Profiling Multithreaded RDF: NUMA Aware Parallelism

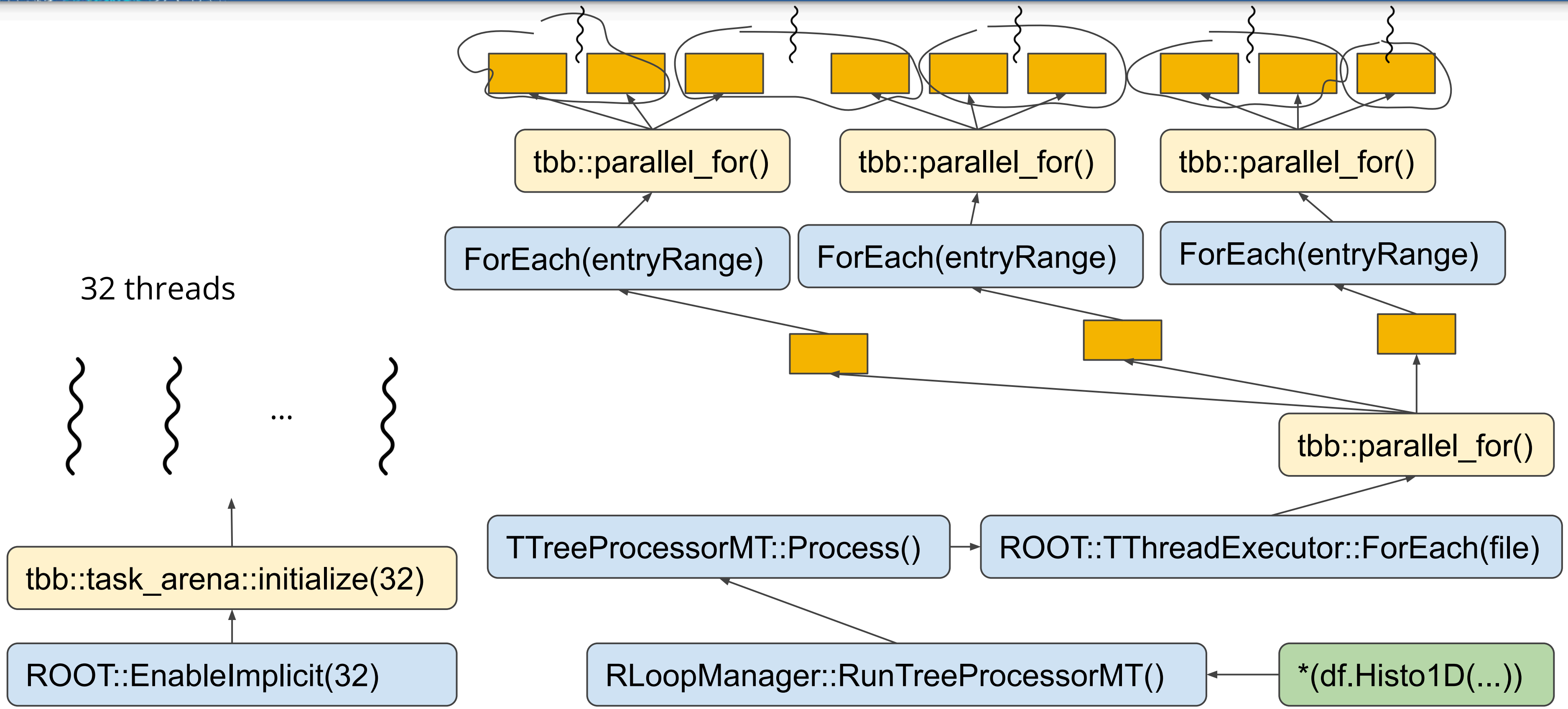
Ivan Kabadzhov

ROOT
Data Analysis Framework
https://root.cern

Outline

1. Understanding the parallelism of Multithreaded RDF
   a. Slots, Tasks, Threads, TBB
2. Dashboards of Multithreaded RDF
   a. Per Slot, Per Thread, Task duration distribution
3. NUMA Architectures and TBB solutions
4. Using TBB API to pin tasks to NUMA domains + Results
5. Comparison with the TNUMAExecutor implementation

Multithreaded RDF



Multithreaded RDF

- Each thread executes multiple of tasks (by default proportional to number of threads) → **granularity**
- Create a slot for each thread - each task reads/writes data allocated for a certain slot; different tasks of the same thread can be done in different slots
- A thread might start in core X, run in core Y, end in core Z

Task Dashboards

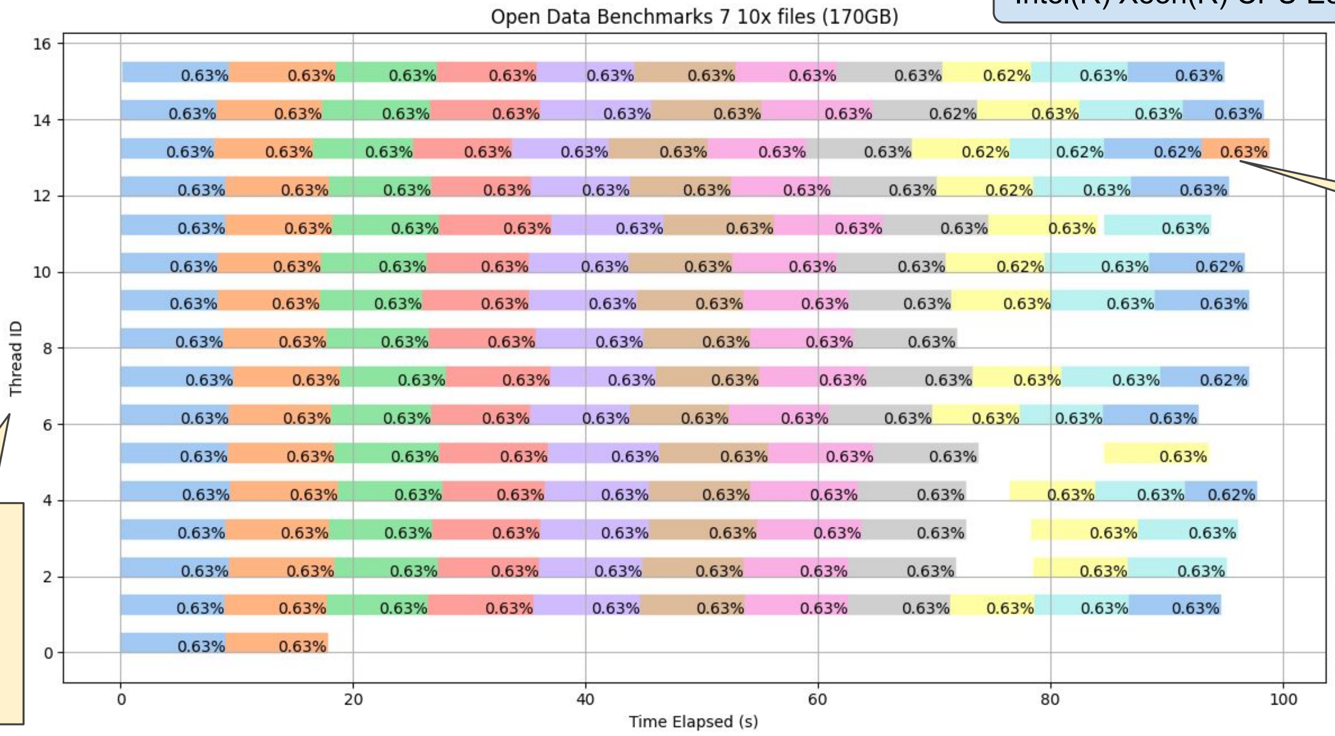
Patch RLoopManager::RunTreeProcessorMT(). Answers:

1. are there long tails of execution?
2. are there gaps between tasks?

Some of the plots suffer interesting problems. But why?

Task Dashboards

Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz

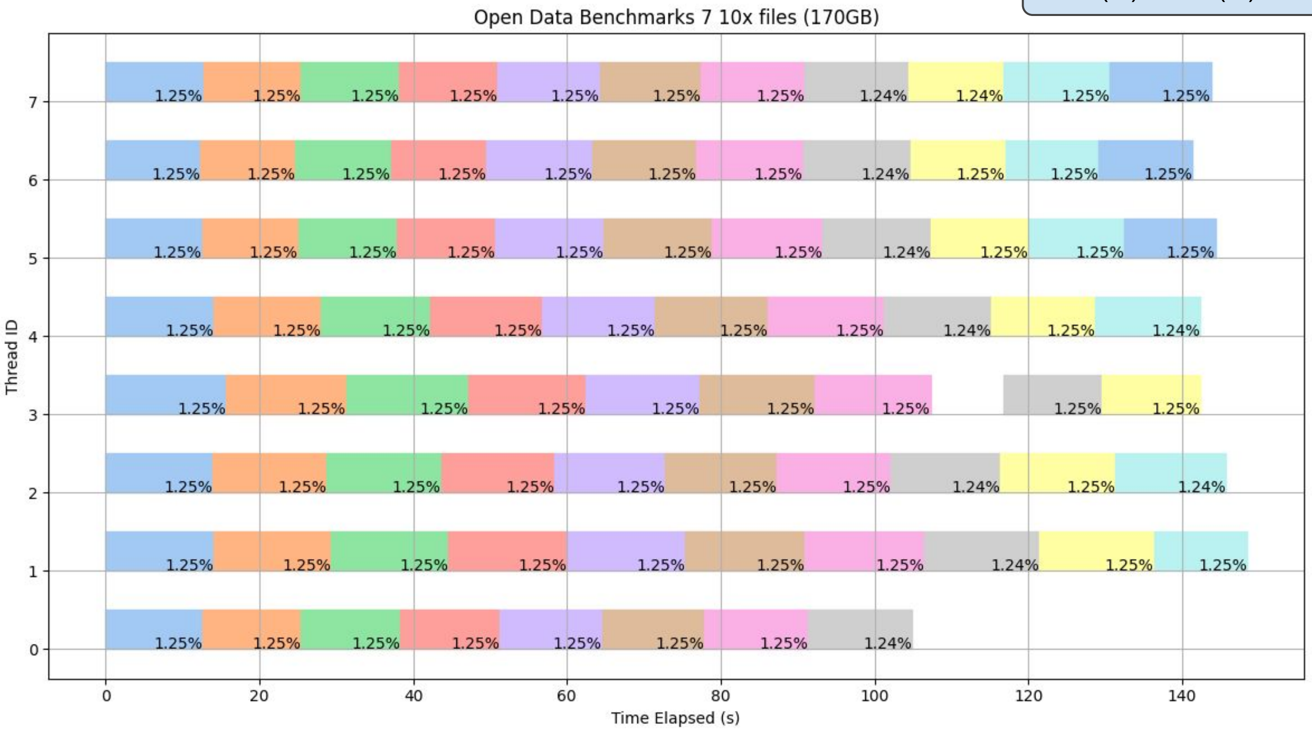


Percentile of the total entry range that must be processed.

I am mapping the threads to indices, depending on the first time they started a task

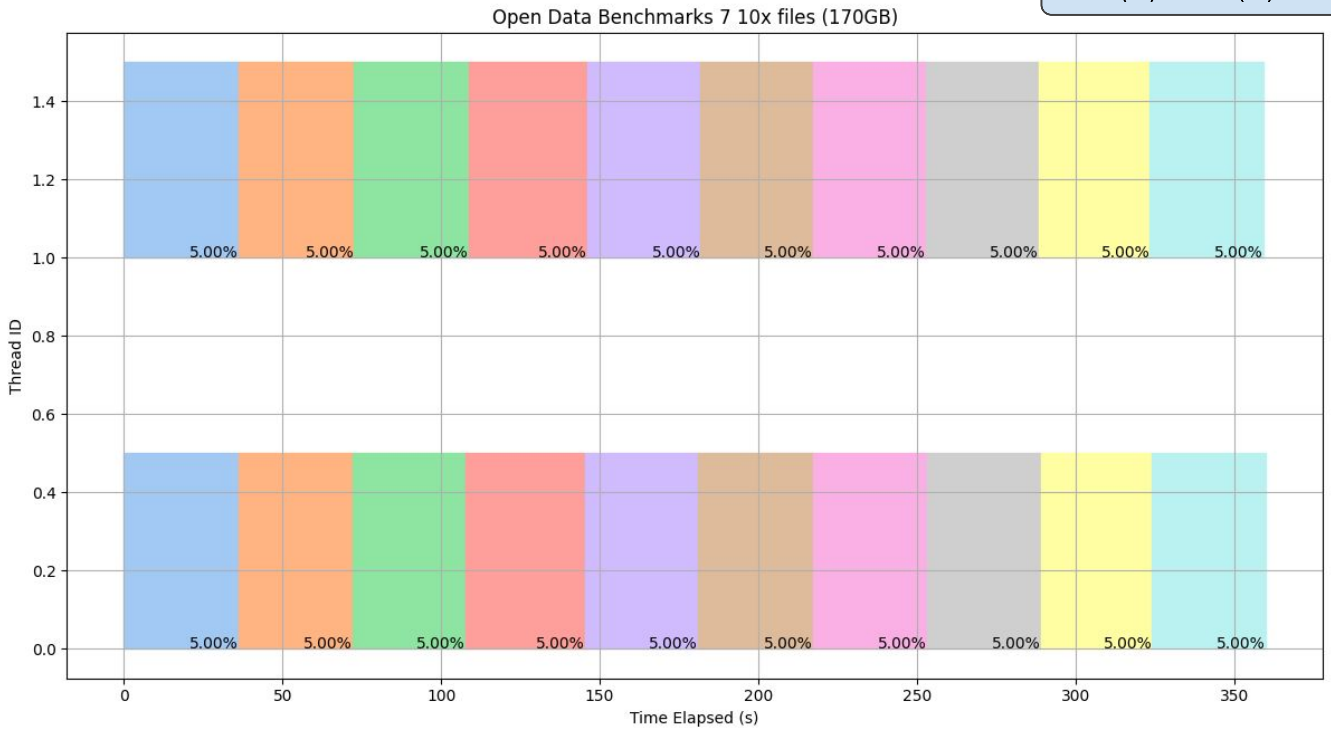
Task Dashboards

Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz



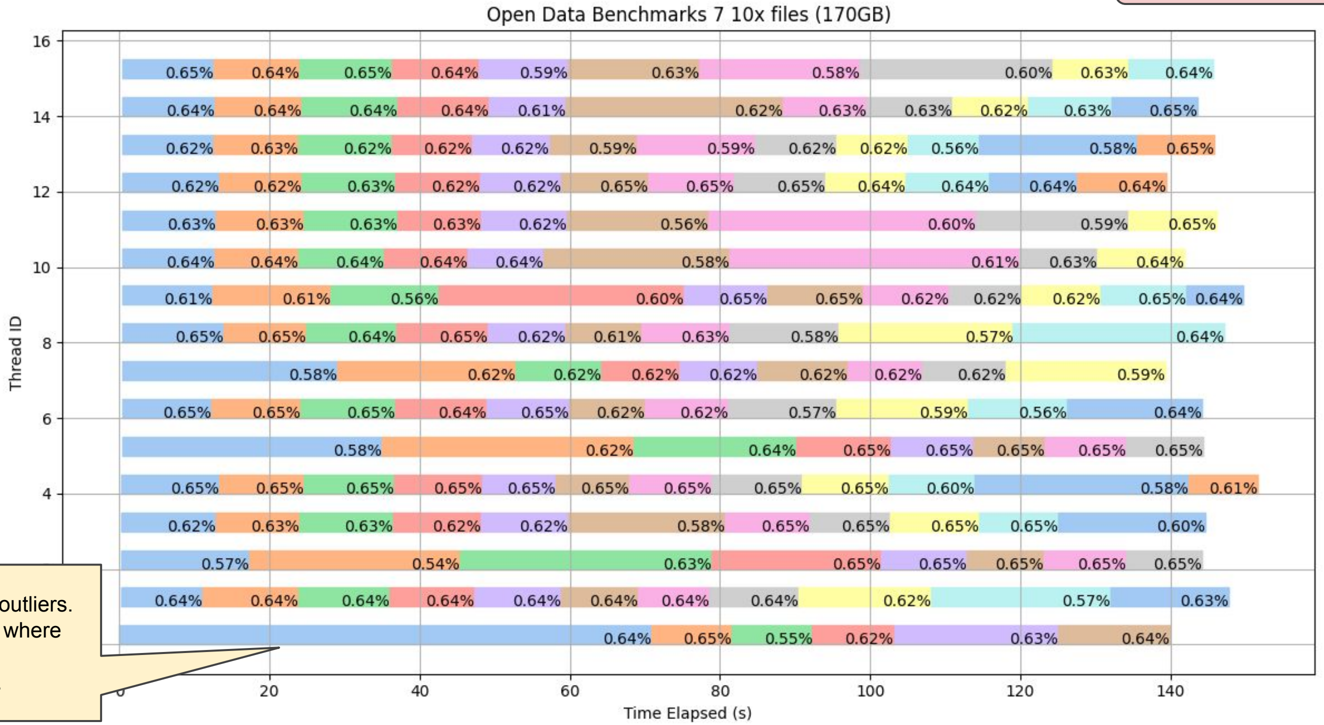
Task Dashboards

Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz



Task Dashboards

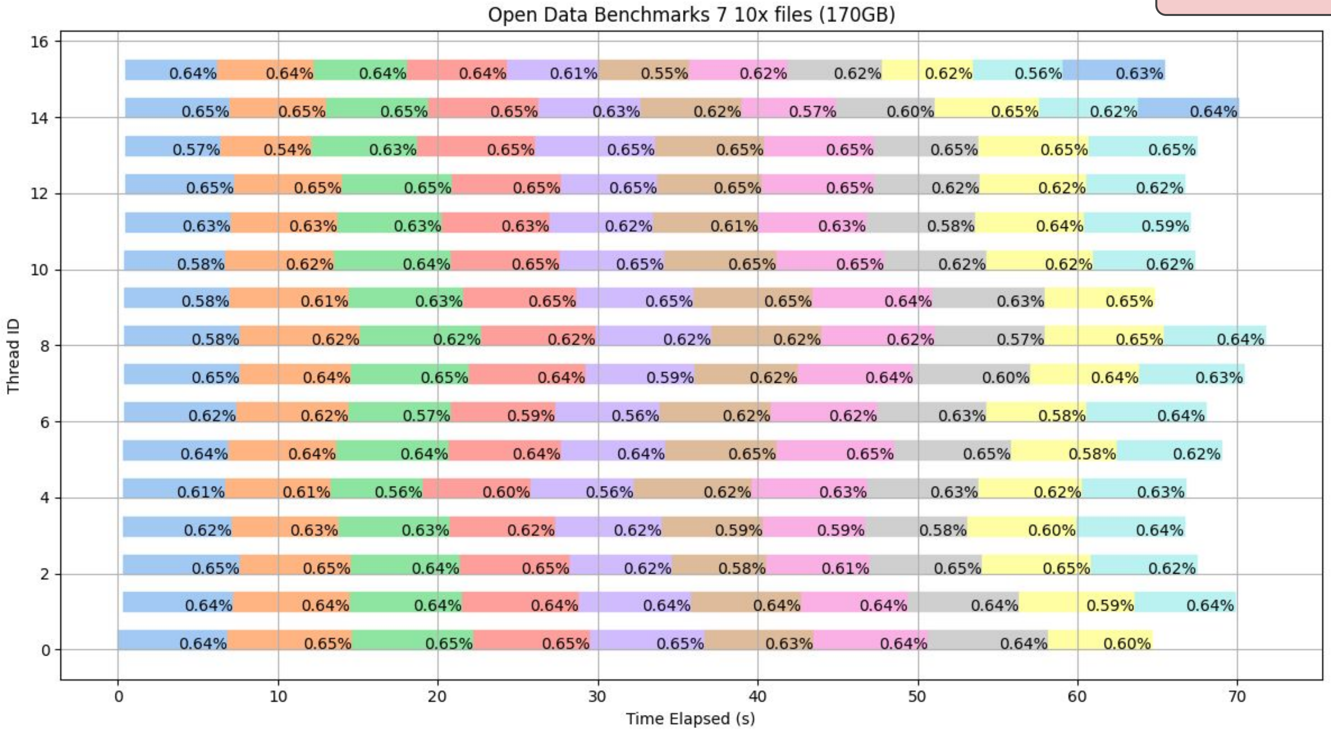
AMD EPYC 7302 16-Core Processor



These were the outliers. I discarded runs where there were such prolonged tasks.

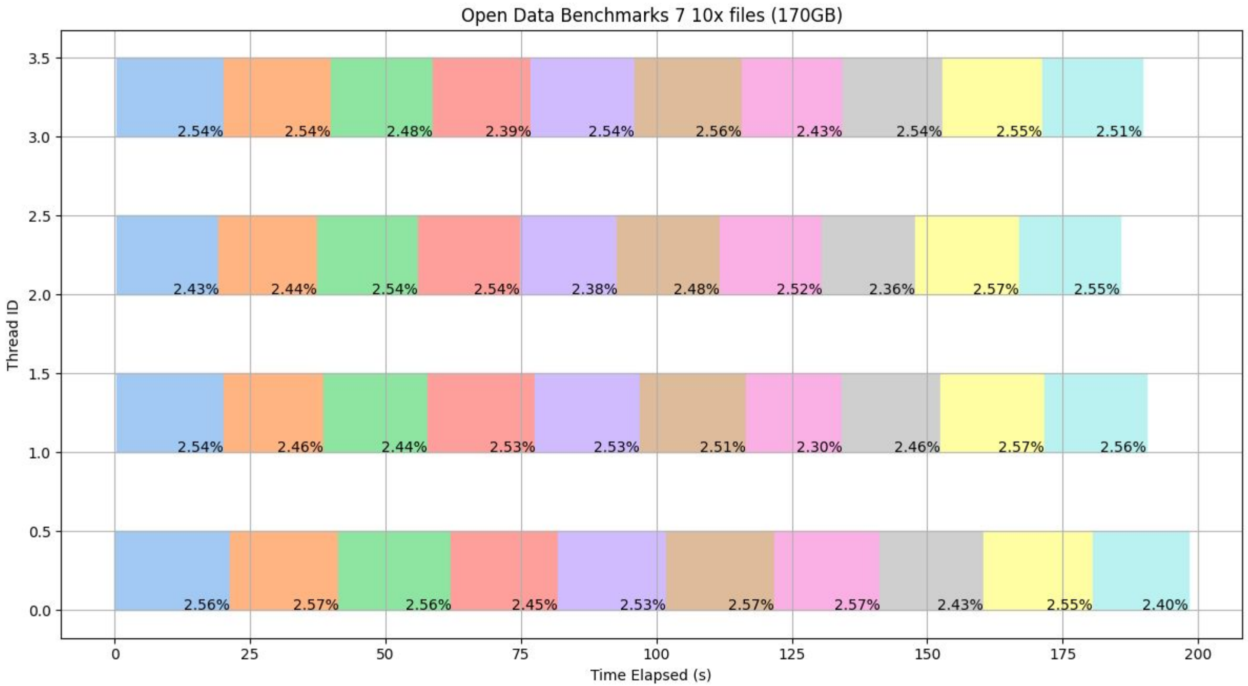
Task Dashboards

AMD EPYC 7302 16-Core Processor



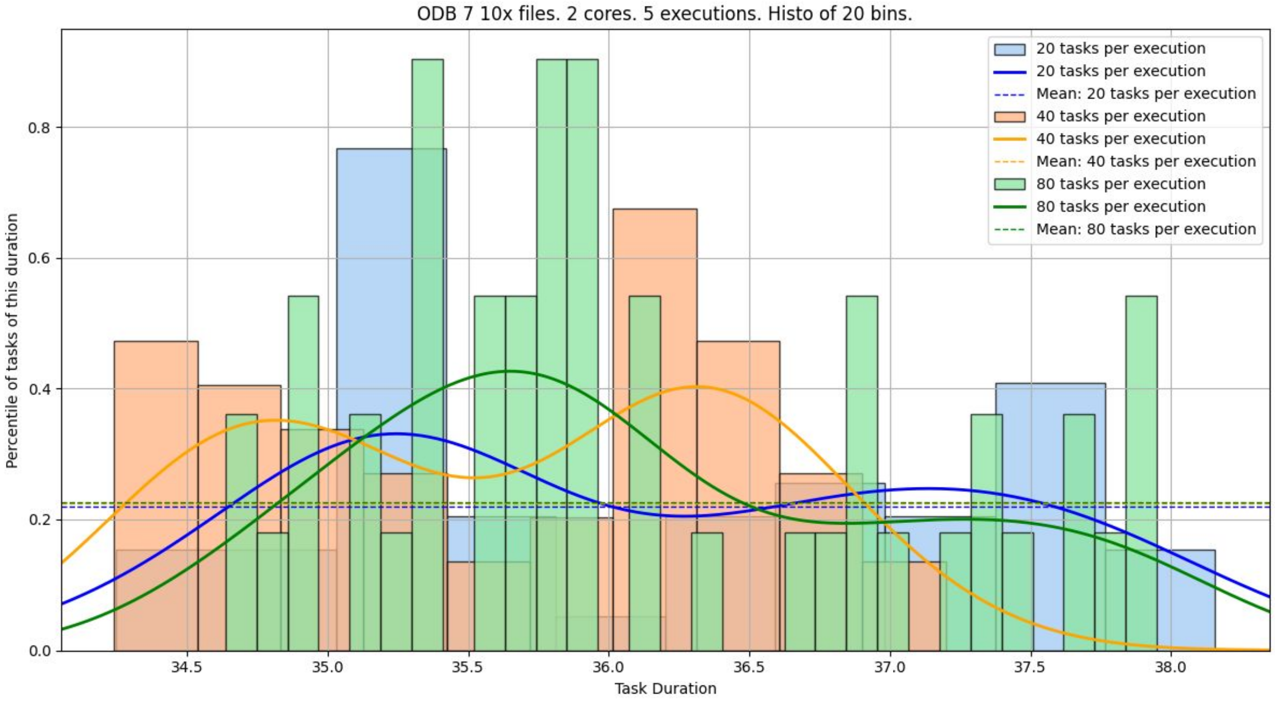
Task Dashboards

AMD EPYC 7302 16-Core Processor



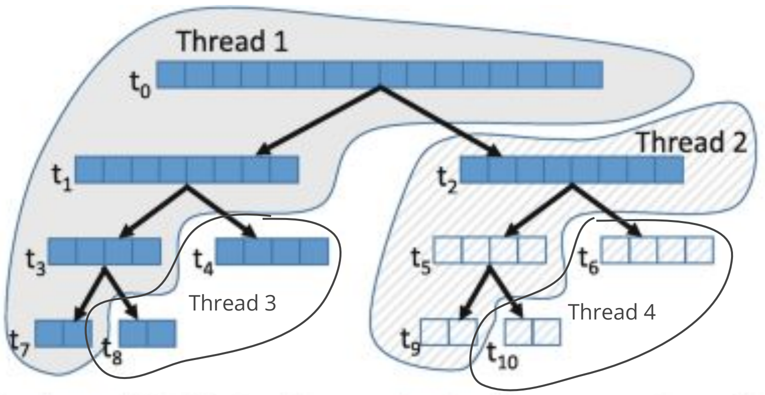
Granularity Task duration distribution

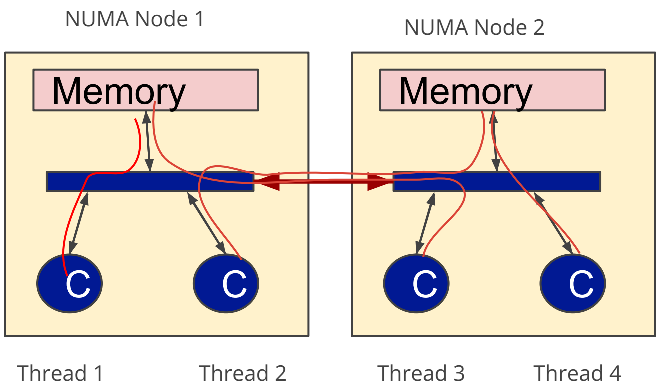
Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz



TBB Major Points

- By default oneTBB does not pin threads to cores.
- By default, oneTBB uses work-stealing and auto-partitioning to balance the load across cores.
- Dynamic nature ⇒ good composability overall (source)





Potential NUMA Effects

- Main Thread allocating a lot of memory, then all threads need to access this memory → isolation is of no help!
- Thread migrating between different NUMA domains within the execution of the same task
  - If load distribution requires migrating a thread off of a processor → the OS pick an arbitrary new processor with sufficient capacity.
  - The newly selected processor should not have higher access costs to the memory → If no free processor matching that criteria, the OS migrates to a processor where memory access is more expensive

TBB NUMA Support (I)

Since TBB 2020 Initial Release, Source: tbb::info::numa_nodes()

1. Need to identify the system topology (fails on some machines!)
2. Create a vector of task arenas for each NUMA domain, statically split the input space to each domain ⇒ each domain can only work on its own partition

```
std::vector<tbb::numa_node_id> numa_nodes = tbb::info::numa_nodes();
std::vector<tbb::task_arena> arenas(numa_nodes.size());
std::vector<tbb::task_group> task_groups(numa_nodes.size());
for (int i = 0; i < numa_nodes.size(); i++) { arenas[i].initialize(tbb::task_arena::constraints(numa_nodes[i])); }
for (int i = 0; i < numa_nodes.size(); i++) {
    arenas[i].execute([&] {
        task_groups[i].run([&] {
            auto lowerBound = start + i * (end - start + size) / size;
            auto upperBound = std::min(start + (i + 1) * (end - start + size) / size, end);
            tbb::parallel_for(lowerBound, upperBound, step, f); }); }); }
for (int i = 0; i < numa_nodes.size(); i++) { arenas[i].execute([&task_groups, i] { task_groups[i].wait(); }); }
```

TBB NUMA Support (II)

Already present in TBB 2017, Source: affinity_partitioner

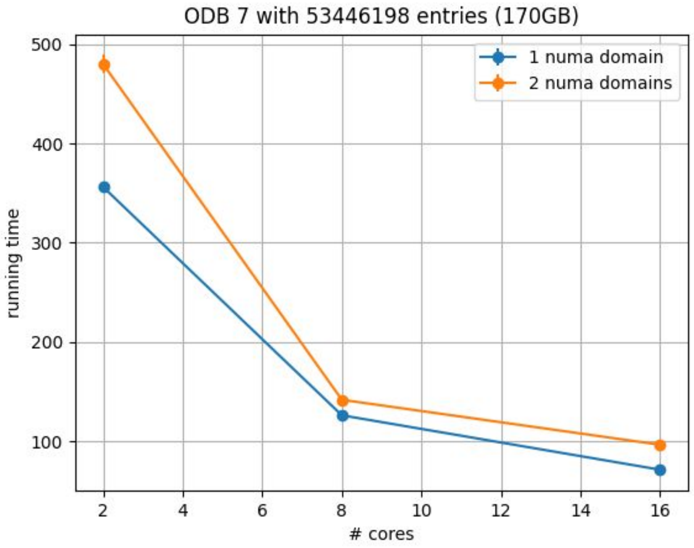
It not only automatically chooses the grain size, but also optimizes for cache affinity and tries to distribute the data uniformly among threads. Using affinity_partitioner can significantly improve performance when:

1. The computation does a few operations per data access.
2. The data acted upon by the loop fits in cache.
3. The loop, or a similar loop, is re-executed over the same data.
4. The affinity_partitioner object lives between loop iterations. It remembers where iterations of the loop ran, so that each iteration can be **hinted** to the same thread that executed it before.

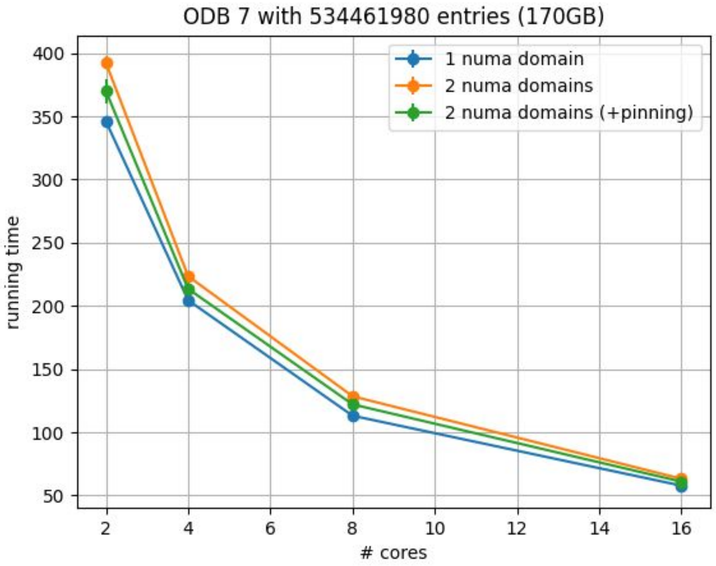
```
tbb::task_arena arena;
arena.initialize(max_concurrency);
arena.execute([&] {
    static tbb::affinity_partitioner ap; //this is the idea, but does not come out of the box, WIP!
    tbb::parallel_for(start, end, step, f, ap);
});
```

Run Times

Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz

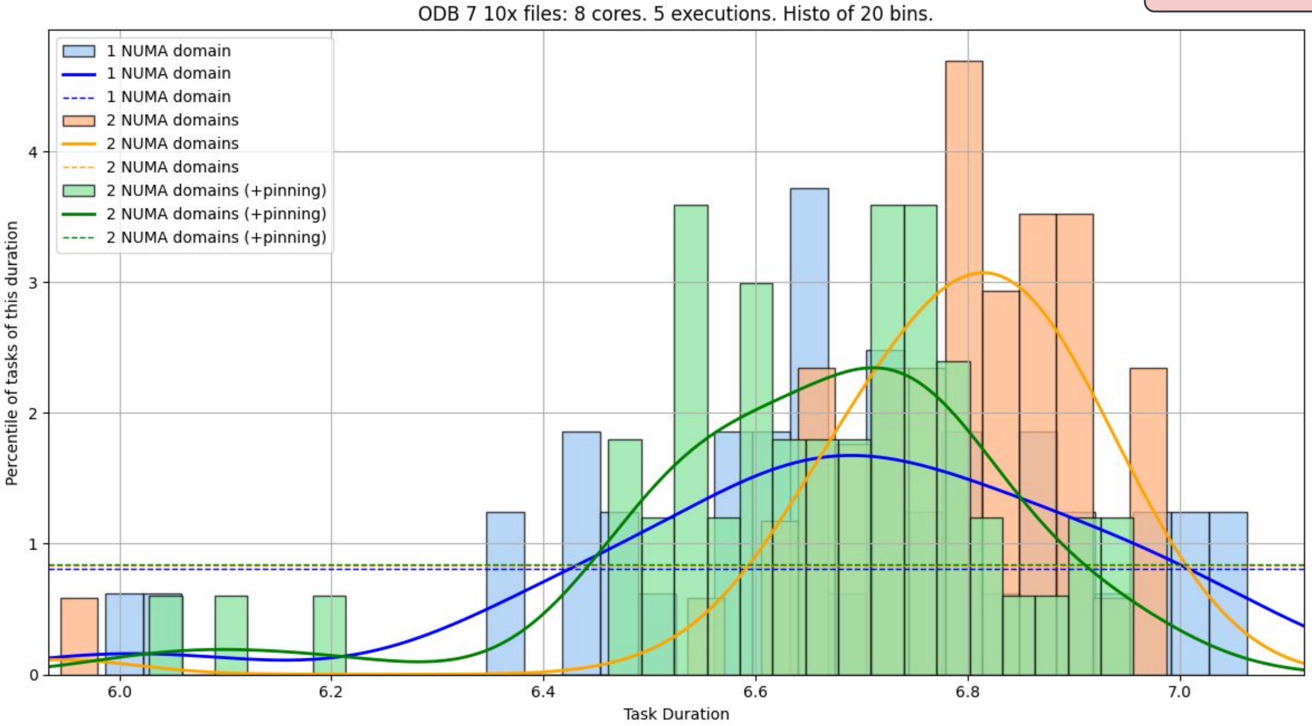


AMD EPYC 7302 16-Core Processor



NUMA Task duration distributions

AMD EPYC 7302 16-Core Processor



TNUMAExecutor (old solution)

- Spawn a process for each NUMA domain
- Use numa.h to pin each process to a different NUMA domain
- All threads spawned by a process will be restricted to the same domain
- Execute in each thread (isolated per process)
- Collect results (first from threads, then from processes)
- **Problems:**
    - MP+MT unsafe?
    - hard to benchmark: numa.h overwrites the core mask specified by taskset or numactl