Home
Architecture »
Blog
Seastar Applications
Documentation
Frequently Asked Questions
ScyllaDB

Shared-nothing Design

Hardware on which modern workloads must run is remarkably different from the hardware on which current programming paradigms depend, and for which current software infrastructure is designed.

Core Counts Grow, Clock Speeds Stay Constant

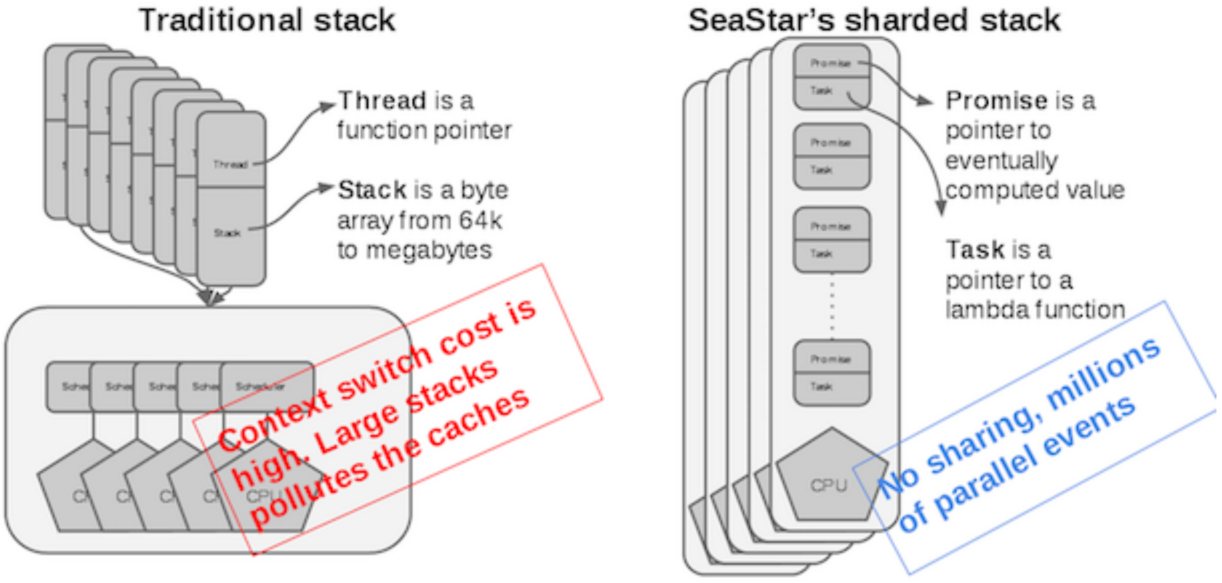
Performance increases in clock speeds of individual cores have stopped. The increase in number of cores means that performance depends on coordination across multiple cores, no longer on throughput of a single core.

On new hardware, the performance of standard workloads depends more on locking and coordination across cores than on performance of an individual core. Software architects face two unattractive alternatives: coarse-grained locking, which will see application threads contend for control of the data and wait instead of producing useful work, and fine-grained locking, which, in addition to being hard to program and debug, sees significant overhead even when no contention occurs, due to the locking primitives themselves.

Meanwhile, I/O Continues to Increase in Speed

The networking and storage devices available on a modern system have also continued to grow in speed. However, CPU cores the ability to process packets on any one core has not.

A 2GHz processor handling 1024-byte packets at wire speed on a 10GBps network has only 1670 clock cycles per packet. (source: Intel DPDK Overview)


Traditional stack
Thread is a function pointer
Stack is a byte array from 64k to megabytes
Thread
Stack
Scheduler
CPU
Context switch cost is high. Large stacks pollutes the caches
SeaStar's sharded stack
Promise
Task
Promise is a pointer to eventually computed value
Task is a pointer to a lambda function
CPU
No sharing, millions of parallel events

The Seastar Model: Shared-nothing

Because sharing of information across cores requires costly locking, Seastar uses a shared-nothing model that shards all requests onto individual cores.

Seastar runs one application thread per core, and depends on explicit message passing, not shared memory between threads. This design avoids slow, unscalable lock primitives and cache bounces.

Any sharing of resources across cores must be handled explicitly. For example, when two requests are part of the same session, and two CPUs each get a request that depends on the same session state, one CPU must explicitly forward the request to the other. Either CPU may handle either response. Seastar provides facilities that limit the need for cross-core communication, but when communication is inevitable, it provides high performance non-blocking communication primitives to ensure performance is not degraded.

Explicit Communication Between Cores

Seastar provides several related functions for communicating between cores. The simplest is:

```
smp::submit_to(cpu, lambda)
```

This is a promise. It returns a future, which is the return value of the lambda. It runs the lambda on the specified cpu and returns a result. For example:

```
smp::submit_to(neighbor, [key] {
        return local_database[key];
}).then([key, neighbor] (sstring value) {
        print("The value of key %s on shard %d is %s\n", key, neighbor, value);
});
```

The equivalent in a threaded environment requires taking a lock around the database object. The locking operation is inherently expensive and can also force a context switch or waste CPU cycles in spinning, depending on the locking scheme used.

Other variants of cross-core communication allow for broadcast of a value to all CPUs, or a map/reduce operation that sends a lambda to all CPUs, collects results, and applies a transformation to reduce to a single value.