**Part A: Synchronous Bulk Task Launch**

In Assignment 1, you used ISPC's task launch primitive to launch N instances of an ISPC task (launch[N] myISPCFunction()). In the first part of this assignment, you will implement similar functionality in your task execution library.

**Task System Implementation**

The task system for Part A is designed to execute synchronous bulk task launches efficiently by leveraging multi-core processors for parallel execution. Three distinct implementations are provided, each improving upon the previous:

### **Managing Threads**

In my implementation, I designed a **thread pool** to efficiently manage worker threads. Instead of creating and destroying threads for each task execution, which incurs significant overhead, I initialized a fixed number of worker threads equal to the number of available CPU cores. These threads persist throughout the program’s execution, reducing the overhead associated with frequent thread creation and destruction.

### **Tasks Division**

To distribute tasks among worker threads, I used a **dynamic assignment** strategy. Tasks were placed in a **shared task queue**, from which worker threads fetched tasks as they became available. This approach ensures better load balancing, particularly when tasks have uneven execution times. Each worker continuously checks the queue for new tasks, processes them, and then fetches the next available task. Synchronization was handled using **mutexes and condition variables** to avoid race conditions and ensure threads remain idle without busy-waiting when the queue is empty. This reduces CPU consumption during idle periods, enhancing efficiency.

Each version progressively reduces overhead and improves resource utilization for parallel task execution. The thread pool approach in the latter two versions minimizes overhead and scales better for repeated task launches compared to the spawn-based system.

**Task distribution strategies vary as follows:**

**TaskSystemParallelSpawn:**

Employs *static assignment*, where each thread receives a contiguous range of task IDs at launch. Ensures even distribution but lacks flexibility for tasks with varying complexity.

**TaskSystemParallelThreadPoolSpinning:**

Uses *dynamic assignment* with an atomic counter (next\_task\_id). Worker threads fetch and increment the counter to claim tasks, enabling load balancing as threads take work as they become available.

**TaskSystemParallelThreadPoolSleeping:**

Also uses *dynamic assignment*, but the counter is protected by a mutex to coordinate sleeping threads. Ensures thread-safe task distribution while maintaining efficiency.

Linux x86\_64

================================================================================

Running task system grading harness... (11 total tests)

- Detected CPU with 2 execution contexts

- Task system configured to use at most 2 threads

================================================================================

================================================================================

Executing test: **super\_super\_light...**

Reference binary: ./runtasks\_ref\_linux

Results for: super\_super\_light

STUDENT REFERENCE PERF?

[Serial] 9.398 9.199 1.02 (OK)

[Parallel + Always Spawn] 27.428 27.257 1.01 (OK)

[Parallel + Thread Pool + Spin] 42.843 60.554 0.71 (OK)

[Parallel + Thread Pool + Sleep] 16.578 15.892 1.04 (OK)

================================================================================

Executing test**: super\_light...**

Reference binary: ./runtasks\_ref\_linux

Results for: super\_light

STUDENT REFERENCE PERF?

[Serial] 43.268 58.055 0.75 (OK)

[Parallel + Always Spawn] 60.196 79.866 0.75 (OK)

[Parallel + Thread Pool + Spin] 231.954 254.541 0.91 (OK)

[Parallel + Thread Pool + Sleep] 50.266 66.99 0.75 (OK)

================================================================================

Executing test**: ping\_pong\_equal...**

Reference binary: ./runtasks\_ref\_linux

Results for: ping\_pong\_equal

STUDENT REFERENCE PERF?

[Serial] 700.772 946.713 0.74 (OK)

[Parallel + Always Spawn] 717.851 1081.24 0.66 (OK)

[Parallel + Thread Pool + Spin] 1597.961 2394.826 0.67 (OK)

[Parallel + Thread Pool + Sleep] 703.534 1024.756 0.69 (OK)

================================================================================

Executing test: **ping\_pong\_unequal...**

Reference binary: ./runtasks\_ref\_linux

Results for: ping\_pong\_unequal

STUDENT REFERENCE PERF?

[Serial] 1324.909 1340.104 0.99 (OK)

[Parallel + Always Spawn] 1249.198 1254.191 1.00 (OK)

[Parallel + Thread Pool + Spin] 2565.371 2569.854 1.00 (OK)

[Parallel + Thread Pool + Sleep] 1245.321 1228.481 1.01 (OK)

================================================================================

Executing test**: recursive\_fibonacci...**

Reference binary: ./runtasks\_ref\_linux

Results for: recursive\_fibonacci

STUDENT REFERENCE PERF?

[Serial] 1103.17 1328.446 0.83 (OK)

[Parallel + Always Spawn] 902.372 1072.724 0.84 (OK)

[Parallel + Thread Pool + Spin] 1356.078 1683.348 0.81 (OK)

[Parallel + Thread Pool + Sleep] 904.862 1053.878 0.86 (OK)

================================================================================

Executing test: **math\_operations\_in\_tight\_for\_loop...**

Reference binary: ./runtasks\_ref\_linux

Results for: math\_operations\_in\_tight\_for\_loop

STUDENT REFERENCE PERF?

[Serial] 488.348 490.128 1.00 (OK)

[Parallel + Always Spawn] 514.128 512.57 1.00 (OK)

[Parallel + Thread Pool + Spin] 1605.48 1595.574 1.01 (OK)

[Parallel + Thread Pool + Sleep] 489.594 502.429 0.97 (OK)

================================================================================

Executing test: **math\_operations\_in\_tight\_for\_loop\_fewer\_tasks...**

Reference binary: ./runtasks\_ref\_linux

Results for: math\_operations\_in\_tight\_for\_loop\_fewer\_tasks

STUDENT REFERENCE PERF?

[Serial] 483.261 490.535 0.99 (OK)

[Parallel + Always Spawn] 527.918 528.842 1.00 (OK)

[Parallel + Thread Pool + Spin] 1549.19 1580.939 0.98 (OK)

[Parallel + Thread Pool + Sleep] 488.275 511.296 0.95 (OK)

================================================================================

Executing test: **math\_operations\_in\_tight\_for\_loop\_fan\_in...**

Reference binary: ./runtasks\_ref\_linux

Results for: math\_operations\_in\_tight\_for\_loop\_fan\_in

STUDENT REFERENCE PERF?

[Serial] 251.895 245.021 1.03 (OK)

[Parallel + Always Spawn] 235.755 227.303 1.04 (OK)

[Parallel + Thread Pool + Spin] 784.525 738.868 1.06 (OK)

[Parallel + Thread Pool + Sleep] 232.863 236.381 0.99 (OK)

================================================================================

**Executing test: math\_operations\_in\_tight\_for\_loop\_reduction\_tree...**

Reference binary: ./runtasks\_ref\_linux

Results for: math\_operations\_in\_tight\_for\_loop\_reduction\_tree

STUDENT REFERENCE PERF?

[Serial] 247.653 251.661 0.98 (OK)

[Parallel + Always Spawn] 223.221 226.111 0.99 (OK)

[Parallel + Thread Pool + Spin] 415.409 387.457 1.07 (OK)

[Parallel + Thread Pool + Sleep] 221.157 228.461 0.97 (OK)

================================================================================

**Executing test: spin\_between\_run\_calls...**

Reference binary: ./runtasks\_ref\_linux

Results for: spin\_between\_run\_calls

STUDENT REFERENCE PERF?

[Serial] 392.769 470.184 0.84 (OK)

[Parallel + Always Spawn] 318.561 371.459 0.86 (OK)

[Parallel + Thread Pool + Spin] 448.655 537.72 0.83 (OK)

[Parallel + Thread Pool + Sleep] 316.306 370.57 0.85 (OK)

================================================================================

**Executing test: mandelbrot\_chunked...**

Reference binary: ./runtasks\_ref\_linux

Results for: mandelbrot\_chunked

STUDENT REFERENCE PERF?

[Serial] 305.165 302.718 1.01 (OK)

[Parallel + Always Spawn] 164.896 163.767 1.01 (OK)

[Parallel + Thread Pool + Spin] 226.508 223.087 1.02 (OK)

[Parallel + Thread Pool + Sleep] 165.041 164.488 1.00 (OK)

================================================================================

Overall performance results

[Serial] : All passed Perf

[Parallel + Always Spawn] : All passed Perf

[Parallel + Thread Pool + Spin] : All passed Perf

[Parallel + Thread Pool + Sleep] : All passed Perf

**Task System Implementation for Parallel Computing PART\_B**

### **Dependency Tracking in Part B**

For Part B, I introduced a **dependency graph** to track and enforce task dependencies. Each task maintained a list of dependent tasks that needed to complete before execution. I used an **adjacency list representation** where each node represented a task and stored its dependencies. A **countdown counter** tracked the number of remaining dependencies, and only when this count reached zero was a task pushed to the ready queue. A **mutex-protected condition variable** ensured that workers executed only tasks with all dependencies met.

### **Performance Analysis**

#### **When Simpler Implementations Perform Well**

In some cases, simpler implementations like the **sequential task system** or the **spawn-every-launch system** performed comparably to the thread pool-based implementation. This was particularly evident in tasks with **low computational intensity**, where the overhead of managing multiple threads outweighed any potential speedup from parallel execution. For example, in the **small Mandelbrot test**, the sequential implementation performed well since each task's execution time was short, making parallelization overhead dominate execution time.

#### **Spawn-Every-Launch vs. Thread Pool**

The **spawn-every-launch** implementation performed similarly to the thread pool approach in cases where there were **only a few tasks** with long execution times (e.g., large **chunked Mandelbrot rendering**). However, when dealing with many short-lived tasks, the overhead of creating and destroying threads for each launch became a bottleneck, making the thread pool approach superior.

### **Test Case Implementation**

To evaluate my implementation, I designed a test that **executes multiple dependent tasks** in a tree-like dependency structure. The test generated a set of root tasks that spawned sub-tasks, ensuring that no task executed before its dependencies completed.

#### **Purpose & Verification**

This test verified that:

1. The dependency system correctly enforced execution order.
2. The thread pool efficiently executed tasks without deadlocks.
3. Performance improved compared to sequential execution.

I validated my results by inserting **logging checkpoints** and ensuring that dependent tasks never executed before their dependencies. The test also measured execution times, confirming that parallel execution provided a significant speedup over sequential execution.

This test revealed an inefficiency in my initial dependency tracking logic, leading me to optimize the **ready queue management** by reducing redundant checks on task completion, which improved performance in dependency-heavy workloads.

**Introduction**

This project is an assignment for a parallel computing course, aimed at implementing and comparing different task systems for executing tasks in parallel. The implemented task systems include:

- **Serial Task System:** Executes tasks sequentially in a single thread.

- **Parallel Task System with Spawning Threads:** Spawns new threads for each bulk task launch.

- **Parallel Task System with Spinning Thread Pool:** Uses a thread pool that spins (busy-waits) when idle.

- **Parallel Task System with Sleeping Thread Pool:** Uses a thread pool that sleeps when idle, optimized for efficiency.

**Prerequisites**

To build and run this project, ensure you have the following:

- A C++ compiler supporting C++11 (e.g., `g++` version 4.8 or later).

- The `make` utility for building the project.

- A Unix-like environment (e.g., Linux, macOS, or WSL on Windows).

**Building the Project**

The project uses a `Makefile` to compile the source code. Follow these steps to build it:

1. Navigate to the project directory (e.g., `part\_b/):

2. Compile the code using make:

This generates the runtasks executable in the current directory.

**Running the Project**

The runtasks executable allows you to run tests and evaluate the task systems. Here are some common usage examples:

**Running the Full Test Harness**

To run multiple tests using the provided Python script:

python3 ../tests/run\_test\_harness.py -n <num\_threads> -t <test1> <test2> ...

Example:

python3 ../tests/run\_test\_harness.py -n 16 -t simple\_test\_async super\_super\_light\_async

**Testing**

The project includes tests to verify correctness and performance, defined in `tests/tests.h` and listed in tests/main.cpp.

**Running Tests**

- Run individual tests with `runtasks` as shown above.

- Use the Python test harness for a full suite and comparison with a reference implementation.

**Interpreting Results**

- Correctness: A passing test indicates correct output. Failures suggest issues in task execution or dependencies.

- Performance: The test harness reports ratios against a reference. A ratio ≤ 1.2 (within 20%) is typically acceptable.

**Extending the Project**

To create custom tests:

1. **Define a New Test** in tests/tests.h:

.cpp

class MyCustomTask : public IRunnable {

public:

void runTask(int task\_id, int num\_total\_tasks) override {

// Custom task logic

}

};

void myCustomTest(ITaskSystem\* t, int num\_threads) {

// Test setup and execution

}

2. Register the Test in tests/main.cpp:

.cpp

const char\* test\_names[] = { ..., "my\_custom\_test" };

TestFunc tests[] = { ..., myCustomTest };

int n\_tests = sizeof(tests) / sizeof(TestFunc);

3. Run your test using runtasks or the test harness.

**Contributing**

- Optimizing the sleeping thread pool.

- Adding new synchronization methods.

- Supporting more complex task graphs.

**Running Example:**

