A FORAY INTO THREAD-PER-CORE ARCHITECTURE

FRIDGESEAL

OR “I BOUGHT THE WHOLE CPU, SO I’LL USE THE WHOLE CPU” - MAKING YOUR PROGRAM WORK FAST AND DO OTHER THINGS FAST TOO

SETTING THE SCENE

- Lots of applications, especially those with a reasonable amount of complexity likely already have a number of parallel and concurrent tasks that need to be run as part of the application flow

SETTING THE SCENE

- Lots of applications, especially those with a reasonable amount of complexity likely already have a number of parallel and concurrent tasks that need to be run as part of the application flow
- Core counts are continuing to go up on devices

SETTING THE SCENE

- Lots of applications, especially those with a reasonable amount of complexity likely already have a number of parallel and concurrent tasks that need to be run as part of the application flow
- Core counts are continuing to go up on devices
- Rust is already nice and speedy, and has some amazing benefits for parallel programming, is there anything we can do to push this further?

SETTING THE SCENE

- Lots of applications, especially those with a reasonable amount of complexity likely already have a number of parallel and concurrent tasks that need to be run as part of the application flow
- Core counts are continuing to go up on devices
- CPU's are incredibly fast already, and the performance gap between CPU and NVMe storage devices is notably smaller than HDD's and even older SSD's
- Rust is already nice and speedy, and has some amazing benefits for parallel programming, is there anything we can do to push this further?

SETTING THE SCENE

INTRODUCING THREAD-PER-CORE PROGRAMMING

We want to equip each thread with the ability to do as much work as possible, without needing to co-ordinate or wait for locks.

INTRODUCING THREAD-PER-CORE PROGRAMMING

INTRODUCING THREAD-PER-CORE PROGRAMMING

If we can pin a thread to a specific CPU, and shard the data we work on, so that work is self contained, we can reduce the need for locks to almost zero, and exploit both data and instruction caches

INTRODUCING THREAD-PER-CORE PROGRAMMING

If we're not shuffling data around cores, and we're thrashing instruction/data caches because of context switched, etc, we can instead spend that time doing valuable work, which is beneficial for both throughout, and latency

INTRODUCING THREAD-PER-CORE PROGRAMMING

- Some 2019 research demonstrated a 71% reduction in tail latencies when using thread-per-core (TPC) architecture:
- https://helda.helsinki.fi/bitstream/handle/10138/313642/tpc_ancs19.pdf
- ScyllaDB, uses TPC architecture via the C++ library Seastar, for significant performance improvements over Cassandra
- RedPanda is a Kafka-compatible replacement that uses TPC and offers improved end-to-end data latencies, more hardware efficiency and more

THAT SOUNDS GREAT!
HOW DO WE GET STARTED?

INTRODUCING THREAD-PER-CORE PROGRAMMING

- First, go to your nearest Spotlight and pick out your preferred colour of woollen thread for your CPU…

ENTER: GLOMMIO

- TPC framework in Rust + more!
- Io_Uring support with dedicated rings per core, as well as dedicated latency vs throughput rings
- Direct IO support for compatible NVMe drives
- Control-theory inspired schedulers, enabling easy way to statically and dynamically balance resources between latency-sensitive and insensitive applications
- Each thread gets its own Local Executor, supporting async execution, which means futures no longer need be `Send`

PUTTING IT ALL TOGETHER

TARKINE

TARKINE

- “Percolate” style full-text-search queries:
- queries are persisted, text documents are streamed through instead
- search results are asynchronous from the perspective of the user
- useful if you want to search a lot of data and know what you’re looking for, or you need to re-run the query lots and only care about new results, etc

- Makes a nice fit for TPC architecture + glommio feature set:
- threads have little need of communication other than receiving work
- we care about maximum utilisation of all resources, because throughput of documents is paramount
- Io_uring and direct IO for NVMe devices is a nice benefit - we'd like the results to be written as fast as we can feed them to the drive, and blocking would reduce time spent searching new docs!

```rust
use tokio::sync::broadcast;
use glommio::executor::LocalExecutorBuilder;

fn main() -> Result<(), Box<dyn error::Error>> {
  let core_count = std::thread::available_parallelism()?;
  let indexer_thread_limit = 12;
  let bind_addr = SocketAddr::from(([127, 0, 0, 1], 8765));
  let (doc_stream_channel, mut doc_recv) = broadcast::channel(5000);


  // Spin up our tokio server
  let server_thread_handle = std::thread::spawn(move ||
    server_runtime(doc_stream_channel, bind_addr));

  // Spin up our glommio-based-workers
  let indexer_handles = for cpu_id in 1..indexer__thread_limit {
    LocalExecutorBuilder::new(Placement::Fixed(cpu_id)).spawn(||
      indexing_runtime(doc_recv.clone(), ...))?;
   };

  server_thread_handle.join()?;
  for h in indexer_handles {
    h.join()?;
  }
```

```rust
async fn indexing_runtime(
    query_stream: spsc_queue::Consumer<PersistentQuery>,
    doc_stream: spsc_queue::Consumer<TextSource>,
) {
    println!(
        "Starting up resources on executor id: {}",
        glommio::executor().id()
    );
    let search = search::Searcher::new();
    let mut query_processor = QueryProcessor::new(query_stream, doc_stream);
    let mut indexing_results = Vec::with_capacity(5000);
    loop {
        // Grab all the queries submitted since the last loop
        while let Some(new_query) = query_processor.incoming_queries.try_pop() {
            query_processor.queries.store_query(new_query);
        }
        // Stream a single document through all this shards queries
        if let Some(document) = query_processor.incoming_docs.try_pop() {
            query_processor.run_queries(&search, document, &mut indexing_results);
        } else {
            continue;
        };
        if indexing_results.is_empty() {
            continue;
        }
        // Push data about matches into storage (on disk in this case)
        for index_data in indexing_results.drain(0..) {
            glommio::spawn_local(async move {
                let path = format!(
                    "output_data/{}.rkyv",
                    index_data.document_id
                );
                let output_path = Path::new(&path);
                let mut sink = glommio::io::ImmutableFileBuilder::new(output_path)
                    .build_sink()
                    .await
                    .unwrap();
                let index_buffer = rkyv::to_bytes::<_, 1024>(&index_data).unwrap();
                sink.write(&index_buffer.as_slice())
                    .await
                    .expect("Couldn't write buffer");
                sink.seal().await.expect("Couldn't close seal");
            })
            .await;
        }
    }
}
```

QUESTIONS?

THANK YOU FOR LISTENING!