Fast Parallel Programming using Modern C++

Tsung-Wei Huang
Research Assistant Professor, CSL & ECE

“The cleanest tasking API ever,”
user remark
This is a 15-minute Lightning Talk

- Identify your need of parallel programming
- Parallelize your workload using the right tools
- Boost your performance in writing parallel code
Identify Your Need of Parallel Programming

- Why should I care?
  - Your computer is forced to design with multiple cores
  - Want performance
  - Want throughput

- We are in many-core era

*Parallel programming is needed more than ever!*

Intel Sandy Bridge quad-core processor
Parallel Programming is VERY Difficult due to **Task Dependency**

- Painful debugging
- Max concurrency
- Race condition
- Lock guard
- Thread contention
Parallelize Your Workload using the Right Tools

Sequential version

```c++
int CookGarnish();
int CookEntree();
pair<int, int> Plate(int, int);
void Serve(int);

int garnish, entree;
pair<int, int> plates;

garnish = CookGarnish();
entree = CookEntree();
pates = Plate(garnish, entrees);
Serve(plates.first);
Serve(plates.second);
```

Parallel version (really?)

```c++
thread cook1 ([[&]] { garnish = CookGarnish(); });
thread cook2 ([[&]] { entree = CookEntree(); });
thread chief ([[&]] { plates = Plate(garnish, entree); });
thread waiter1([[&]] { Serve(plates.first); });
thread waiter2([[&]] { Serve(plates.second); });
```
A hard-coded yet “common” solution

- Limit max concurrency to two
- Use locks to add dependencies
- Thread contention
- Waste CPU resources
- Replace spin lock with mutex?
- Wait on conditional variable?
- How can I debug it?
- What if I have only one core?
- Rewrite the program?

Oh my gosh …
Boost Your Performance in Writing Parallel Code

[Cpp-Taskflow: A C++17 Header-only Parallel Programming Library]

```cpp
// create a taskflow object
Taskflow tf;

// create five tasks
auto [cook1, cook2, chief, waiter1, waiter2] = tf.silent_emplace(
    [&] () { garnish = CookGarnish(); },
    [&] () { entree = CookEntree(); },
    [&] () { plates = Plate(garnish, entree); },
    [&] () { Serve(plates.first); },
    [&] () { Serve(plates.second); });

// add dependencies
cook1.precede(chief);
cook2.precede(chief);
chief.precede(waiter1);
chief.precede(waiter2);

// execute
tf.wait_for_all();
```
A Slightly More Complicated Example

// source dependencies
S.precede(a0); // S runs before a0
S.precede(b0); // S runs before b0
S.precede(a1); // S runs before a1

// a_ -> others
a0.precede(a1); // a0 runs before a1
a0.precede(b2); // a0 runs before b2
a1.precede(a2); // a1 runs before a2
a1.precede(b3); // a1 runs before b3
a2.precede(a3); // a2 runs before a3

// b_ -> others
b0.precede(b1); // b0 runs before b1
b1.precede(b2); // b1 runs before b2
b2.precede(b3); // b2 runs before b3
b2.precede(a3); // b2 runs before a3

// target dependencies
a3.precede(T); // a3 runs before T
b1.precede(T); // b1 runs before T
b3.precede(T); // b3 runs before T
Taskflow Application Programming Interface

- **parallel_for**
  
  ```cpp
  // apply callable to each container item in parallel
  auto v = {'A', 'B', 'C', 'D'};
  auto [S, T] = tf.parallel_for(
      v.begin(), // beg of range
      v.end(),   // end of range
      [] (int i) {
        cout << "parallel in " << i << '\n';
      });
  // add dependencies via S and T.
  
  - **transform_reduce**
  
  - **linearize**
    
    ```cpp
    tf.linearize(A, B, C, D)
    ```
  
  - **dump**
    
    ```cpp
    ...
    ```
Dynamic Tasking

Create a task dependency graph at runtime

```cpp
// create three regular tasks
auto A = tf.silent_emplace([]().{}).name("A");
auto C = tf.silent_emplace([]().{}).name("C");
auto D = tf.silent_emplace([]().{}).name("D");

// create a subflow graph (dynamic tasking)
auto B = tf.silent_emplace([](auto& subflow) {
    auto B1 = subflow.silent_emplace([]().{}).name("B1");
    auto B2 = subflow.silent_emplace([]().{}).name("B2");
    auto B3 = subflow.silent_emplace([]().{}).name("B3");
    B1.precede(B3);
    B2.precede(B3);
}).name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C

// execute the graph without cleaning up topologies
tf.dispatch().get();
cout << tf.dump_topologies();
```
Real Application – VLSI Timing Analysis

- OpenTimer 1.0 (OpenMP)
- OpenTimer 2.0 (Cpp-Taskflow)
  - 10-30% faster than OpenMP
- Circuit graphs with 10-100M gates

**Runtime**

- vga_lcd
- netcard
- leon

**Memory**

- vga_lcd
- netcard
- leon
Thank you!

Tsung-Wei Huang
thuang19@Illinois.edu
402 CSL MC 228

<table>
<thead>
<tr>
<th></th>
<th>Without cpp-taskflow</th>
<th>With cpp-taskflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.0%</td>
<td>32.2%</td>
</tr>
<tr>
<td>2</td>
<td>1.3%</td>
<td>5.9%</td>
</tr>
<tr>
<td>3</td>
<td>2.7%</td>
<td>17.2%</td>
</tr>
<tr>
<td>4</td>
<td>0.7%</td>
<td>6.7%</td>
</tr>
<tr>
<td>Mem</td>
<td>4.56G/8.00G</td>
<td>4.76G/8.00G</td>
</tr>
<tr>
<td>Swp</td>
<td>183M/1.00G</td>
<td>183M/1.00G</td>
</tr>
</tbody>
</table>

Cpp-Taskflow Github: https://github.com/cpp-taskflow/cpp-taskflow

Acknowledgment: Chun-Xun Lin, Guannan Guo, Martin Wong
Sponsor: DARPA, NSF