HPC - TBB lab 2 - parallel do & thread local

Table of Contents

Files for TBB lab 2: tbb-lab-02.zip

1 Adding work on the go: `parallel_for_each`

`parallel_for` (introduced in the previous lab) allows processing in parallel some workload known in advance. Sometimes, however, we don't know how much total work is available. Imagine, for instance, traversing a graph to process each edge. If the graph is stored as adjacency lists, it's hard to know how many edges are there without traversing the whole graph. Of course, we could generate a `tbb::task` for each edge, but TBB offers a more convenient solution, `tbb::parallel_for_each`.

`tbb::parallel_for_each(first, last, body)` applies `body()` over elements from the sequence `[first,last)`. The function object `body` takes one or two arguments: (1) the item; (2) a `tbb::feeder` through which `body` can dynamically add (feed) further elements to process.

`pardo.cpp` shows an example of traversing a DAG:

```
tbb::parallel_for_each(&node, &node+1, [&graph, &par_edge_count,
                                         &stdout_mutex]
                       (const node_t& node, tbb::feeder<node_t>& feeder) {
        {
            // this mutex is just for stdout formatting
            std::scoped_lock lock(stdout_mutex);
            std::cout << "Now processing: " << node << "." << std::endl;
        }  // scoped_lock mutex will be released here
        for (const auto& neighbor : graph[node]) {
            {
                // this mutex is just for stdout formatting
                std::scoped_lock lock(stdout_mutex);
                std::cout << "edge: " << node << "->" << neighbor
                          << std::endl;
            }  // scoped_lock mutex will be released here
            par_edge_count++;  // this variable is atomic<int>, so no races
```

```
            feeder.add(neighbor);
        }
    }
    );
```

Exercise: rewrite `nqueens.cpp` from the previous lab to use `parallel_for_each`.

2 Per-thread local variables: `enumerable_thread_specific`

When individual tasks need to update a shared data structure, to avoid data races, we need to either apply mutual exclusion, atomic variables, or lock-free data structures. However, sometimes the update can be deferred until the parallel work is over. TBB proposes a convenient mechanism called `tbb::enumerable_thread_specific<T>`. A task is executed by a thread and each thread accesses its local field using the `local()` method of the enumerable. After the parallel work is done, one can just iterate through the enumerable.

`pardo_th_sp.cpp` updates the DAG traversal with per-thread counters:

```
tbb::parallel_for_each(&node, &node+1, [&graph, &counters, &stdout_mutex]
                       (const node_t& node, tbb::feeder<node_t>& feeder) {
        {
            // this mutex is just for stdout formatting
            std::scoped_lock lock(stdout_mutex);
            std::cout << "Now processing: " << node << "." << std::endl;
        }  // scoped_lock mutex will be released here
        for (const auto& neighbor : graph[node]) {
            {
                // this mutex is just for stdout formatting
                std::scoped_lock lock(stdout_mutex);
                std::cout << "edge: " << node << "->" << neighbor
                          << std::endl;
            }  // scoped_lock mutex will be released here
            (counters.local())++;
            feeder.add(neighbor);
        }
    }
    );
int par_edge_count = 0;
for (const auto& counter : counters) {
```

```
    par_edge_count += counter;
}
```

Exercise: apply `enumerable_thread_specific` to `nqueens.cpp`.

3 Concurrent containers

TBB defines a few data structures optimized for parallel access. A naive solution is to have a standard data structure and a global lock: whenever a task needs to access the data, it waits on the lock until other tasks finish. Such sequenced access is suboptimal, as many data structures naturally support parallelism (consider for example adding elements to a hash map). A concurrent data structure uses more fine-grained locking to optimize access performance. The actual locking mechanism depends on implementation but rarely uses operating system-level mutexes (as they are too heavy). Consistency models extend the standard semantics of concurrent operations on such data structures.

TBB defines 3 classes of concurrent containers:

1. Queues, like `tbb::concurrent_queue` we saw in the previous class.
2. Maps like `tbb::concurrent_unordened_map` and `tbb::concurrent_hash_map`.
3. A vector: `tbb::concurrent_vector`.

Exercise: Solve the single source shortest path problem in a randomly-initialized graph. Use `tbb::concurrent_hash_map` (or `tbb::concurrent_unordered_map`) that maps a node to its distance from the source; and `tbb::concurrent_priority_queue` to keep a list of nodes to visit. See https://stackoverflow.com/questions/23501591/tbb-concurrent-hash-map-find-insert for example.

4 Bibliography

- Concurrent containers: https://software.intel.com/en-us/node/506169

Date: 2023/06/02

Author: Krzysztof Rządca

Created: 2024-05-29 Wed 16:06
Emacs 25.3.50.1 (Org mode 8.2.10)
Validate