.14)</td>
<td>Thread b</td>
</tr>
<tr>
<td>fib(21)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sin(3.14)</th>
<th>Thread a</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(21)</td>
<td>Thread b</td>
</tr>
<tr>
<td>sqrt(42)</td>
<td></td>
</tr>
</tbody>
</table>
X10

- Research language by IBM
  - Object-oriented, Java-like
  - Translated to Java
  - Strong focus on parallel programming

- Async-Finish construct in X10:
  ```java
  finish {
    async a = sqrt (42); // May run concurrently
    b = fib (21);
  }
  // Thread join point
  c = a+b;
  ```
Async-Finish and Work Stealing

- When encountering async
  - push the *continuation* to work queue
  - execute the *async* immediately

- Continuation
  - The code following an async statement up to the end of finish block

```plaintext
finish {  
    async a = sqrt (42);  
    b = fib (21);  
}
```
Translation to Java

● Put continuation into separate method
  ○ called *continuation method*

● All variables accessed within continuation method are heap-allocated
  ○ i.e. generate frame classes and instantiate *frame objects* that hold these variables

● Work queue entries:
  ○ continuation method
  ○ frame object (as its argument)
Finish block semantics

- Worker retrieves job from queue
  - No steal -> No waiting necessary

- Worker retrieves **null**
  - continuation stolen!
  - How to check if thief has finished execution?

- Atomic integer for each finish block
  - Denotes number of active threads
  - Increment when stealing
  - Decrement when completing job
  - Worker: proceed when zero
Performance analysis

- Authors interested in *sequential overhead*
  - Compare performance between:
    - Work-stealing with only one thread
    - Sequential version (no async-finish statements)

- Several benchmarks
  - e.g. Fibonacci number, LU-Decomposition

- Results
  - Sequential overhead is huge!
  - up to 16x slower than sequential version
  - Best result has still overhead of 1.5
Performance analysis

- Some operations very costly
  - Synchronization of work queue
  - Allocation (and deallocation) of frame objects

- Method splitting prevents optimizations
  - plus overhead of additional call

- This applies even when there's no steal!

- Moreover, further analysis has shown that steals are very rare
  - usually 1 steal among 1'000'000 tasks
  - at most 1 in 10
X10 (Try-Catch)

- **New way to translate async-finish**
  - main contribution of the paper

- **Goal: No sequential overhead**
  - Preparing for a potential steal is too expensive
  - Use call stack as implicit queue
  - Copy values only during an actual steal operation

- **Control Flow modelled with Java exceptions**
  - First step: wrap async-finish into try-catch
X10

```x10
def fib (n:Int):Int {
    val a:Int; val b:Int;
    if (n < 2) return n;

    finish {
        async a = fib(n-1);
        b = fib (n-2);
    }

    return a + b;
}
```

Java

```java
int fib (int n) {
    int a,b;
    if (n < 2) return n;
    try {
        try {
            a = fib (n-1);
        }
        catch (...) {} 
        b = fib (n-2);
    }
    catch(...) {} 

    return a + b;
}
```
Informing thieves

```c
int fib (int n) {
    int a, b; if (n < 2) return n;
    try {
        try {
            // Atomically set a flag indicating that work can be stolen
            // WS is a class with some static support methods
            WS.setFlag();
            a = fib (n-1);
        } catch (...) {}
        b = fib (n-2);
    } catch (...) {} return a + b;
}
```

Performing a steal

- Thief forcibly stops worker
  - using Java VM functionality

- Copy call stack of worker
  - and update flag

- Restart worker thread

- Dive into execution by throwing Continuation exception
  - Requires catch clause for thief!
int fib (int n) {
    int a,b; if (n < 2) return n;
    try {
        try {
            try {
                WS.setFlag();
                a = fib (n-1);
            }
            catch (Continuation c) {
                // Empty catch clause.
                // Thief will start here after throwing Continuation exception!
            }
            b = fib (n-2);
        }
        catch(...) {} 
        return a + b;
    }
}
Control flow

- **Goals**
  - Worker must not execute stolen continuation
  - None shall leave finish{} while the other is running
  - Exactly one must proceed after finish{}

- **Guard exit**
  - join() method at end of try blocks
  - Correct control flow through exceptions

- **Finish node**
  - Available in join()
  - Created lazily during steal
  - Atomic integer, represents active threads (initially 2)
Join method

// in class WS

static void join () {
    if (WS.getFlag() == false) {
        // A steal has occurred
        int active = finish_node.count.decrementAndGet();
        if (active == 0)
            // I'm the last thread
            throw new Finish();
    } else {
        // The other thread hasn't finished
        throw new JoinFirst();
    }
}
```c
int fib (int n) {
    int a,b; if (n < 2) return n;
    try {
        try {
            WS.setFlag();
            a = fib (n-1);
            WS.join();
        } catch (JoinFirst j) {
            WS.exit(); // Search for other work!
        } catch (Continuation c) { // Thief entry point
        }
        b = fib (n-2);
        WS.join();
    } catch (JoinFirst j) {
        WS.exit(); // Search for other work
    } catch (Finish f) {
    }
    return a + b;
}
```
```c
int fib (int n) {
    int a,b; if (n < 2) return n;
    try {
        try {
            WS.setFlag();
            a = fib (n-1);
            WS.join();
        } catch (JoinFirst j) {
            WS.exit(); // Search for other work!
        } catch (Continuation c) { // Thief entry point
        }
        b = fib (n-2);
        WS.join();
        } catch (JoinFirst j) {
            WS.exit(); // Search for other work
        }
    } catch (Finish f) {
    }
    return a + b;
}
```

Example:
No steal
int fib (int n) {
int a, b; if (n < 2) return n;
try {
    try {
        WS.setFlag();
        a = fib (n-1);
        WS.join();
    } catch (JoinFirst j) {
        WS.exit(); // Search for other work!
    }
    catch (Continuation c) { // Thief entry point
        b = fib (n-2);
        WS.join();
    } catch (JoinFirst j) {
        WS.exit(); // Search for other work
    }
    catch (Finish f) {
    }
    return a + b;
}
int fib (int n) {
    int a,b; if (n < 2) return n;
    try {
        try {
            WS.setFlag();
            a = fib (n-1);
            WS.join();
        }
        catch (JoinFirst j) {
        }
        catch (JoinFirst j) {
            WS.exit(); // Search for other work!
            WS.exit(); // Search for other work!
        }
        catch (Continuation c) { // Thief entry point
            b = fib (n-2);
            WS.join();
        }
    }
    catch (JoinFirst j) {
        WS.exit(); // Search for other work
    }
    catch (Finish f) {
    }
    return a + b;
}
State management

● Computed variables in two different stacks!

● Move values to correct stack
  ○ depends on who finishes last

● First thread stores its values to the finish node
  ○ in JoinFirst catch blocks

● Last thread retrieves values from finish node
  ○ in Finish catch block
Improvements

● `WS.join()` only needed when steal occurs

● Generate two versions of method
  ○ Slow version: as seen previously
  ○ Fast version: `join()` replaced with NOP
  ○ Default to fast version
  ○ Stack frame layout and jump offsets remain the same!

● Thief switches worker to slow version when stealing
Performance Evaluation

● **Sequential overhead**
  ○ Usually a lot smaller than previous solution
  ○ Between 1.15 and 1.5
  ○ (one outlier has little more than 2)

● **Speedup**
  ○ compared to purely sequential version
  ○ very good for fine-grained concurrency (e.g. Fibonacci), up to 7x speedup for 12 threads
  ○ at least on par with old solution on other benchmarks
Conclusion

- Interesting approach
- Requires managed runtime (Java VM)
- Impressive results
- Possible improvements
  - Reduce worker downtime
  - Translate X10 arrays correctly