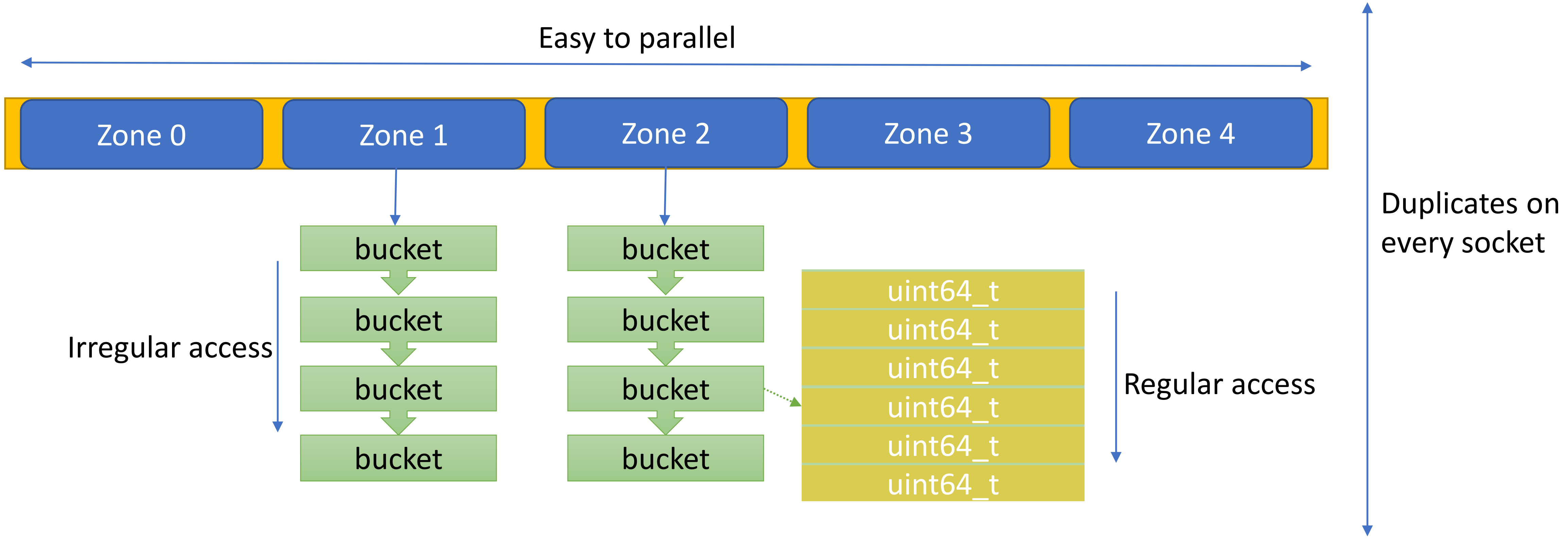
Mini benchmark – things to consider

- Cache: impact the performance across multiple runs
- CPU pipeline: memory latency might be hidden by the hardware prediction (prefetch).
- Compiler Optimization: some operations might not be actually executed if workload is too trivial
- NUMA effect: workload should always local to threads

Workload design

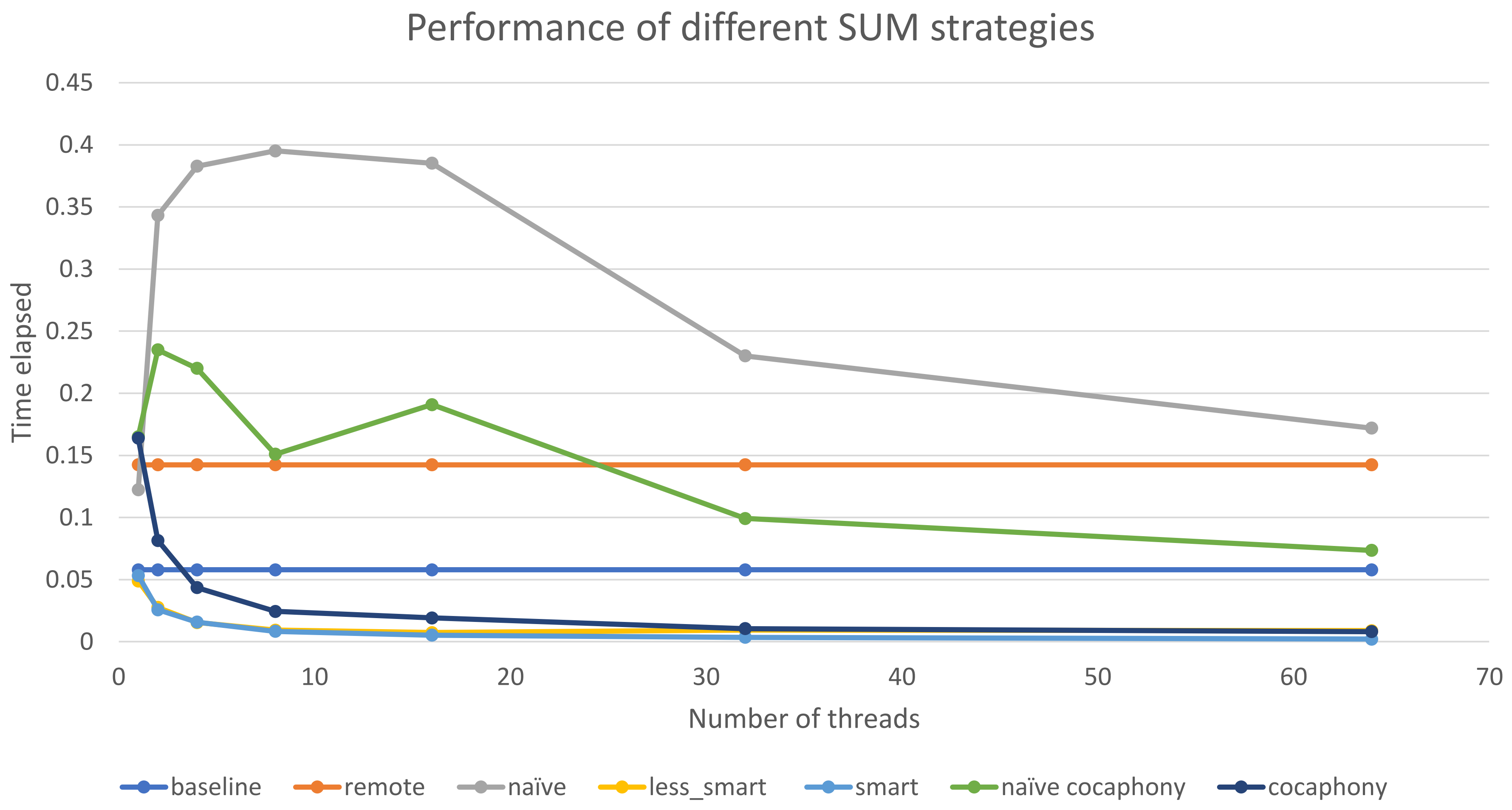


Workload goal: to sum all the uint64_t

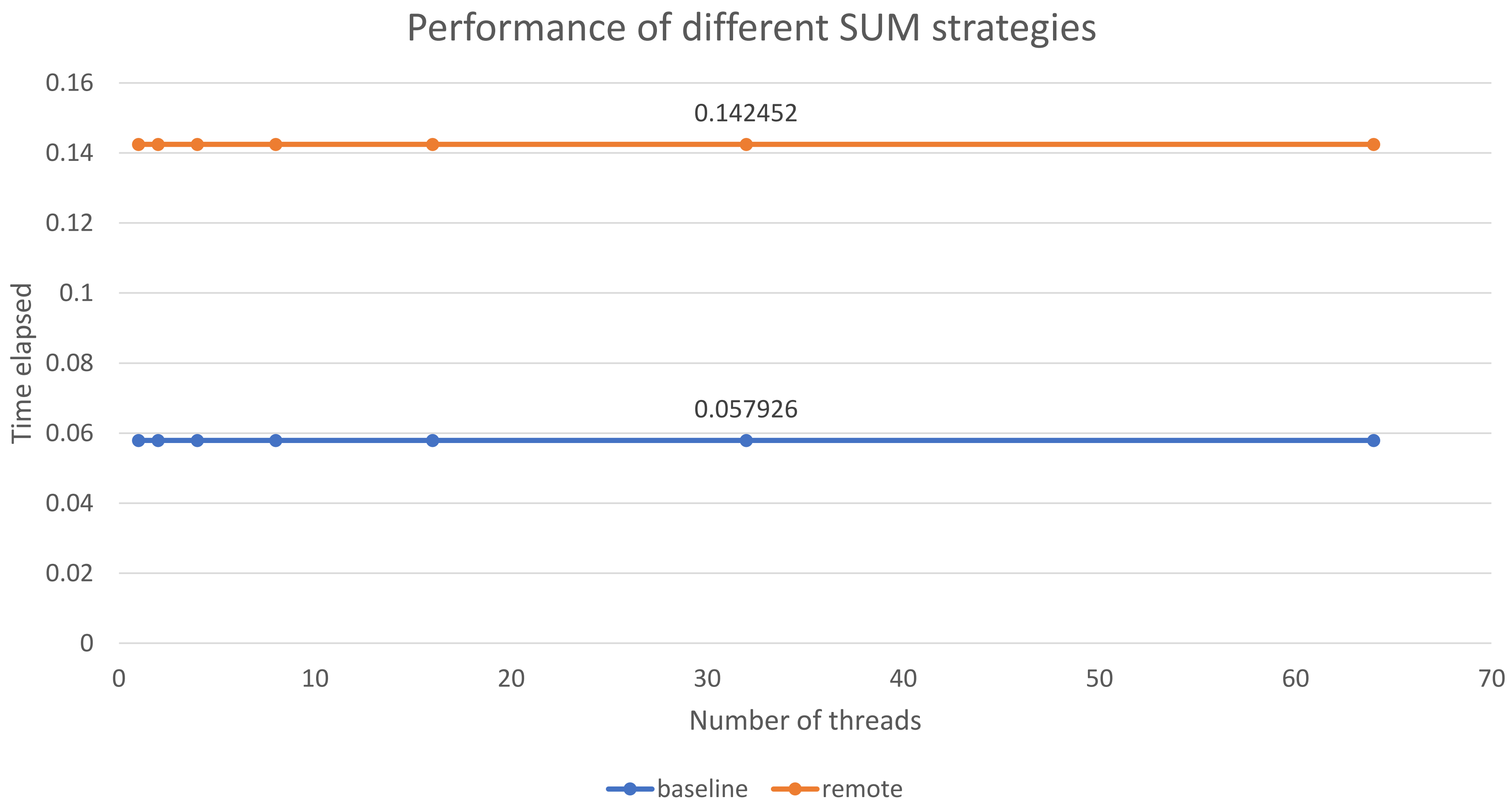
Implementation Details

- All the experiments are repeated at least 3 times, the reported results are the average of them.
- Cache lines are evicted (using clflush) from all sockets across different runs.
- Thread creation/joining time not included, all threads are guaranteed to start at the same time.
- Each thread will only access its socket-local workload, in other words, no cross-socket memory access (except for the global sum, if applicable)

Results - Overall



Results – Baseline & NUMA effect

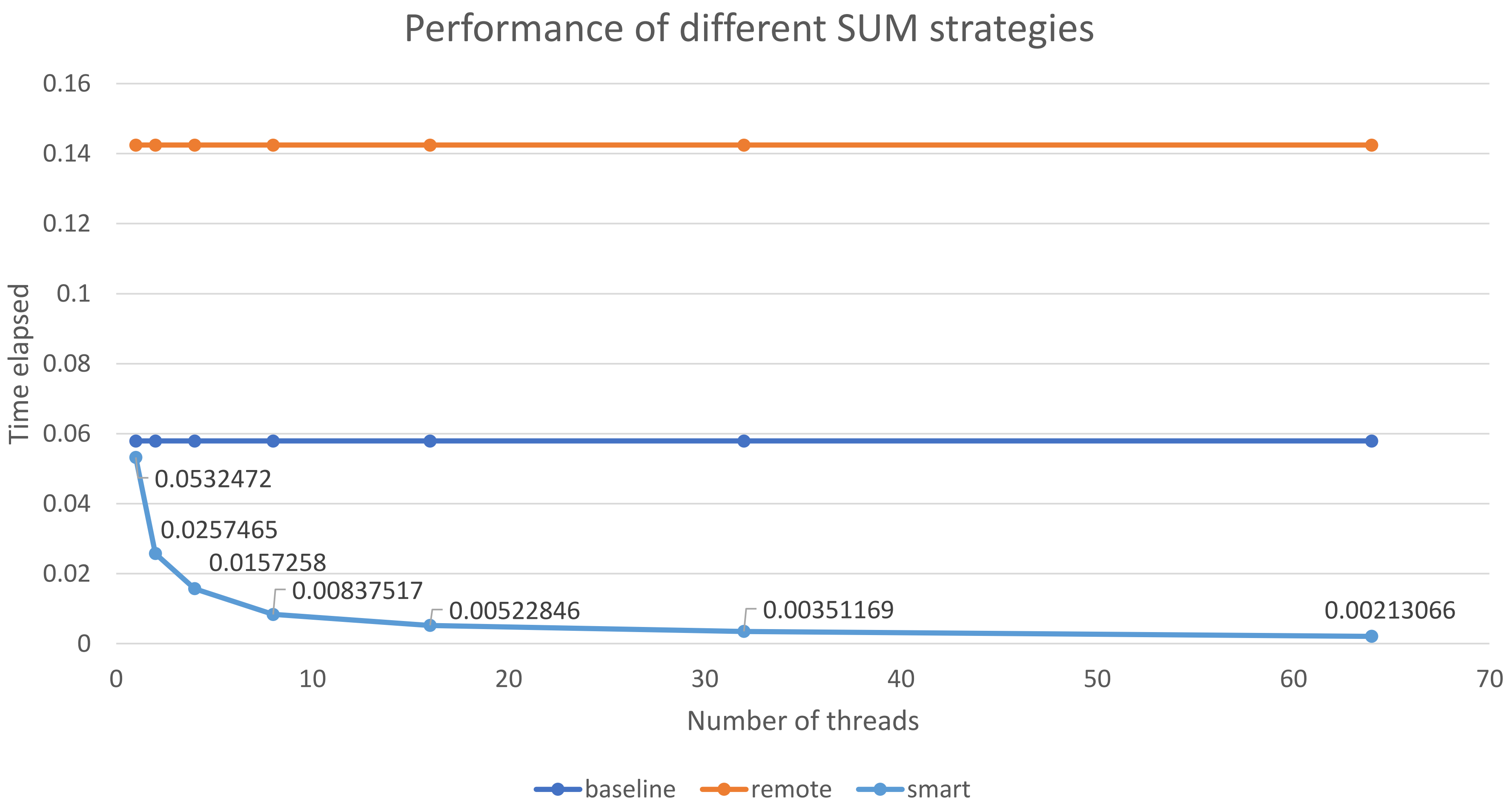


Baseline: single thread, local workload

Remote: single thread, remote workload

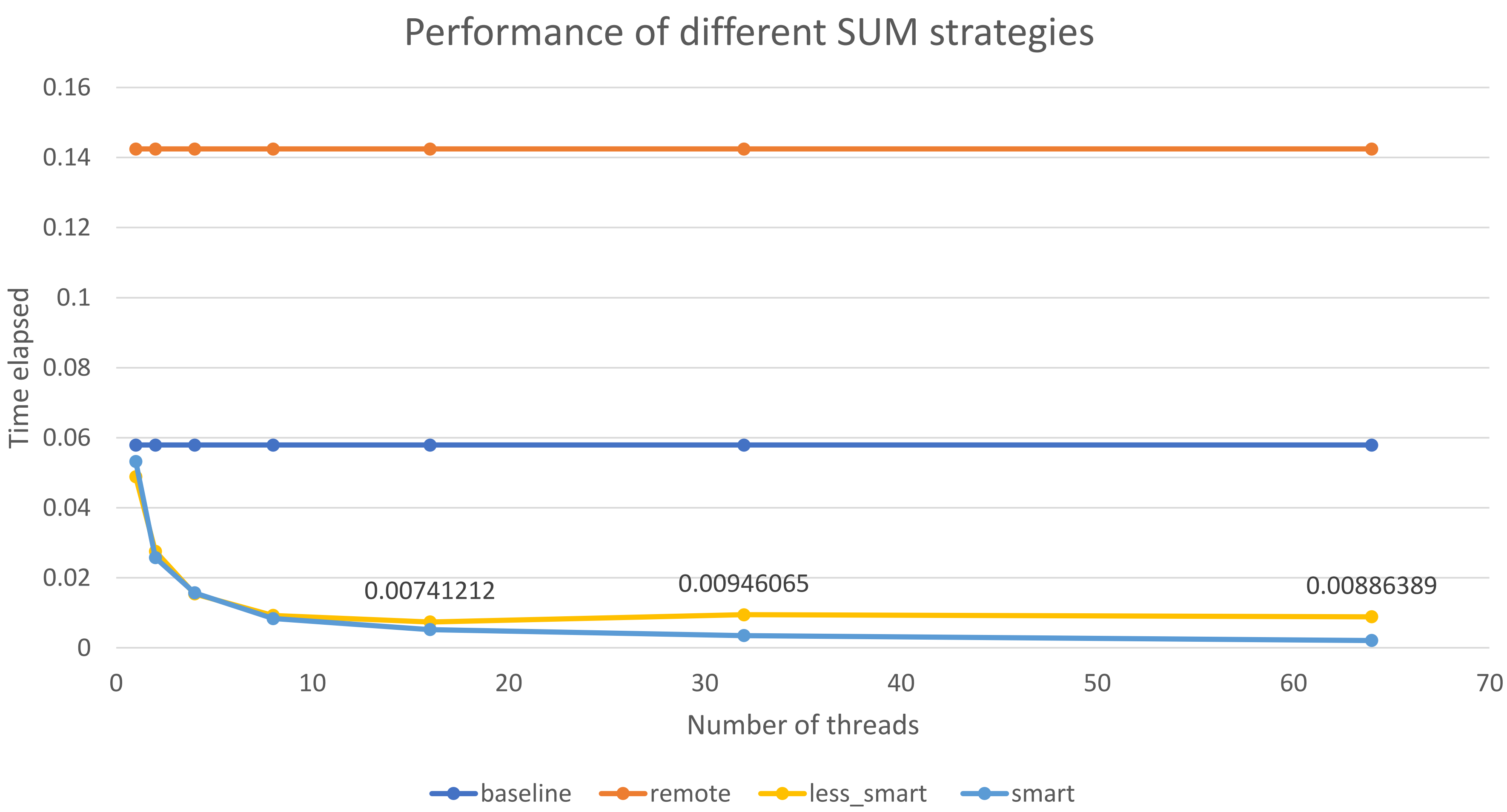
Local access is about 2.4 times faster than remote access

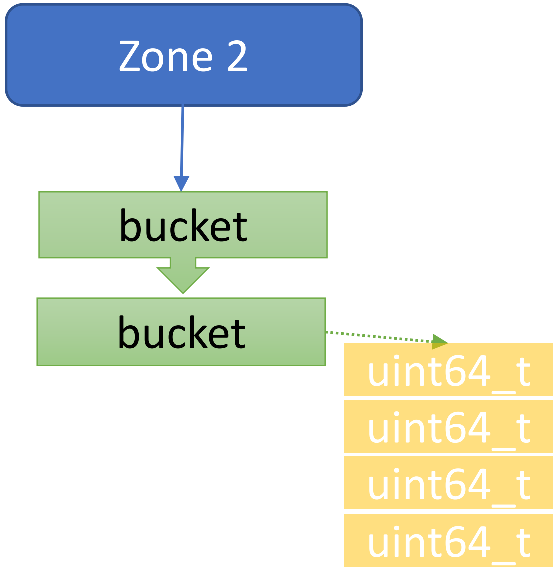
Results – Best Possible



Smart: each thread commits to its local variable, only sync with global sum once

Results – Less Smart

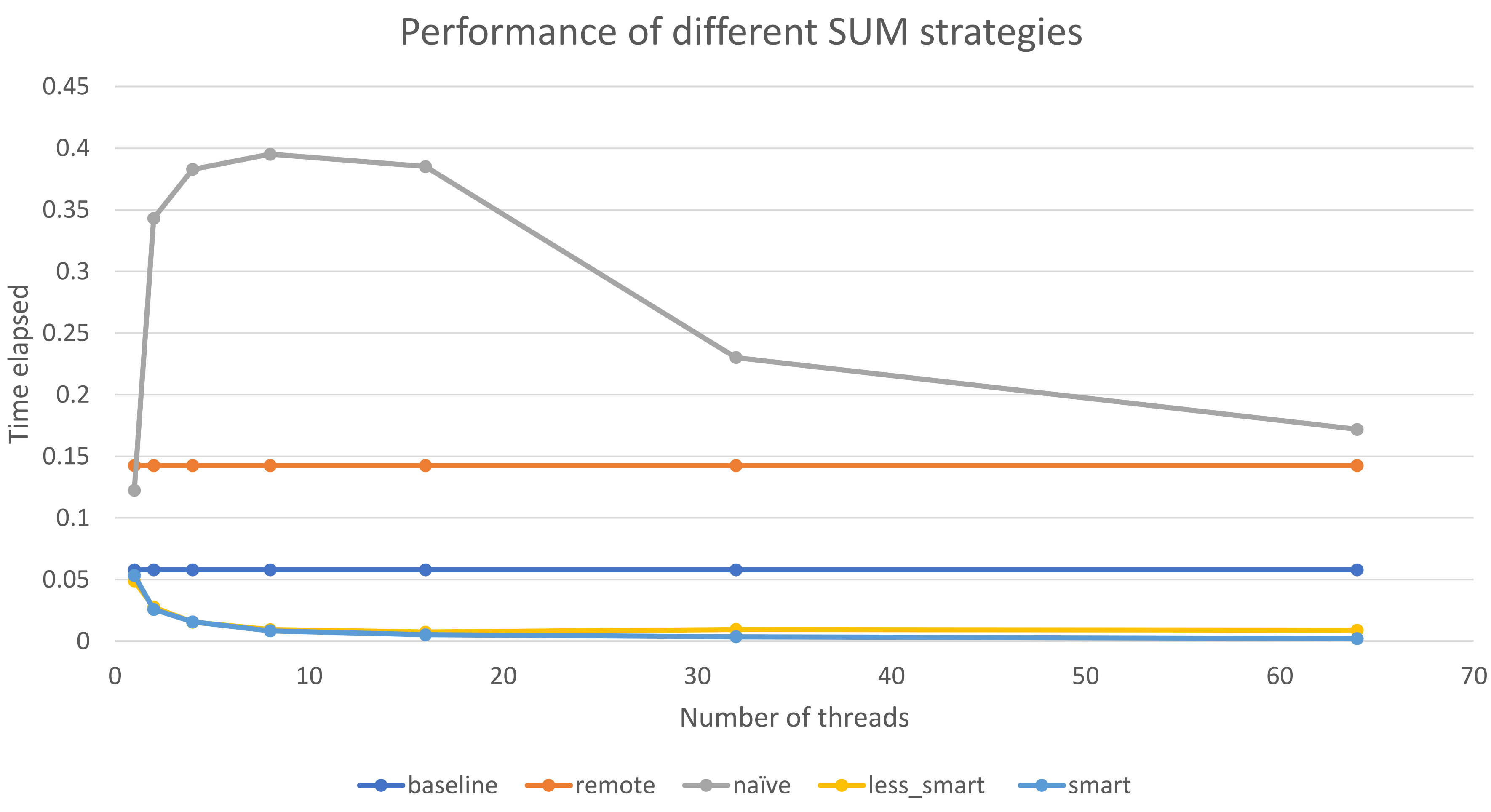




Less_smart: threads commit to global sum after summing each bucket

Throughput slightly decreases as contention goes high.

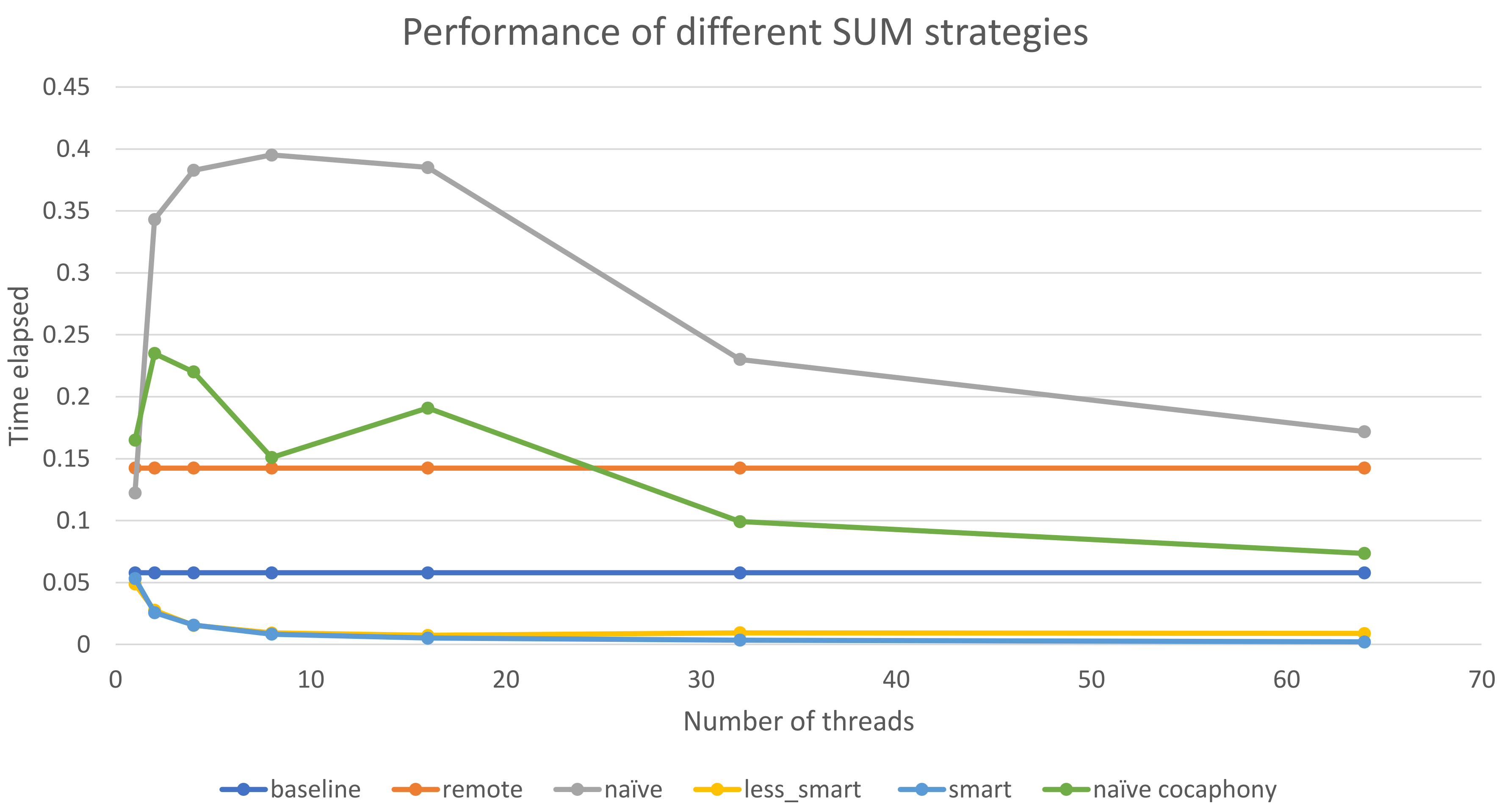
Results – Naïve



Naive: threads directly add to the global sum

Combination of cache coherence cost and NUMA effects

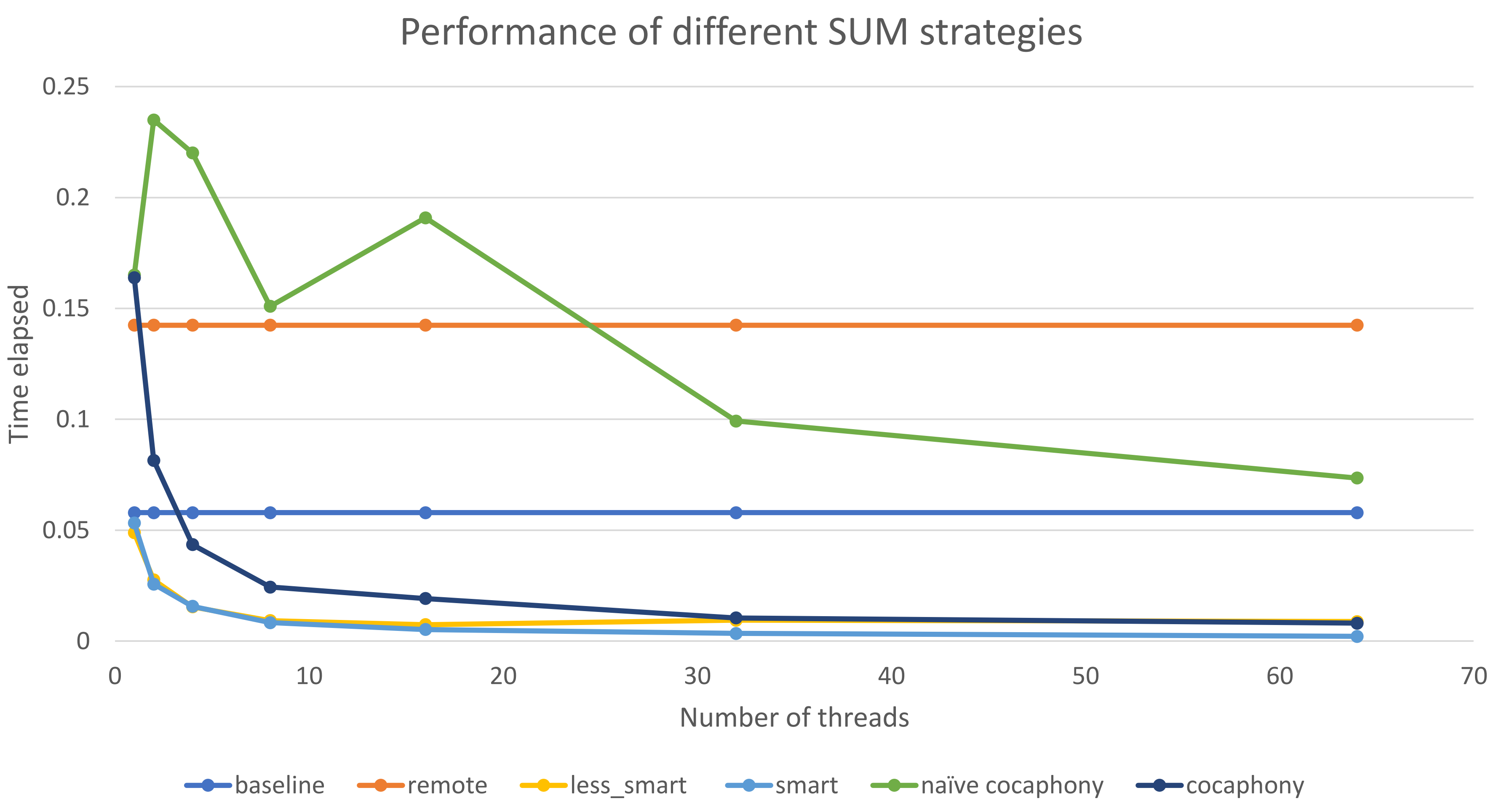
Results – Naïve cacophony



Naïve cacophony: threads adds to its core-local copy, and sync with global sum once.

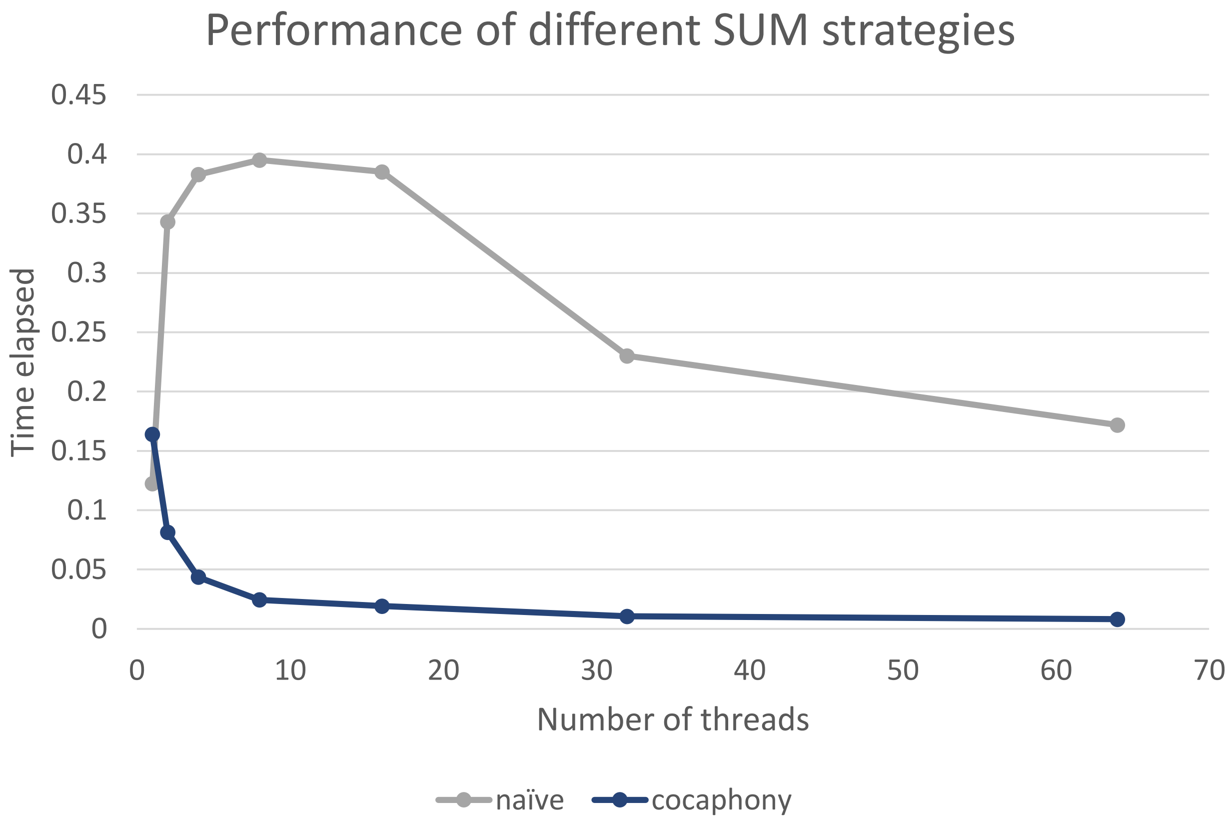
False sharing without NUMA effects.

Results – Smart cacophony



Smart cacophony: threads adds to its core-local copy, but each copy is padded to cache line size to prevent false sharing

Results – Programmability



Cacophony

```
for (uint32_t k = 0; k < zones_per_thread; k += 1) {
  auto curr_zone = workload->zones[zones_per_thread * thread_idx + k];
  auto ptr = curr_zone.start;
  while (ptr->next != nullptr) {
    for (uint32_t i = 0; i < ptr->data.size(); i += 1) {
      op.IncValue(ptr->data[i]);
    }
    ptr = ptr->next;
  }
}
```

Naïve

```
for (uint32_t k = 0; k < zones_per_thread; k += 1) {
  auto curr_zone = workload->zones[zones_per_thread * thread_idx + k];
  auto ptr = curr_zone.start;
  while (ptr->next != nullptr) {
    for (uint32_t i = 0; i < ptr->data.size(); i += 1) {
      shared_rv.fetch_add(ptr->data[i]);
    }
    ptr = ptr->next;
  }
}
```