Designing an ultra low-overhead multithreading runtime for Nim

Mamy Ratsimbazafy
mamy@numforge.co

Weave
https://github.com/mratsim/weave

Hello!

I am Mamy Ratsimbazafy

During the day blockchain/Ethereum 2 developer (in Nim)

During the night, deep learning and numerical computing developer (in Nim) and data scientist (in Python)

You can contact me at mamy@numforge.co

Github: mratsim

Twitter: m_ratsim

Where did this talk came from?

- 3 years ago: started writing a tensor library in Nim.
- 2 threading APIs at the time: OpenMP and simple threadpool
- 1 year ago: complete refactoring of the internals

Agenda

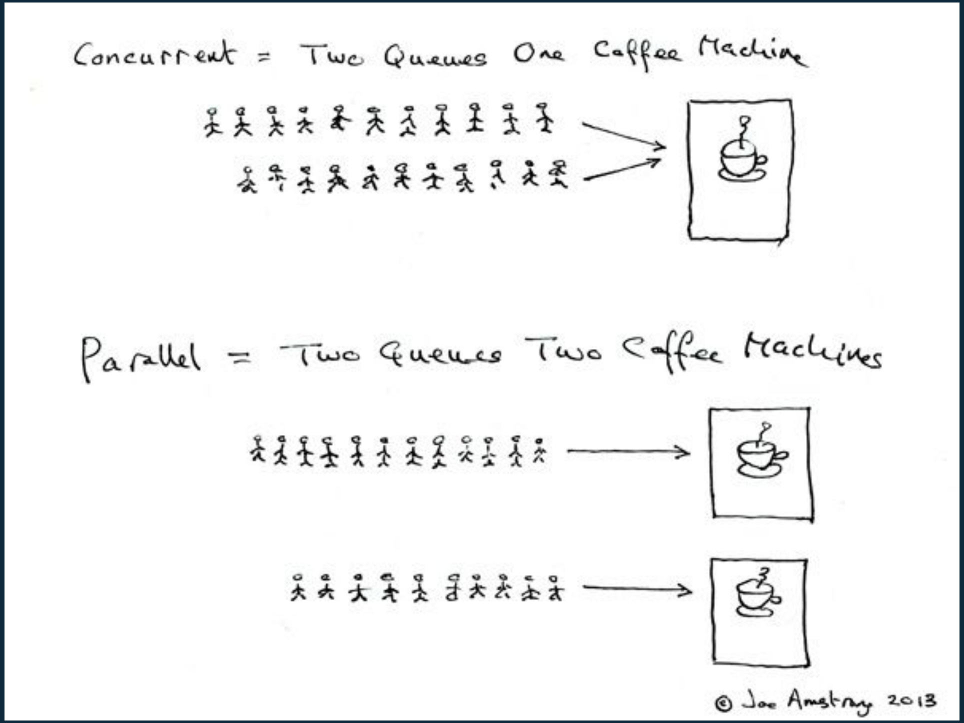
- Understanding the design space
- Hardware and software multithreading: definitions and use-cases
- Parallel APIs
- Sources of overhead and runtime design
- Minimum viable runtime plan in a weekend

1

Understanding the design space

Concurrency vs parallelism, latency vs throughput
Cooperative vs preemptive, IO vs CPU

Parallelism is not concurrency



Kernel threading models

1:1 Threading

1 application thread -> 1 hardware thread

N:1 Threading

N application threads -> 1 hardware thread

M:N Threading

M application threads -> N hardware threads

The same distinctions can be done at a multithreaded language or multithreading runtime level.

The problem

How to schedule M tasks on N hardware threads?

Latency vs Throughput

- Do we want to do all the work in a minimal amount of time?
  - Numerical computing
  - Machine learning
  - ...

- Do we want to be fair?
  - Clients-server
  - Video decoding
  - ...

Cooperative vs Preemptive

Cooperative multithreading:

- Coroutines, fibers, green threads, first-class continuations
- Userland, lightweight context switches
- Cannot use hardware threads

Preemptive:

- PThreads (OpenMP, TBB, Cilk, ...)
- Scheduled by the OS, heavier context switches
- Need synchronization primitives:
    - Locks
    - Atomics
    - Transactional memory
    - Message-passing

IO-tasks vs CPU-tasks

IO-tasks:

- Latency optimized
- async/await

CPU-tasks:

- Throughput optimized
- spawn/sync

Doing both in the same runtime is complex:

- Different skills
- Different OS APIs (kqueue, epoll, IOCP vs PThreads, Windows Fiber)
- Different requirements
- Same public APIs/data-structure (async/spawn await/sync, Task, Future)

Focus of the talk

- CPU-tasks
- Throughput optimized
- Preemptive scheduling

2

1001 forms of multithreading

Hardware vs Software multithreading
Data parallelism, Task parallelism, Dataflow parallelism

Hardware-level multithreading

ILP - Instruction-level Parallelism

1 CPU, multiple “execution ports”

SIMD - Single Instruction Multiple Data

a.k.a. Vector instructions (SSE, AVX, Neon)

SIMT - Single Instruction Multiple Thread

GPUs (Warp for Nvidia, Wavefront for AMD)

SIMT - Simultaneous Multithreading

Hyperthreading (2x logical siblings core usually, 4x on Xeon Phi)

Share execution ports, memory bus, caches, ...

Data parallelism

Parallel for loop

- Same instructions on multiple data

- OpenMP

- Use-cases
  - Vectors, matrices, multi-dimensional arrays and tensors

- Challenges:
  - Nested parallelism
  - Splitting the loop
    - Static splitting
    - Eager binary splitting
    - Lazy tree splitting

Task parallelism

spawn/sync

- “Function call” that may be scheduled on another hardware threads

- Intel TBB (Threads Building Blocks), OpenMP Tasks (since 3.0)

- Use-cases
  - Anywhere you want a parallel function call
  - Parallel tree algorithms, divide-and-conquer, …

- Challenges:
  - API: futures? (in Nim “Flowvar” to distinguish from IO-tasks futures)
  - Synchronization
  - Scheduling overhead
  - Thread-safe memory management

Dataflow parallelism

- Alternative names
  - Pipeline parallelism
  - Graph parallelism
  - Stream parallelism
  - Data-driven task parallelism

- OpenMP Tasks with depends “in”, “out”, “inout” clauses
- Intel TBB Flowgraph

- Use-cases: expressing precise data dependencies (beyond barriers)

  For example: frame processing in a video encoding pipeline.

- Challenges: API, thread-safe data structure for dependency graphs

3

Parallel APIs

Task parallelism

Copy IO-task API “async/await” with different keywords

- async/await => spawn/sync
- Future => Flowvar

Why:

- Reuse knowledge from async/await which is actually applicable
- Different keywords to expose different requirements

Synchronization:

- Channels / Shared memory for data
- Dataflow parallelism for dependency
  - Or Barriers with “async/finish” model of Habanero Java
  - OpenMP barriers do not work with task parallelism (taskwait instead).

Data parallelism

Parallel for loop

- Start, stop, step (stride)
- Abstraction detail if non-lazy splitting:
  - “Grain size”

Why:

- Easier to port decades of OpenMP scientific code

Synchronization:

- Shared memory for data
- Barriers (if not built on top of task parallelism)
- Dataflow parallelism for fine-grained dependencies

Dataflow parallelism

No established API

1. Declarative: depends clause in/out/inout

   => OpenMP

   Requires a thread-safe hash-table

2. Imperative: pass a “ready” handle between the data producer and the consumer(s).

   => Strategy used in Weave, the handle is called a Pledge (~Promises with adapted semantics)

   Can be implemented with broadcasting SPMC queues

4

Sources of overhead And "Implementation details"

Characterizing performance of a runtime

Scheduling overhead

Context switching is costly

Context switching to the kernel (syscall, creating threads) is very costly

- At least 200 cycles: 200 additions
  - 3GHz = 1 cycle every 0.33 ns
  - 1 us = 3000 cycles
  - 1 ms = 3 000 000 cycles

- https://gist.github.com/jboner/2841832

  “Latency Numbers Every Programmer Should Know”

Don’t create/destroy threads, use a threadpool and have threads sleep

Memory overhead

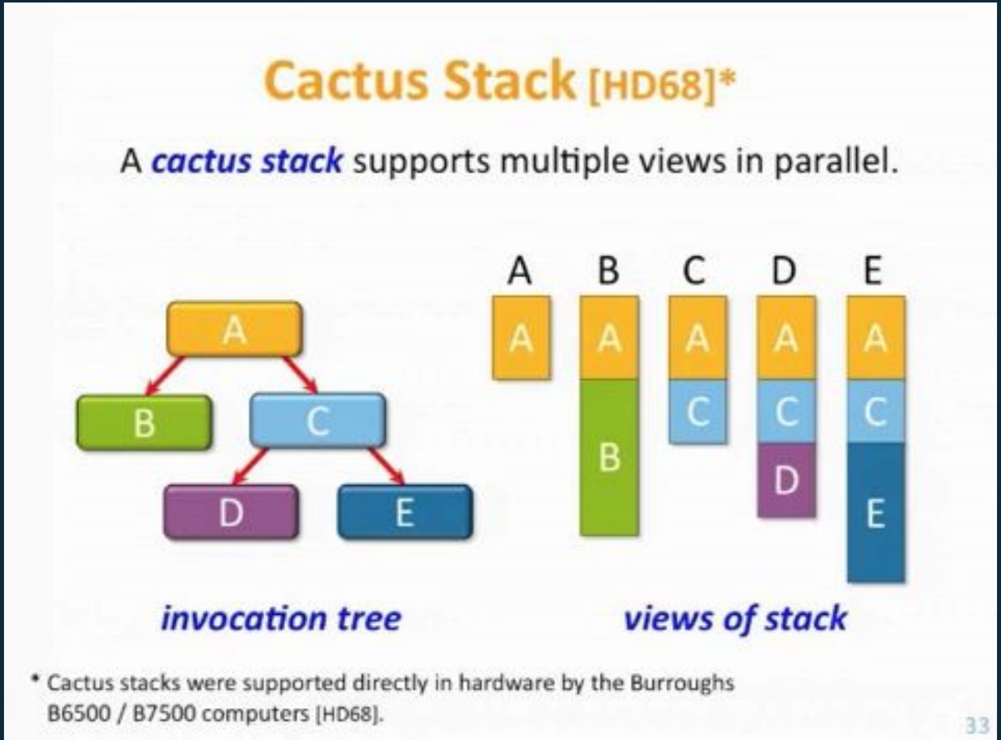
Task parallelism might generates billions or trillions of tasks and futures

- **Access from multiple threads:**
  - **Heap allocation**
  - **Threadsafe allocation/deallocation**

- **Challenges**
  - **Large number of tasks (fibonacci)**
  - **Producer-Consumer workloads**

  Lead to task cache imbalance

Memory overhead



Credits: Angelina Lee

Memory overhead

Zoom on cactus stacks / segmented stacks

https://github.com/mratsim/weave/blob/v0.3.0/weave/memory/multithreaded_memory_management.md

- **Plagued Go and Rust (abandoned)**
- **Decades of research including OS kernel forks, mmap changes**
  - A cactus stack is a memory abstraction
  - That deals with thread memory/variable concurrent views
  - Challenges:
    - heap fragmentation
    - serial/parallel reciprocity / calling convention
    - Scalability (TBB is depth-restricted and does not scale on certain workloads)
- **Practical solutions for passing task inputs**
  - **coroutines/continuation (save/restore a "task frame")**
  - **capturing inputs by value and saving in the task**

Load Balancing

Simple threadpool

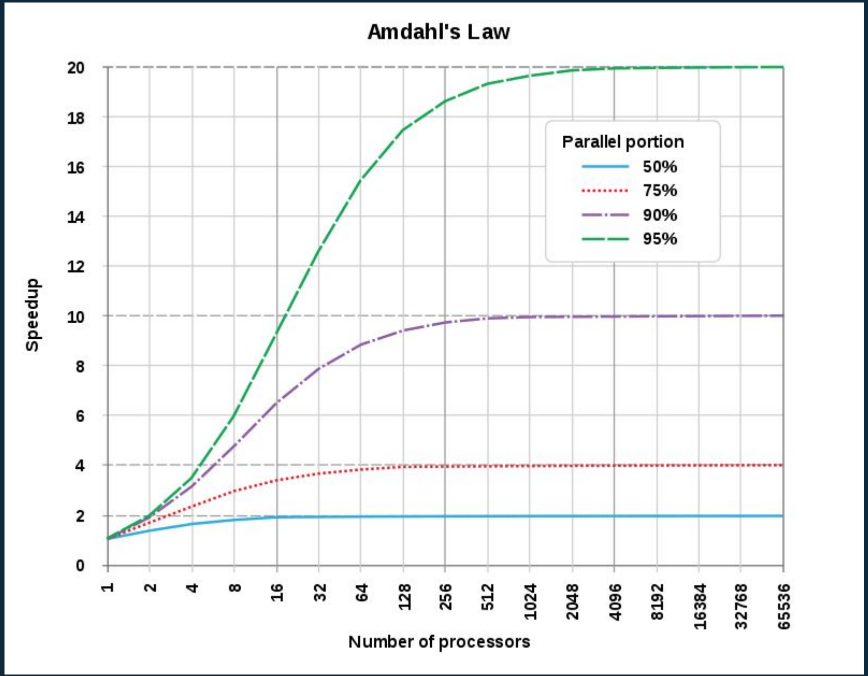
- One global task queue
- Dispatch task to a ready thread

=> Contention

The best way to scale a parallel program is to share nothing

Load Balancing

Amdahl's Law



Load Balancing

Sources of serialization

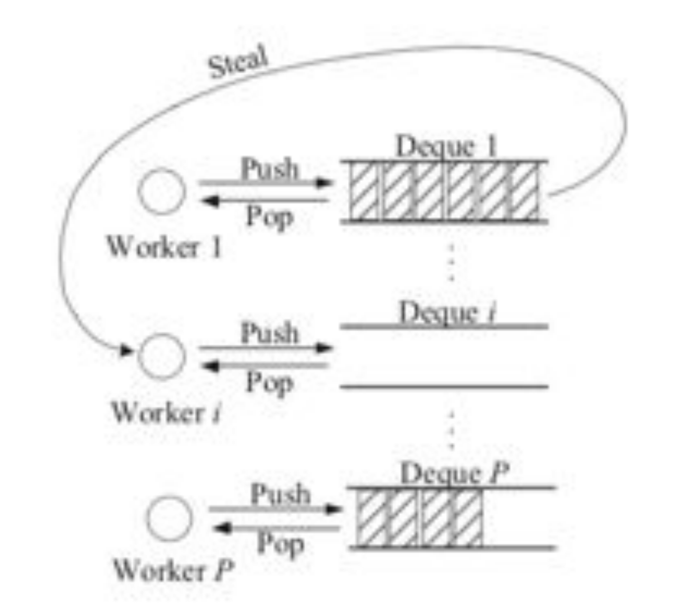
- Shared memory access (be it locks or atomics)
- Single task queue
- Single memory pool

=> Distribute on N threads

Load Balancing

Work-stealing

Image credits: Yangjie Cao



Load Balancing

Work-stealing

1 deque per worker

- Enqueue locally created tasks at the head
- Dequeue tasks at the head
  - Improve locality
- Steal in other workers from the tail
  - Synchronization only on empty deque
- Mathematical proof of optimality
- Papers (including C/C++ implementation and proof)
  - Chase, Lev
  - Arora, Blumofe and Plaxton (non-blocking)
  - Lê, Pop, Cohen, Nardelli (weak memory models)

Alternative: Parallel Depth-First Scheduling (Julia), steal from the head.

Parenthesis on memory models

Memory models:

- **The semantics of threads reading and writing the same memory location**
- **Specification of "happens-before" relationship**
  - Disable compiler reordering
  - Forces memory invalidation at the hardware level
- **Goal: have a lock-less program be sequentially consistent**
- "Relaxed", "Acquire", "Release", "Acquire-Release", "Sequentially Consistent" atomics

- C++11 is dominant (used in Rust, Nim, …).

  Watch Herb Sutter talk "atomic<> Weapons: The C++ Memory Model and Modern Hardware"

  https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

Load Balancing

Adaptative work-stealing

- Steal-one strategy
- Steal-half strategy
- Adaptative

Public vs Private vs Hybrid deques

- Public deques are constrained by push/pop/steal/steal-half
  - Steal requests are implicit and have very low-overhead
  - Thieves can check if a victim deque is empty
  - They don’t work in a distributed setting
- Private deques can implement very complex strategies
  - Steal requests are explicit data structure like tasks
  - Thieves are “blind”
  - They work in distributed settings

5

Work-stealing runtime
In a weekend

Minimal viable runtime

Task data structure

- Function pointer + blob for task inputs or a closure
- start/stop/step (for data parallelism)
- prev/next field for intrusive queues/deques
- Future pointer

Work-stealing deque

- head/tail
- pushFirst
- popFirst
- stealLast

API

- **init**
- **exit**
- **spawn/sync**

References

Weave design

- https://github.com/mratsim/weave (several markdown design files)
  - https://github.com/mratsim/weave/tree/v0.3.0/benchmarks
  - https://github.com/mratsim/weave/tree/v0.3.0/weave/memory
- RFC: https://github.com/nim-lang/RFCs/issues/160

Research

- https://github.com/numforge/laser/blob/master/research/runtime_threads_tasks_allocation_NUMA.md
  - Runtimes, NUMA, CPU+GPU computing, distributed computing

Designing an ultra low-overhead multithreading runtime for Nim

Mamy Ratsimbazafy
mamy@numforge.co

Weave
https://github.com/mratsim/weave