Analyzing Parallel Real-Time Tasks Implemented with Thread Pools

Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo

ReTiS Lab, Scuola Superiore Sant’Anna, Pisa, Italy
Motivations

How to model the workload due to a Deep Neural Network?
How inference engines schedule Deep Neural Networks?

Case study

- InceptionV3: powerful image recognition DNN
- Tensorflow: open-source machine learning framework by Google
- Tensorflow with Eigen math library on CPUs
- Strongly parallel workload

Nodes typically perform mathematical computations (e.g., tensor convolutions) whose implementation is platform-dependent and extremely parallel

DNN can be modeled as a direct acyclic graph (DAG)
How Tensorflow works on CPUs?

TensorFlow (Eigen) assigns ready nodes to threads of a thread pool.
TensorFlow (Eigen) assigns ready nodes to threads of a thread pool.

What if one of these functions blocks on a condition variable?
TensorFlow (Eigen) assigns ready nodes to threads of a thread pool.

Nodes are C++ Functions: the OS is not directly aware of them!

What if one of these functions blocks on a condition variable?
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

A **sequential flow** of execution that forks in **multiple parallel branches** and joins again in a sequential flow.
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v1v5 () {
    <execute v1>
    <fork v2,v3,v4>
    <wait for v2,v3,v4>
    <execute v5>
}

void v_i ( ) { (i=2,3,4)
    <execute v_i>
    <signal>
}
```
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```
void v1v5 ( ) {
  <execute v1>
  <fork v2,v3,v4>
  <wait for v2,v3,v4>
  <execute v5>
}
```

```
void v_i ( ) { (i=2,3,4)
  <execute v_i>
  <signal>
  <signal>
}
```

Signaling the condition variable

```
void v1v5 ( ) {
  <execute v1>
  <fork v2,v3,v4>
  <wait for v2,v3,v4>
  <execute v5>
}
```

Blocking on a condition variable
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

void v1v5 ( ) {
    <execute v1>
    <fork v2, v3, v4>
    <wait for v2, v3, v4>
    <execute v5>
}

void v_i ( ) { (i=2,3,4)
    <execute v_i>
    <signal>
}
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v1v5 () {
    <execute v1>
    <fork v2, v3, v4>
    <wait for v2, v3, v4>
    <execute v5>
}

void v_i () { (i=2,3,4)
    <execute v_i>
    <signal>
}
```

How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v1v5 ( ) {
    <execute v1>
    <fork v2,v3,v4>
    <wait for v2,v3,v4>
    <execute v5>
}
```

```c
void v_i ( ) { (i=2,3,4)  
    <execute v_i>  
    <signal> 
}
```
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

void v_i ( ) { (i=2,3,4)  
  <execute v_i>  
  <signal>  
}

void v1v5 ( ) {  
  <execute v1>  
  <fork v2,v3,v4>  
  <wait for v2,v3,v4>  
  <execute v5>  
}
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v1v5() {
    <execute v1>
    <fork v2,v3,v4>
    <wait for v2,v3,v4>
    <execute v5>
}
```

```c
void v_i() {
    (i=2,3,4)
    <execute v_i>
    <signal>
}
```
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v1v5() {
    <execute v1>
    <fork v2,v3,v4>
    <wait for v2,v3,v4>
    <execute v5>
}
```

```c
void v_i() { (i=2,3,4)
    <execute v_i>
    <signal>
}
```
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

```c
void v_1v5 () {
    <execute v1>
    <fork v2, v3, v4>
    <wait for v2, v3, v4>
    <execute v5>
}

void v_i () { (i=2, 3, 4)
    <execute v_i>
    <signal>
}
```
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

Thread 1

Thread suspended

Thread 2

Available concurrency

Reduction of the concurrency available to execute functions!

Current analysis techniques not considering this effect would produce unsafe results.
How Tensorflow works on CPUs?

Blocking implementation of fork-join parallelism:

Available concurrency

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target of this paper:
how to analyze parallel real-time tasks implemented with thread pools and blocking on condition variables?
Deadlocks can also occur

Assume two instances are released concurrently*

*Deadlocks are prevented in Tensorflow by serializing the execution of nodes blocking on condition variables
We have shown that thread pools and blocking synchronization may reduce performance.

Can we then conclude that this implementation paradigm should be avoided in real-time systems?

NO:

• Unfortunately, these paradigms are commonly used in real implementations.
• Not only Deep Neural Networks and Tensorflow, thread pools are commonly adopted also for cloud computing and web-services.

State-of-the-art analysis techniques do not consider this implementation paradigm and hence could lead to unsafe results!
Nodes are assigned to types

Limited-concurrency model
Nodes are assigned to types

- **Limited-concurrency model**

- **Blocking-join**
- **Blocking-child**
- **Blocking-fork**
- **Non-blocking**
Nodes are assigned to types

blocking-fork

blocking-child

blocking-join

Limited-concurrency model

Code executed before blocking on the condition variable

Code executed after being awakened

Nodes are assigned to types.
Nodes are assigned to types

Recall

```c
void v1v5 ( ) {
    <execute v1>
    <fork v2,v3,v4>
    <wait for v2,v3,v4>
    <execute v5>
}
```
Schedulability Analysis: intuition

Global Scheduling

Partitioned Scheduling

An approximate response-time bound is computed by leveraging the concept of available concurrency.

Partitioning algorithm allowing to safely re-use state-of-the-art analysis techniques by isolating concurrent BF nodes.

For additional details, please look at the paper.
Avoiding deadlocks: intuition

Reason in terms of available concurrency

Condition: Available concurrency > 0

Necessary condition for both global and partitioned scheduling

Sufficient for global scheduling

void v1v5 ( ) {
  <execute v1>
  <fork v2,v3,v4>
  <wait for v2,v3,v4>
  <execute v5>
}

void v_i ( ) { (i=2,3,4)
  <execute v_i>
  <signal>
}
**Goal:** how much is the *optimism* incurred by analyzing parallel tasks with limited concurrency with state-of-the-art techniques?

- Based on synthetic task sets

  \[ l_{\text{min}} = 3, U = 2, 8 \text{ threads, 6 tasks} \]

  - **Partitioned Scheduling**

  - **Global Scheduling**

  \[ l_{\text{min}} = 3, U = 3, 8 \text{ threads, 8 tasks} \]

- Each task has a lower bound to the available concurrency in \([l_{\text{min}}, l_{\text{max}}]\)

  Graceful degradation in the case of partitioned scheduling

  Pessimism due to the usage of lower-bound to the available concurrency
Conclusions

**Task model** for analyzing parallel tasks implemented with thread pools

**Conditions** for guaranteeing the absence of deadlocks

**Schedulability analysis**

**Experimental results**

to assess the optimism incurred by state-of-the-art analyses when parallel tasks are implemented with thread pools

**Future work:**
- New analysis approaches for parallel tasks with thread pools
- Design of partitioning algorithms aimed at optimizing schedulability
Thank you!

Daniel Casini
daniel.casini@sssup.it