Deadlock Avoidance in Parallel Programs with Futures

Why Parallel Tasks Should Not Wait for Strangers

Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco Vasconcellos, Max Grossman

OOPSLA, Vancouver, 2017
Futures

Fork-join model + Data
Widespread use of futures

1. Asynchrounous programming
   - Language support (async, await): Python, Javascript, Rust

2. Task parallel programming
   - Language support: Java, C#, C++, Kotlin
   - Library support: C++ (TBB, Kokkos, Charm++), Java (HJ-Lib, Quasar)
Uses of futures with shared memory

Task-DAG parallelism

- Data-flow parallelism
- Shared collections of futures (matrices)
Uses of futures with shared memory

Task-DAG parallelism

- Data-flow parallelism
- Shared collections of futures (matrices)

Problem: Cyclic data-dependencies cause deadlocks!
Off-by-one errors cause deadlocks
Shared memory and futures

Intuition:

- Root-cause of future-deadlocks are data races.

1. Is it true?
2. Why?
3. How can we use this property for verification?
Outline

1. Futures and its deadlocks

2. **Known Joins ⇒ DF**: Deadlock avoidance with futures & benchmarks

3. **DRF ⇒ Known Joins**: How DRF enjoys Deadlock-Freedom

4. Conclusion and future work
1. Futures and its deadlocks
Futures: Tasks that "return" values

async: (unit→T) → Future<T>

- **Control**: Forks a task A
- **Data**: Returns the future value of type Future<T>

get: Future<T> → T

- **Control**: Joins with task A
- **Data**: Returns the value of type T "produced" by task A
Deadlocked example

// Task P
1 shared Future<Integer> x, y;
2 x = async(() -> y.get()); // Task A
3 y = async(() -> x.get()); // Task B

1. P forks A writes to x
   ○ A waits for the task in y
2. P forks B writes to y
   ○ B waits for the task in x

Data-race causes 2 traces
Trace 1 (no deadlock)

1 shared Future<Integer> x, y;
2 x = async(() -> y.get()); // y = null
3 y = async(() -> x.get);

<table>
<thead>
<tr>
<th>Task P</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork A</td>
<td>read y null</td>
<td>read x A</td>
</tr>
<tr>
<td>write x A</td>
<td></td>
<td>get A</td>
</tr>
<tr>
<td>fork B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write y B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace 2 (deadlock)

```java
1. shared Future<Integer> x, y;
2. x = async(() -> y.get()); // y = B
3. y = async(() -> x.get);
```

<table>
<thead>
<tr>
<th>Task P</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork A</td>
<td>read y B</td>
<td>read x A</td>
</tr>
<tr>
<td>write x A</td>
<td>get B</td>
<td>get A</td>
</tr>
<tr>
<td>fork B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write y B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proving DRF $\Rightarrow$ DF

DRF $\Rightarrow$ $\$POLICY$ $\Rightarrow$ DF

Deadlock-freedom policy valid in all DRF programs
2. Known Joins ⇒ DF

Deadlock avoidance with futures & benchmarks
Known-Joins implementation overview

Program start (empty-known set)

async

1. Before: parent copies known-set to child
2. After: parent extends known-set with new task

get

1. Before: membership-test fail ⇒ POLICY ABORT
2. After: merge known-set of task
Running example knowledge

Knowledge: {}

```java
1  shared Future<Integer> x, y;
2  x = async(() -> y.get());
```

Knowledge: {A}

```java
3  y = async(() -> x.get());
```

Knowledge: {A, B}
Know-Joins in practice

Habanero-Java: A Java 8 parallel programming library

Extends the deadlock-free API subset with futures!

- isolated: mutual-exclusion
- phaser: barrier and producer-consumer
- finish: descendant task termination
- future (with the known-joins policy)
Evaluation

- 2,300 assignments checked (1 unknown join, deadlocked example)
- 5 benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of async</th>
<th># of get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>15,872</td>
<td>37,696</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>21,000</td>
<td>4,641</td>
</tr>
<tr>
<td>Crypt</td>
<td>16,384</td>
<td>16,384</td>
</tr>
<tr>
<td>Strassen</td>
<td>30,811</td>
<td>44,816</td>
</tr>
<tr>
<td>Series</td>
<td>999,999</td>
<td>999,999</td>
</tr>
</tbody>
</table>
## Evaluation: time overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Snapshot-sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>0.99×</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>0.96×</td>
</tr>
<tr>
<td>Crypt</td>
<td>1.04×</td>
</tr>
<tr>
<td>Strassen</td>
<td>1.07×</td>
</tr>
<tr>
<td>Series</td>
<td>1.06×</td>
</tr>
</tbody>
</table>
## Evaluation: memory overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Snapshot-sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>1.00×</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>1.00×</td>
</tr>
<tr>
<td>Crypt</td>
<td>1.08×</td>
</tr>
<tr>
<td>Strassen</td>
<td>1.28×</td>
</tr>
<tr>
<td>Series</td>
<td>2.34×</td>
</tr>
</tbody>
</table>
3. DRF ⇒ Known Joins
Computation Graphs

- Nodes: instruction instances
- Edges: happens-before dependencies (async, get, and sequential)
- Node annotations: known tasks and local memory
Knowledge flows with reachability

v4 knows g
v4 happens-before w2
-----------------------
w2 knows g
Knowledge must contain tasks in memory

In DRF graphs
Main results

Mechanized and proved in Coq

1. Known-Joins $\Rightarrow$ Deadlock Freedom
2. Data-Race Freedom $\Rightarrow$ Known-Joins
3. Known-Joins interpretation as a causality query
Conclusion

- Introduced a theory of futures and shared memory (CG)
- Showed that data-races are the root cause of deadlocks
- Talked about a deadlock avoidance tool (1.06× time-overhead for 1 million tasks)

Future work

- Promises lack runtime-information to derive deadlock detection
- Extend the theoretical framework for nondeterminism
Deadlock Avoidance in Parallel Programs with Futures

Why Parallel Tasks Should Not Wait for Strangers

Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco Vasconcelos, Max Grossman

OOPSLA, Vancouver, 2017