15-618 Project Proposal: Distributed work queues with task priority

Group: Akshay Joshi (akshayjo) & Amogha Hegde (amoghah)

URL: https://shadowcode0007.github.io/

Summary:

We intend to build a lock-free, priority-aware work-stealing deque for dynamic task scheduling in parallel computing systems. In this model, each worker thread maintains a priority deque of its own and is able to push and pop tasks at the local end, whereas other waiting threads steal tasks from the top, according to task priorities when required. The parallelizable parts are insertion and execution per-thread and random task stealing among threads. The aim is to construct a totally lock-free organization based on atomic operations (i.e., compare_exchange, fetch_add) in OpenMP, optionally supplemented by MPI for parallel environments. It will be tested on a multi-core shared memory platform (8–128 cores), and in multi-node setup using MPI with one process per node and OpenMP threads per core. It will be benchmarked with task throughput, steal attempts, and priority correctness with various workloads.

Also we will experiment by having different counts of elements being pulled for a single steal (like steal n elements at once) and compare the tradeoffs between communication and inefficient computation.

Background:

Effective load balancing and task scheduling are essential to providing high performance in parallel computing systems. One common approach is work stealing, where idle processors dynamically steal tasks from loaded processors to maintain the workload balance. A key data structure enabling this technique is the double-ended queue (deque). In a typical work-stealing approach, each worker thread maintains a local deque where it can push and pop work from one end, and other threads will steal work from the other end when idle.

To enhance responsiveness and support real-world scheduling requirements, this work explores a lock-free, priority-aware work-stealing deque. Both local execution and stealing in this approach respect the task priority. Preference of higher-priority task execution guarantees timely processing of high-priority workloads. The structural core relies on lock-free atomic operations such as compare_exchange and fetch_add, avoiding the use of traditional locking to avoid contention and improve scalability.

Each thread Pushes, Pops, or Steals separately, and the data structure allows for fine-grained concurrency. OpenMP is employed for parallelism in shared-memory systems, and optionally MPI for scaling across more than a node. The implementation will be exercised on 8–128 processor shared-memory systems and distributed clusters with MPI. Benchmarking will quantify performance in the form of overall throughput on tasks, successful steals, and adherence to task priority in the face of changing and unpredictable workloads.

We will also compare this by having a random steal (maintain without priority) to compare and measure the overhead v/s efficiency we gain by maintaining the priority of the next task to be stolen. We will experiment by having different counts of elements being pulled for a single steal (like steal n elements at once) and compare the tradeoffs between communication and computation.

Every task has a priority level P, and deques are sorted in two ways:

1. Multiple Deques by Priority (Priority Buckets)
In thread i within priority p, let D_p[i] indicate the corresponding deque. The thread always:

Deque operations at the topmost current highest-priority task:
task = pop_bottom(D_p[i]) for p = MAX_PRIORITY down to 0

When stolen:
task = steal_top(D_p[j]) for p = 0 to MAX_PRIORITY

A simplistic pseudo code would look something like this, but done in a lock free manner for the dequeue internally

```
while not done:
   task = pop_local_highest_priority_task()
   if task != NULL:
      execute(task)
   else:
      victim = random_other_thread()
      task = steal_lowest_priority_task_from(victim)
      if task != NULL:
         execute(task)
      else:
```

spin_wait()

The Challenge:

If we have n threads and every thread has 1000 mixed-priority tasks. In every iteration, a thread will execute k local tasks. Once it has done executing its local tasks, it will attempt to steal tasks from an arbitrarily chosen victim thread. Meanwhile, other threads are stealing, which may result in priority inversion when lower-priority stolen tasks get executed ahead of higher-priority tasks due to non-atomic scan-steal interactions.

Option 1: Maintain a shared deque structure, which all threads reference. Centralizing this makes simple stealing but increases false sharing and cache line contention.

Option 2: Every thread maintains a fully local deque and shares stolen task metadata periodically. This reduces contention but introduces state synchronization overheads, especially for task status and completion tracking.

These methods will be implemented and compared based on throughput, steal effectiveness, and priority correctness. It will degrade the performance if the threads frequently battle over the identical victim deque or if atomic operation overhead is unacceptable under saturation. Load balancing well is particularly not good if task priorities are skewed because high-priority tasks can backlog on fewer threads. Ensuring fairness, correctness, and low overhead in this dynamic environment without sacrificing the lock-free property is our goal for this.

Resources:

The task scheduler and the deque system will be written entirely in C/C++. This system will use OpenMP for shared memory parallelism and use MPI optionally for distributed node cases.. To guide our architectural decisions and ensure correctness, we are going through relevant research papers and guides on lock-free data structures, work stealing, and priority scheduling. We also plan to point to forums and low-level atomic operation documentation (std::atomic, __sync_*, or __atomic_* intrinsics) as needed. Testing and benchmarking will be done on multicore machines available and university GHC clusters based on scalability needs.

Goals and Deliverables

Plan to Achieve:

- ➔ Implement a priority deque with local push pop and remote steal operations. Initially the part of stealing the work and updating my queue and processing the elements in my queue is done sequentially

- ➔ Introduce lock-free atomic building blocks (e.g., compare_exchange, fetch_add) to avoid synchronization locks across threads in stealing tasks from the dequeue.

- ➔ Compare different priority-handling approaches: separate deques by priority level vs. single-deque combination of priority tagging.

- ➔ Implement a distributed-memory version using MPI, with each process being a node with local worker threads (OpenMP), and stealing between processes is simulated by message-passing.

- ➔ Measure tradeoff between stealing 1 task v/s n tasks at once with respect to time taken, synchronisation time, communication time and other important parameters.

- ➔ Perform extensive benchmarking on performance characteristics such as task throughput, steal attempts, priority accuracy, idle time, and scaling behavior.

- ➔ Finally benchmark these results without having a priority based stealing (random stealing) to compare if the extra overhead introduced is worth it.

Platform Choice:

X86-64 multi-core cpu architecture with support for OpenMP & MPI libraries; source code in C/C++

Rough Schedule:

Week 1: Setup, approach design finalise

Week 2: Lock free implementation for the dequeue complete

Week 3: OpenMP and MPI implementation

Week 4: Benchmark testing and final report