Lab 8: Taskflow Programming

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 11:59 PM on 11/07

Overview

This lab introduces **Taskflow**, a modern C++ library for parallel and heterogeneous task programming (taskflow.github.io). You will learn how to express and execute different types of task dependency graphs in Taskflow, including static graphs, dynamic asynchronous tasks, and conditional control flows.

Learning Objectives

- Understand Taskflow's programming model and core abstractions.
- Build and execute static task graphs.
- Construct dynamic dependency graphs using asynchronous tasks.
- Express iterative control-flow graphs using conditional tasks.

Euler Instruction

```
~$ ssh your_CAE_account@euler.engr.wisc.edu
~$ sbatch your_slurm_script.slurm
```

Do not run on the login node. Work locally, push to GitHub, and run on Euler using Slurm.

Submission Instruction

Specify your GitHub link here:
https://github.com/YourGitHubName/ECE455/LAB08

Problem 1: Static Task Graph

Task: Build a simple static task graph using Taskflow. You will define several independent and dependent tasks, connect them with explicit precedence relations, and execute the graph using a Taskflow executor.

Kernel

Filename: static_tasking.cpp

```cpp
#include <taskflow/taskflow.hpp>  // Taskflow header

int main() {
  tf::Executor executor;
  tf::Taskflow taskflow("Static Taskflow Demo");

  auto A = taskflow.emplace([](){ printf("Task A\n"); });
  auto B = taskflow.emplace([](){ printf("Task B\n"); });
  auto C = taskflow.emplace([](){ printf("Task C\n"); });
  auto D = taskflow.emplace([](){ printf("Task D\n"); });

  // Define dependencies: A precedes B and C; both B and C precede D.
  A.precede(B, C);
  B.precede(D);
  C.precede(D);

  executor.run(taskflow).wait();
}
```

Goal: Observe how Taskflow executes tasks respecting dependencies and possible parallel execution.

Slurm Script

Filename: static_tasking.slurm

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --output=static_tasking.output

cd $SLURM_SUBMIT_DIR
module load gcc
g++ -std=c++20 static_tasking.cpp -o static_tasking -I path/to/taskflow/ -
   pthread
./static_tasking
```

Problem 2: Dependent Async with Dynamic Graph

Task: Use Taskflow's `executor.async` and dynamic tasking features to create a graph that spawns new tasks at runtime. This demonstrates Taskflow's ability to handle dynamically expanding dependency graphs.

Kernel

Filename: `dependent_async.cpp`

```cpp
#include <taskflow/taskflow.hpp>

int main() {
  tf::Executor executor;
  tf::AsyncTask A = executor.silent_dependent_async([](){
    printf("A\n");
  });
  tf::AsyncTask B = executor.silent_dependent_async([](){
    printf("B\n");
  }, A);
  tf::AsyncTask C = executor.silent_dependent_async([](){
    printf("C\n");
  }, A);
  auto [D, fuD] = executor.dependent_async(
    [](){ printf("D\n");
  }, B, C);

  // wait for D to finish, which in turn means A, B, C have finished
  fuD.get();
}
```

Goal: Understand how asynchronous tasks can be dynamically created and synchronized within a Taskflow execution.

Slurm Script

Filename: `dependent_async.slurm`

```
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --output=dependent_async.output

cd $SLURM_SUBMIT_DIR
module load gcc
g++ -std=c++20 dependent_async.cpp -o dependent_async -I path/to/taskflow/
    -pthread
./dependent_async
```

Problem 3: Conditional Task Graph

Task: Use a `condition` task to express an iterative control flow (loop) within a Taskflow graph. The graph should repeatedly execute certain tasks until a given condition is met.

Kernel

Filename: `condition_task.cpp`

```cpp
#include <taskflow/taskflow.hpp>

int main() {
  tf::Executor executor;
  tf::Taskflow taskflow("Condition Task Demo");

  int counter = 0;
  const int limit = 5;

  auto init = taskflow.emplace([&](){
    printf("Initialize counter = %d\n", counter);
  });

  auto loop = taskflow.emplace([&](){
    printf("Loop iteration %d\n", counter);
    counter++;
    return (counter < limit) ? 0 : 1;  // 0 => go back, 1 => exit
  }).condition();

  auto done = taskflow.emplace([](){
    printf("Loop done.\n");
  });

  init.precede(loop);
  loop.precede(loop, done);  // self-edge enables iteration

  executor.run(taskflow).wait();
}
```

Goal: Learn how Taskflow supports control-flow logic such as loops or conditional branches through condition tasks.

Slurm Script

Filename: `condition_task.slurm`

```bash
#!/usr/bin/env zsh
#SBATCH --partition=instruction
#SBATCH --time=00:03:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --output=condition_task.output

cd $SLURM_SUBMIT_DIR
module load gcc
```

```
g++ -std=c++20 condition_task.cpp -o condition_task -I path/to/taskflow/ -
    pthread
./condition_task
```

Problem 4: Reflection

Task: Briefly summarize what you learned from this lab. Discuss how Taskflow's design simplifies task-based parallel programming compared to traditional thread-based programming.

If you like the project, please give it a star! https://github.com/taskflow/taskflow