Work–Stealing by Stealing State from Live Stack Frames of a Running Application

Vivek Kumar  
Australian National University

Daniel Frampton  
Australian National University

David Grove  
IBM T.J. Watson Research Center

Olivier Tardieu  
IBM T.J. Watson Research Center

Stephen M. Blackburn  
Australian National University

Generously supported by IBM & the Australian Research Council
Motivation

- Multicore era
  - Dynamic task parallelism

- Load balancing
  - Work–sharing
    - Central task queue
    - Scalability bottleneck with increase in threads
  - Work–stealing
    - Fixed number of threads
    - One task queue per thread
Work–Stealing
Work–Stealing

Kumar et al.
Work–Stealing

B

A

Kumar et al.
Work–Stealing

Kumar et al.
Overheads

control Flow

Coordination

Enough context provided??

Kumar et al.
Overheads

X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time
Overheads

X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n); }
        b = B(n-1);
    }
    val result = a + b;
}

final static class _$main extendsMainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                this.result = this.a + this.b;
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}

final static class _$main extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {} 
    }
}

final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0A0A0(this);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {} 
    }
}
Overheads

X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Providing Contexts
Control Flow

Kumar et al.
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n);}
        b = B(n-1);
    }
    val result = a + b;
}

final static class _$main extends MainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                this.result = this.a + this.b;
        }
    }
    public def back(worker:worker,frame:Frame):void {}  
}

final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {}  
    }

final static class _$mainF0A0 extends RegularFrame {
    public def fast(worker:Worker):void {
        this._pc = 1;
        push(worker);
        val tmp:_$mainF0A0B0 = new _$mainF0A0B0(ff);
        tmp.fast(worker);
        _$main.b = B(_$main.n - 1);
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
                _$main.b = B(_$main.n - 1);
        }
    }
    public def back(worker:worker,frame:Frame):void {}  
}

final static class _$mainF0B0A0 extends AsyncFrame {
    public def fast(worker:Worker):void {
        _$main.a = A(_$main.n);
        poll(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {}  
    }

Steal Point
Overheads

X10 (2.0) Fib (40) – Single Thread
Work-Stealing Execution Time Normalized with Sequential Execution Time

Kumar et al.
public static def main() {
    val n = 20;
    val a:int;
    val b:int;
    finish {
        async{ a = A(n);}
        b = B(n-1);
    }
    val result = a + b;
}

final static class _$main extends MainFrame {
    public def fast(worker:Worker):void {
        this.n = 20;
        this._pc = 1;
        val tmp:_$mainF0 = new _$mainF0(this);
        tmp.fast(worker);
        this.result = this.a + this.b
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
            this.result = this.a + this.b;
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}

final static class _$mainF0 extends FinishFrame {
    public def fast(worker:Worker):void {
        val tmp = new _$mainF0A0(ff,ff);
        tmp.fast(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {}  
    }
}

final static class _$mainF0A0 extends RegularFrame {
    public def fast(worker:Worker):void {
        this._pc = 1;
        push(worker);
        _$main.a = A(_$main.n);
        poll(worker);
    }
    public def resume(worker:Worker):void {
        switch (this._pc) {
            case 1:
            _$main.b = B(_$main.n - 1);
        }
    }
    public def back(workerWorker,frame:Frame):void {}
}

final static class _$mainF0B0A0 extends AsyncFrame {
    public def fast(worker:Worker):void {
        _$main.a = A(_$main.n);
        poll(worker);
    }
    public def resume(worker:Worker):void {
        public def back(worker:Worker,frame:Frame):void {}
    }
}
Kumar et al.

Steal Ratio

Ratio of Total Continuations Stolen to Total Continuations Produced in X10 (2.0) Fibonacci Benchmark
Steal Ratio

Ratio of Total Continuations Stolen to Total Continuations Produced in X10 (2.0) Fibonacci Benchmark
VM Supported Work–Stealing

• Our philosophy:
  – Small steal ratio
    ➢ Saving contexts for every continuation inefficient
  – Provide contexts only when steal occurs

• Our approach:
  – Thief steals victim’s Java stack frame
  – Thief forces the victim to yield to start the steal
VM Assistance

- Stealing the Stack Frames

### Java Stack Growth

- **Thread-0**
  - run()
  - Fast_A()
  - Fast_B()
  - Fast_C()
  - Fast_D()
  - Fast_E()
  - Fast_G()
  - Fast_H()
  - Fast_I()

### Deque

- FP_D
- FP_G
- FP_I

### Thread-1

- run()
### VM Assistance

- **Stealing the Stack Frames**

<table>
<thead>
<tr>
<th>Thread-0</th>
<th>Deque</th>
<th>Thread-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>run()</td>
<td>FP_D</td>
<td>run()</td>
</tr>
<tr>
<td>Fast_A()</td>
<td>FP_G</td>
<td></td>
</tr>
<tr>
<td>Fast_B()</td>
<td>FP_I</td>
<td></td>
</tr>
<tr>
<td>Fast_C()</td>
<td></td>
<td>Steal_FP()</td>
</tr>
<tr>
<td>Fast_D()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast_E()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast_G()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast_H()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast_I()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Java Stack Growth**

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0
- run()
- Fast_A()
- Fast_B()
- Fast_C()
- Fast_D()
- Fast_E()
- Fast_G()
- Fast_H()
- Fast_I()

Deque
- FP_G
- FP_I

Thread-1
- run()
- Initiate()
- Steal()
VM Assistance

- Stealing the Stack Frames
VM Assistance

- Stealing the Stack Frames

<table>
<thead>
<tr>
<th>Thread-0</th>
<th>Deque</th>
<th>Thread-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>run()</td>
<td>FP_G</td>
<td>run()</td>
</tr>
<tr>
<td>Fast_A()</td>
<td>FP_I</td>
<td>Initiate()</td>
</tr>
<tr>
<td>Fast_B()</td>
<td></td>
<td>Fast_A</td>
</tr>
<tr>
<td>Fast_C()</td>
<td></td>
<td>Fast_B</td>
</tr>
<tr>
<td>Fast_D()</td>
<td></td>
<td>Fast_C</td>
</tr>
<tr>
<td>Fast_E()</td>
<td></td>
<td>Fast_D</td>
</tr>
<tr>
<td>Fast_G()</td>
<td></td>
<td>Steal()</td>
</tr>
<tr>
<td>Fast_H()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast_I()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Java Stack Growth

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0

run()
Fast_A()
Fast_B()
Fast_C()
Fast_D()
Fast_E()
Fast_G()
Fast_H()
Fast_I()

Deque

FP_G
FP_I

Thread-1

run()
Initiate()

States_D ➔ States_C ➔ States_B ➔ States_A

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0

run()
Fast_A()
Fast_B()
Fast_C()
Fast_D()
Fast_E()
Fast_G()
Fast_H()
Fast_I()

Deque

FP_G
FP_I

States_D → States_C → States_B → States_A

Thread-1

run()
Fast_D
Fast_J
Fast_K
Fast_L
Fast_M

Java Stack Growth
VM Assistance

- Stealing the Stack Frames

Thread-0

run()

Fast_A()

Fast_B()

Fast_C()

Fast_D()

Fast_E()

Fast_G()

Fast_H()

Fast_I()

Deque

FP_G

FP_I

Thread-1

run()

Fast_D

States_D → States_C → States_B → States_A

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0

run()
Fast_A()
Fast_B()
Fast_C()
Fast_D()

Thread-1

run()
Fast_D

States_D → States_C → States_B → States_A
VM Assistance

- Stealing the Stack Frames

Thread-0

run()

Thread-1

run()

Fast_D

States_D  ➔  States_C  ➔  States_B  ➔  States_A

Java Stack Growth

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0

Java Stack Growth

run()

Thread-1

run()

- Fast_D
- Fast_J
- Fast_K
- Fast_L
- Fast_M

States_D → States_C → States_B → States_A
VM Assistance

• Stealing the Stack Frames

Thread-0

run()

States_D → States_C → States_B → States_A

Thread-1

run()

Fast_D

Java Stack Growth

Kumar et al.

8
VM Assistance

- Stealing the Stack Frames

Thread-0

run()

Thread-1

run()

Slow_D

Java Stack Growth

States_C ➔ States_B ➔ States_A

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0

run()

Thread-1

run()

Slow_D

Slow_C

States_A

States_B

Java Stack Growth
VM Assistance

- Stealing the Stack Frames

Thread-0

- run()

Thread-1

- run()
- Slow_D
- Slow_C
- Slow_B

States_A

Java Stack Growth

Kumar et al.
VM Assistance

- Stealing the Stack Frames

Thread-0
- run()

Thread-1
- run()
- Slow_D
- Slow_C
- Slow_B
- Slow_A

Calculation Finished

Kumar et al.
Experimental Results – Execution Time

Execution Time Normalized With Single Thread Default Work-Stealing Time

a) Integrate (1000)

b) Fib (40)
Experimental Results – Speedup

<table>
<thead>
<tr>
<th>Total Threads</th>
<th>Speedup Relative to Single Thread Default Work Stealing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default WS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

- a) Integrate (1000)
- b) Fib (40)
Summary

- Multicore era
  - Dynamic task parallelism.
- Load balancing
  - Work–stealing schedulers

Overheads = Control Flow + Providing Contexts + Coordination Effort
Future Work

- Multicore era
  - Dynamic task parallelism.
- Load balancing
  - Work-stealing schedulers

Overheads = Control Flow + Providing Contexts + Coordination Effort

- Test with high steal ratio benchmarks
- Research new VM extensions to make X10 run faster.
Questions .....?