Part A:

Task System Implementation

Our task system implementation consists of several variants, each designed to optimize parallel task execution using different threading strategies. Below, we describe how each system works and answer key questions regarding thread management and task assignment.

General System Overview

The task system provides multiple implementations for executing tasks, ranging from a simple serial execution to a more sophisticated thread pool-based system. The core interface, ITaskSystem, is extended by various implementations, including:

1. **TaskSystemSerial** - Executes tasks sequentially on the calling thread.
2. **TaskSystemParallelSpawn** - Spawns a new thread for each task.
3. **TaskSystemParallelThreadPoolSpinning** - Uses a thread pool with busy-waiting.
4. **TaskSystemParallelThreadPoolSleeping** - Uses a thread pool with condition variables for efficient sleeping and waking of threads.

Thread Management

Thread management varies across implementations, with the most sophisticated being TaskSystemParallelThreadPoolSleeping, which maintains a persistent thread pool. In contrast, TaskSystemParallelSpawn spawns and joins threads dynamically, leading to higher overhead.

- **TaskSystemParallelSpawn**: Threads are created for each task, leading to significant overhead when handling many small tasks. A mutex is used to safely add threads to the vector.

- **TaskSystemParallelThreadPoolSpinning**: A fixed-size thread pool is initialized. Each worker thread continuously spins, checking for new tasks using an atomic lock, which can lead to CPU wastage but avoids thread creation overhead.

- **TaskSystemParallelThreadPoolSleeping**: Uses a persistent thread pool where worker threads sleep when idle and are awakened via condition variables. This approach minimizes CPU usage and improves efficiency, particularly for varying workloads.

Task Assignment Strategy

Task assignment determines how work is distributed among worker threads. Our system supports both static and dynamic assignment strategies:

- **TaskSystemParallelSpawn**: Uses a static assignment where each task is directly mapped to a new thread. While straightforward to implement, this approach doesn't scale well with large numbers of tasks.

- **TaskSystemParallelThreadPoolSpinning**: Uses dynamic assignment with a shared task index. Each worker thread fetches the next available task using a spinlock, allowing workers to compete for tasks as they become available.

- **TaskSystemParallelThreadPoolSleeping**: Implements dynamic assignment where tasks are pulled by worker threads upon waking up via a condition variable. This provides better load balancing while minimizing CPU utilization during idle periods.

Performance Considerations

- TaskSystemParallelSpawn incurs high overhead due to frequent thread creation and destruction, but can be effective for workloads with few, long-running tasks.

- TaskSystemParallelThreadPoolSpinning avoids thread creation overhead but wastes CPU resources due to busy waiting. This approach can be beneficial for workloads where tasks arrive rapidly and CPU utilization is less of a concern.

- TaskSystemParallelThreadPoolSleeping achieves the best balance by maintaining a fixed thread pool and waking workers efficiently when tasks are available. This approach is particularly effective for mixed workloads and environments where CPU resources need to be conserved.

Mandelbrot Chunked Test Result

```
================================================================================
Test name: mandelbrot_chunked
================================================================================
[Serial]:    [586.056] ms
[Parallel + Always Spawn]:    [176.779] ms
[Parallel + Thread Pool + Spin]:    [181.042] ms
```

The performance results align with our expectations: both parallel implementations significantly outperform the serial approach. The similarity in performance between the spawning and spinning implementations suggests that for this specific workload (Mandelbrot), the benefit of avoiding thread creation overhead is roughly balanced by the cost of spin-waiting. Different workloads might show more pronounced differences between these approaches.

It's worth noting that the ThreadPoolSleeping implementation isn't included in these test results. Based on our analysis, we would expect it to perform similarly to the spinning implementation but with better CPU utilization, especially under varying workloads or when the system is handling multiple concurrent processes.

Conclusion

Our implementation provides different levels of parallel execution efficiency. The TaskSystemParallelThreadPoolSleeping approach is the most optimal for general use cases, as it minimizes CPU wastage and reduces overhead associated with thread management. By using condition variables, it efficiently schedules tasks dynamically and ensures worker threads are utilized effectively without unnecessary spinning.

Each implementation has its strengths depending on the specific workload characteristics, making our task system adaptable to different performance requirements and resource constraints.

Part B:

Overview:

We are implementing a multi-threaded task system where tasks are executed in parallel using a thread pool. Some tasks may have dependencies, meaning they cannot start until certain other tasks finish.

TaskSystemParallelThreadPoolSleeping:

1. **runAsyncWithDeps(IRunnable* runnable, int num_total_tasks, const std::vector<TaskID>& deps)**
   -Starts tasks asynchronously while ensuring dependencies are resolved.
2. **sync()**
   -Waits for all tasks to finish before returning.

1. Constructor (TaskSystemParallelThreadPoolSleeping)

- Creates a **thread pool** where each thread runs the workerThread() function.
- Initializes control variables (stop = false, runningTasks = 0).

2. Destructor (~TaskSystemParallelThreadPoolSleeping)

- Signals all threads to stop and joins them (waits for them to finish).

3. runAsyncWithDeps()

- Accept a task, check dependencies, and either queue it for execution **(if dependencies are resolved)** or **store it until dependencies finish**.
- It generates a **unique task ID** for this bulk task launch.
- If the task has **no dependencies**, add it to readyQueue and notify a worker.
- If the task has **dependencies**, store it in task_map and track how many dependencies it has in dependencyCount.
- Return the **Task ID** (not really used, but required by the interface).

4. Worker Thread Function (workerThread())

- Pick tasks from the queue and execute them.
- When a task finishes, check if any **waiting tasks** can now run.
- **Wait for a task** (using queueCondition.wait).
- **Run the task**.
- **Mark it as completed** and update dependency tracking.
- **Move dependent tasks to readyQueue** if all their dependencies are resolved.

5. sync()

- Block execution until all tasks are finished.
- Continuously check runningTasks.
- If all tasks are done (runningTasks == 0), return.
- Otherwise, wait.