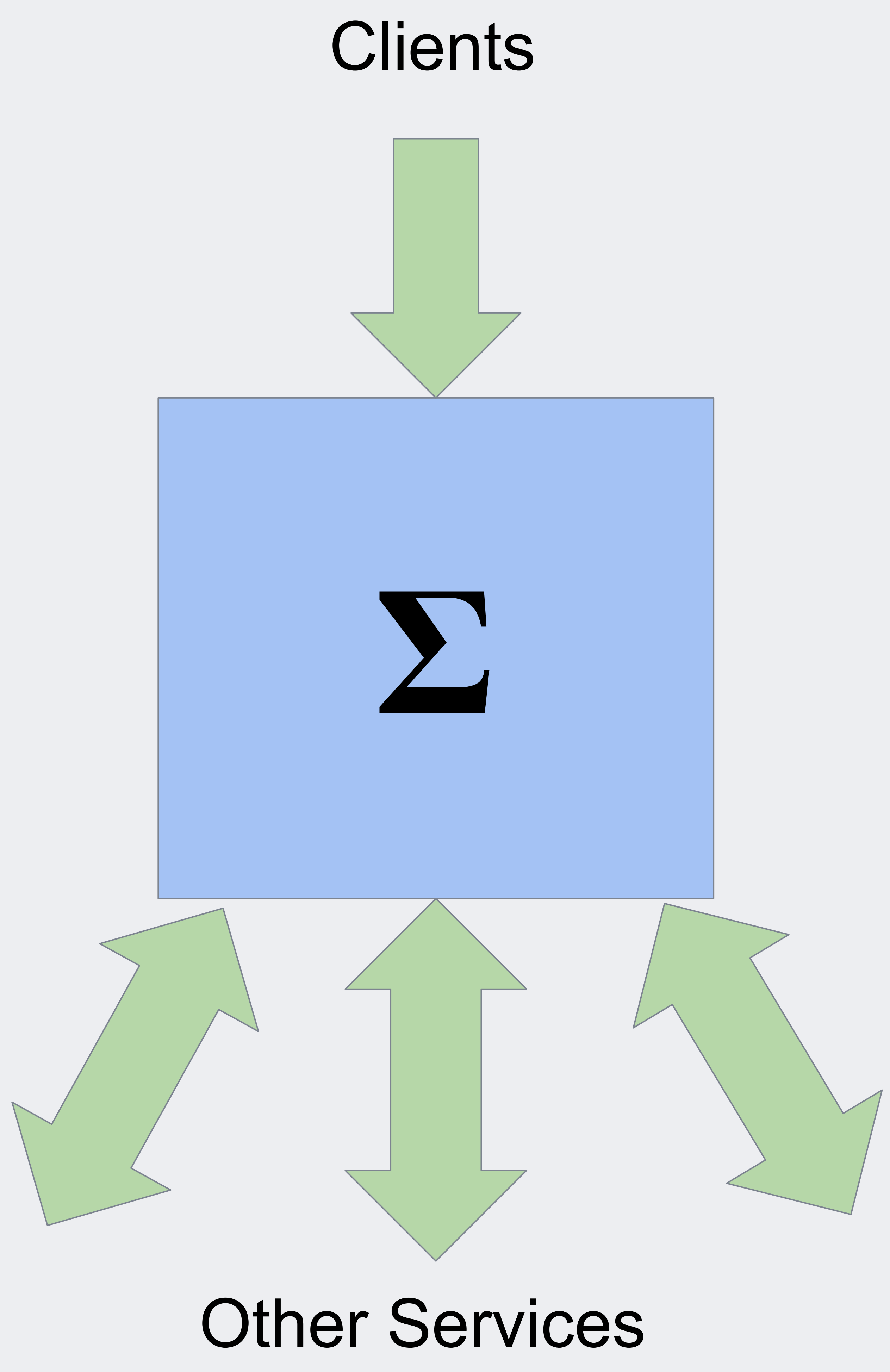
Haskell in the datacentre!

Simon Marlow

Facebook

(FHPC ‘17, September 2017)

Haskell powers Sigma



- A platform for detection
  - Used by many different teams
  - Mainly for anti-abuse
    - e.g. spam, malicious URLs
  - Machine learning + manual rules
  - Also runs Duckling (NLP application)
- Implemented mostly in Haskell
- Hot-swaps compiled code

At scale...

- Sigma runs on thousands of machines
  - across datacentres in 6 locations
- Serves 1M+ requests/sec
- Code updated hundreds of times/day

How does Haskell help us?

- Type safety: pushing changes with confidence
- Seamless concurrency
- Concise DSL syntax
- Strong guarantees:
  - Absence of side-effects within a request
  - Correctness of optimisations
    - e.g. memoization and caching
  - Replayability
  - Safe asynchronous exceptions

This talk: Performance!

- Our service is *latency sensitive*
- So obviously end-to-end performance matters
  - but it’s not all that matters

This talk: Performance!

- Our service is *latency sensitive*
- So obviously end-to-end performance matters
  - but it's not all that matters
- Utilise resources as fully as possible

This talk: Performance!

- Our service is *latency sensitive*
- So obviously end-to-end performance matters
  - but it’s not all that matters
- Utilise resources as fully as possible
- Consistent performance (SLA)
  - e.g. “99.99% within *N* ms”

This talk: Performance!

- Our service is *latency sensitive*
- So obviously end-to-end performance matters
  - but it's not all that matters
- Utilise resources as fully as possible
- Consistent performance (SLA)
  - e.g. "99.99% within *N* ms"
- Throughput vs. latency

Parallelism?

- Platform runs multiple requests in parallel
- **No compute parallelism within a request**
- But we do want **data-fetching parallelism**
- Like a webserver

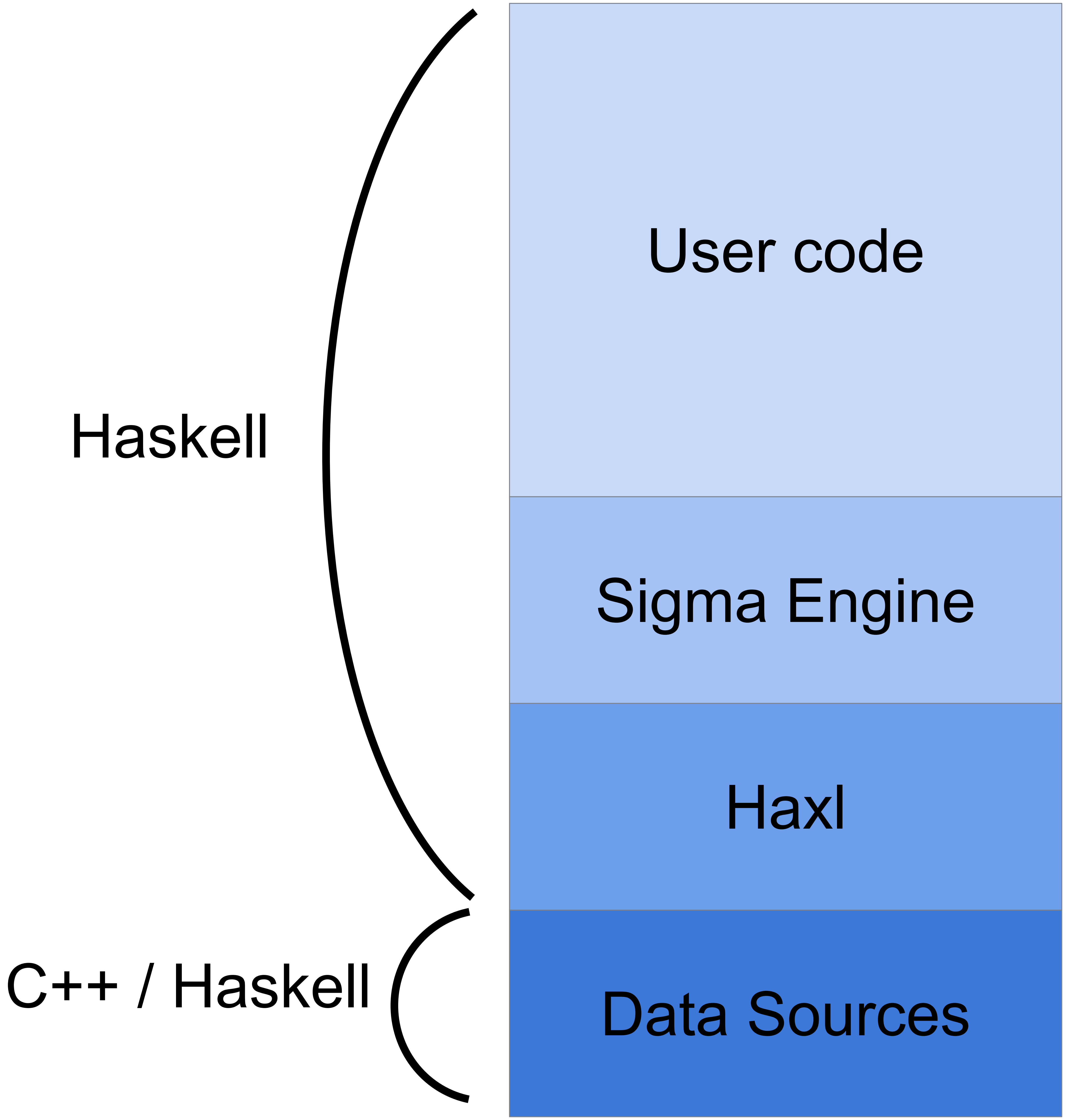
Not a single highly-tuned application

- One platform, many applications
  - under constant development by many teams
- Complexity and rate of change mean challenges for maintaining high performance.
- Lots of techniques
  - both “social” and technical

Tackle performance at the...

- **User level**
  - helping our users care about performance
- **Source level**
  - abstractions that encourage performance
- **Runtime level**
  - low-level optimisations and tuning
- **Service level**
  - making good use of resources

1. Performance at the
2. user level


Haskell
C++ / Haskell
User code
Sigma Engine
Haxl
Data Sources

Connecting users with perf

- Users care firstly about functionality
  - So we made a DSL that emphasizes concise expression of functionality, abstracts away from performance (more later)
  - but we can’t insulate clients from performance issues completely...


Fetch all the data!
Photo: Scott Schiller, CC by 2.0


Log everything!
All the time!

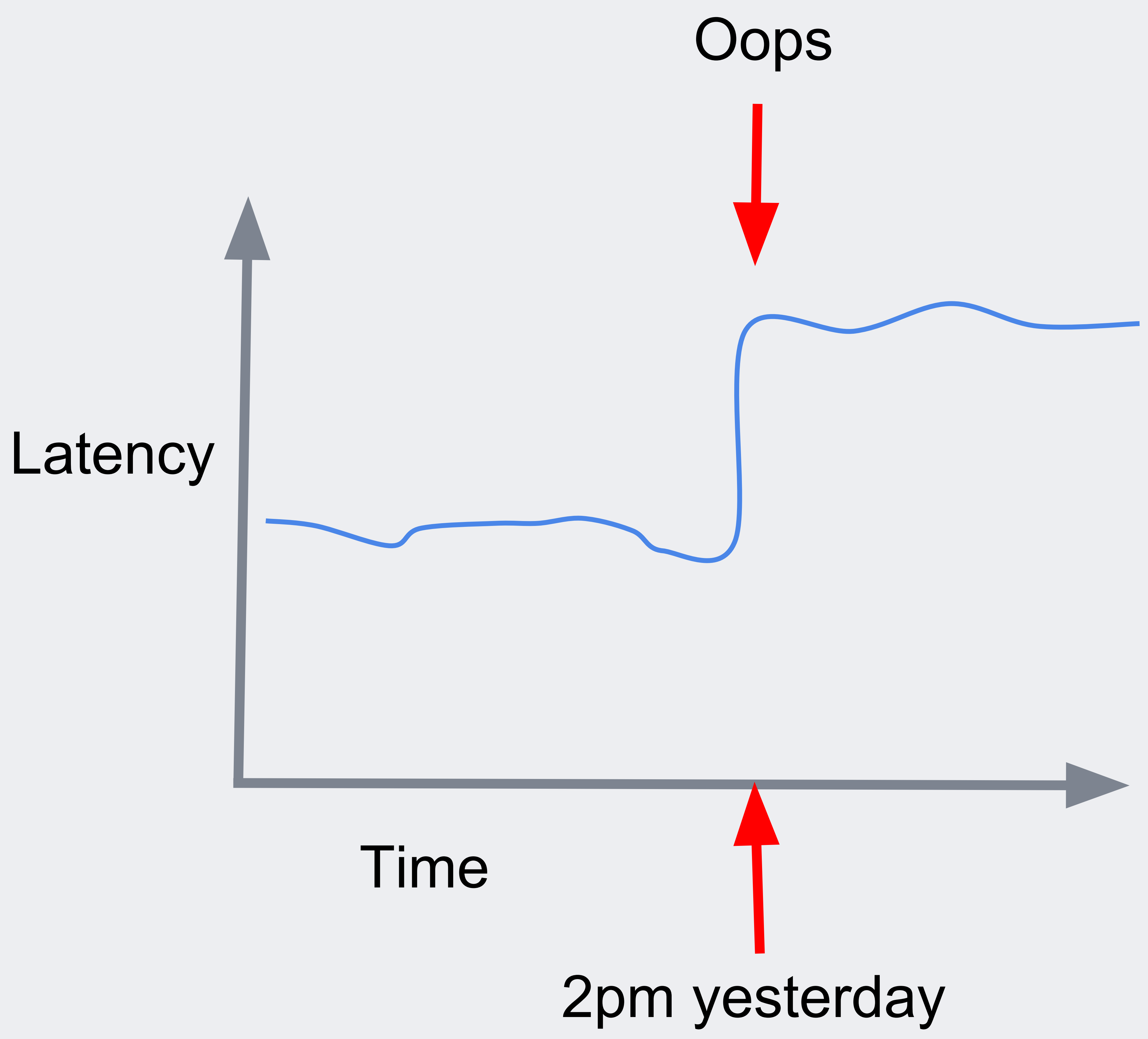
numCommonFriends, two ways

```
numCommonFriends a b = do
  af <- friendsOf a
  aff <- mapM friendsOf af
  return (count (b `elem`) aff)

numCommonFriends a b = do
  af <- friendsOf a
  bf <- friendsOf b
  return (length (intersect af bf))
```

When regressions happen

- Problem: code changes that regress performance
- Platform team must diagnose + fix
- This is bad:
  - time consuming, platform team is a bottleneck
  - error prone
  - some regressions still slip through



Goal: make users care about perf

- But without getting in the way, if possible
- Make perf visible when it matters
  - avoid regressions getting into production
- Make perf hurt when it really matters

Offline profiling is inaccurate

- Accuracy requires
  - compiling the code (not using GHCi)
  - running against representative production data
  - don’t want to make users go through this themselves

Our solution:
Experiments

Experiments: self-service profiling

- At the code review stage, run automated benchmarks against production data, show the differences
- Direct impact of the code change is visible in the code review tool
- Result: many fewer perf regressions get into production

More client-facing profiling

- Can’t run full Haskell profiling in production
  - 2x perf overhead, at least
- Poor-man’s profiling:
  - `getAllocationCounter` counts per-thread allocations
  - instrument the Haxl monad
  - manual annotations (withLabel “foo” $ …)
  - some automatic annotations (top level things)

Make perf hurt when it really matters

- Beware elephants

- (unexpectedly large requests that degrade performance for the whole system)

How do elephants happen?

- Accidentally fetching too much data
- Accidentally computing something really big
  - (or an infinite loop)
- Corner cases that didn't show up in testing
- Adversary-controlled input (avoid where possible)

Kick the elephants off the server

- *Allocation Limits*
  - Limit on the total allocation of a request
  - Counts memory allocation, not deallocation
  - Allocation is a proxy for work

- Catches heavyweight requests (“elephants”)
- And (some) infinite loops

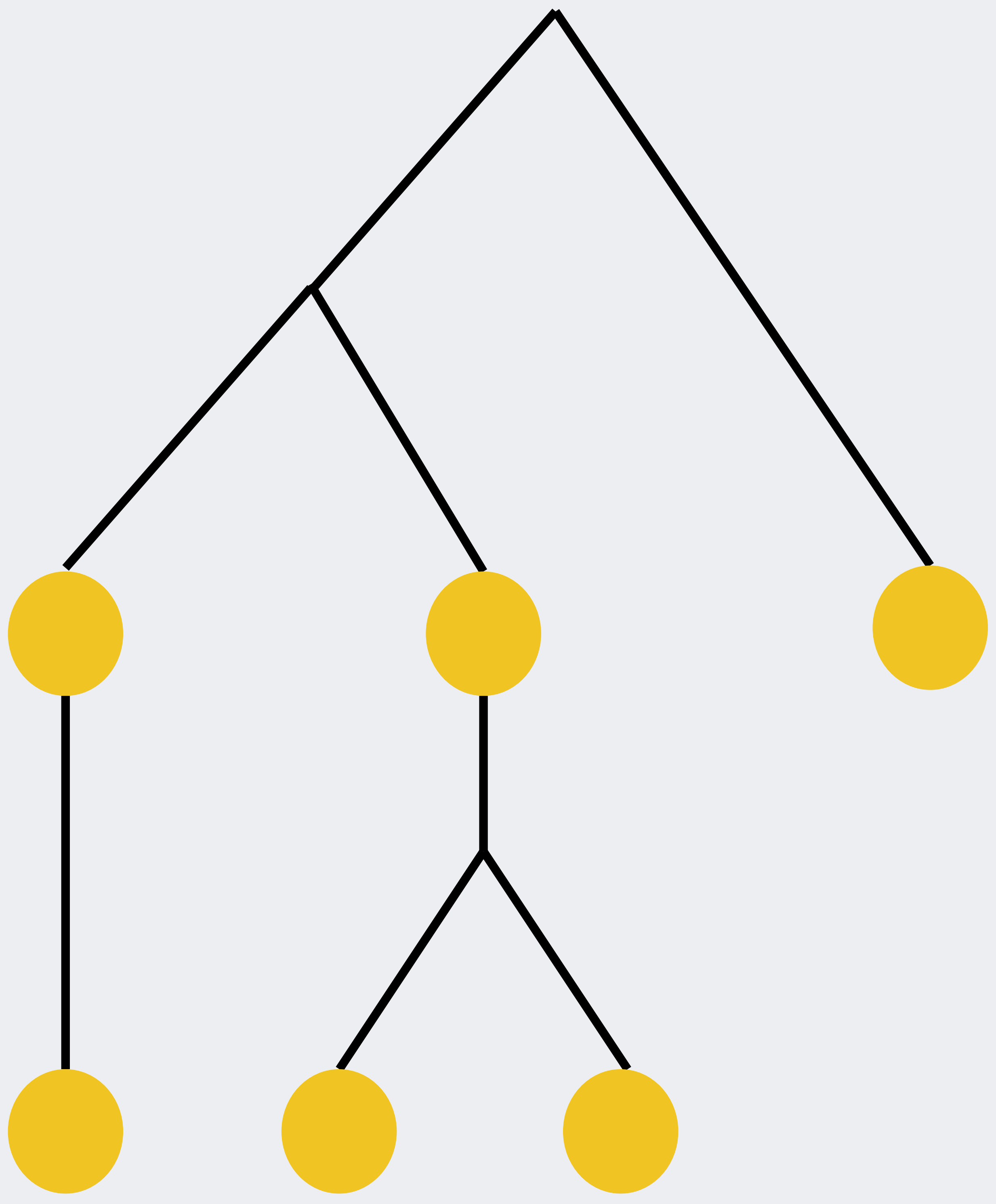
A not-so-gentle nudge

- As well as being an important back-stop to keep the server healthy…
- This also encourages users to optimise their code
  - ...and debug those elephants
  - which in turn, encourages the platform team to provide better profiling tools

Performance at the source level

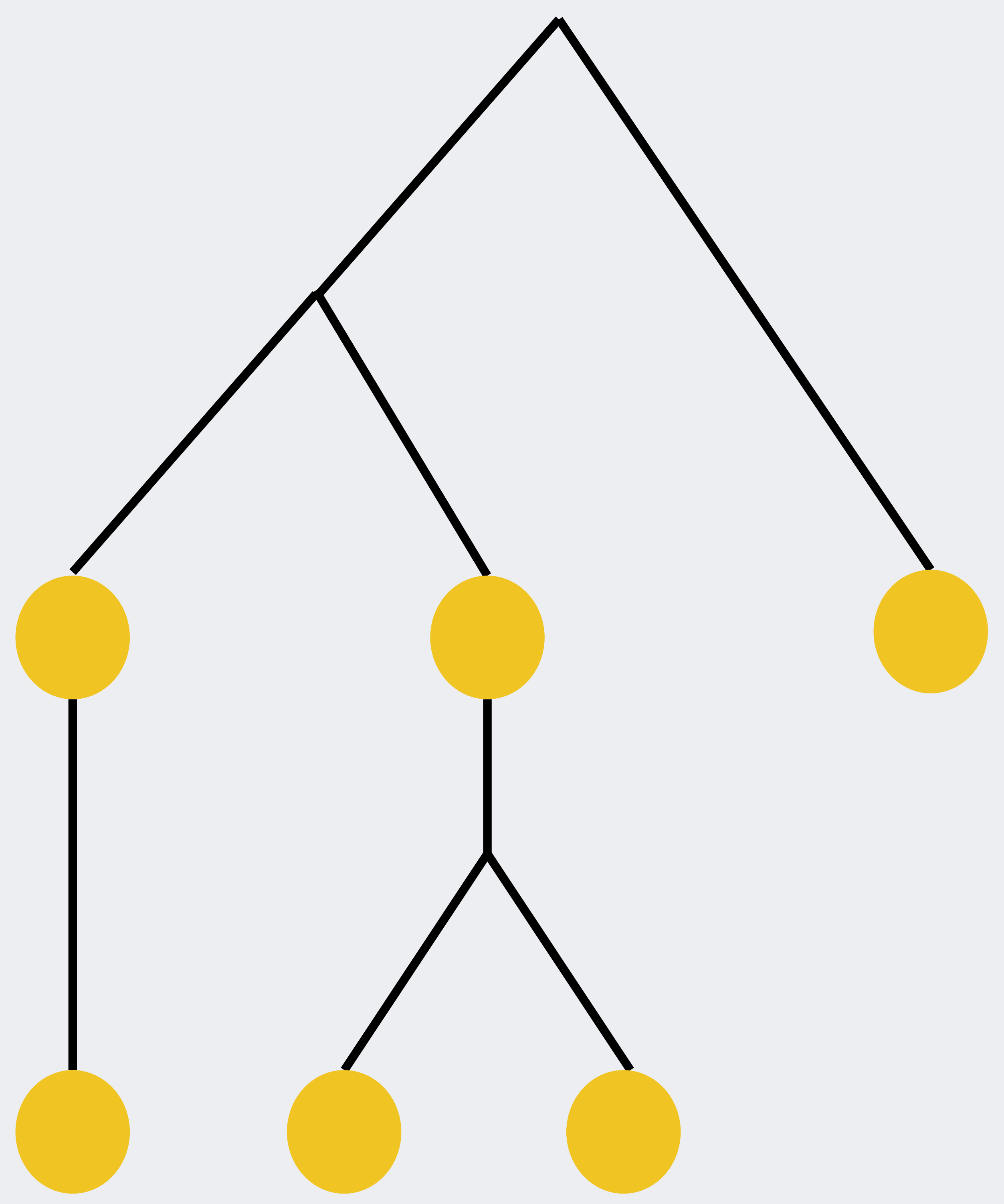
Concurrency matters

- “fetch data and compute with it”
- A request is a graph of data fetches and dependencies
- Most systems assume the worst
  - there might be side effects!
  - so execute sequentially unless you explicitly ask for concurrency.



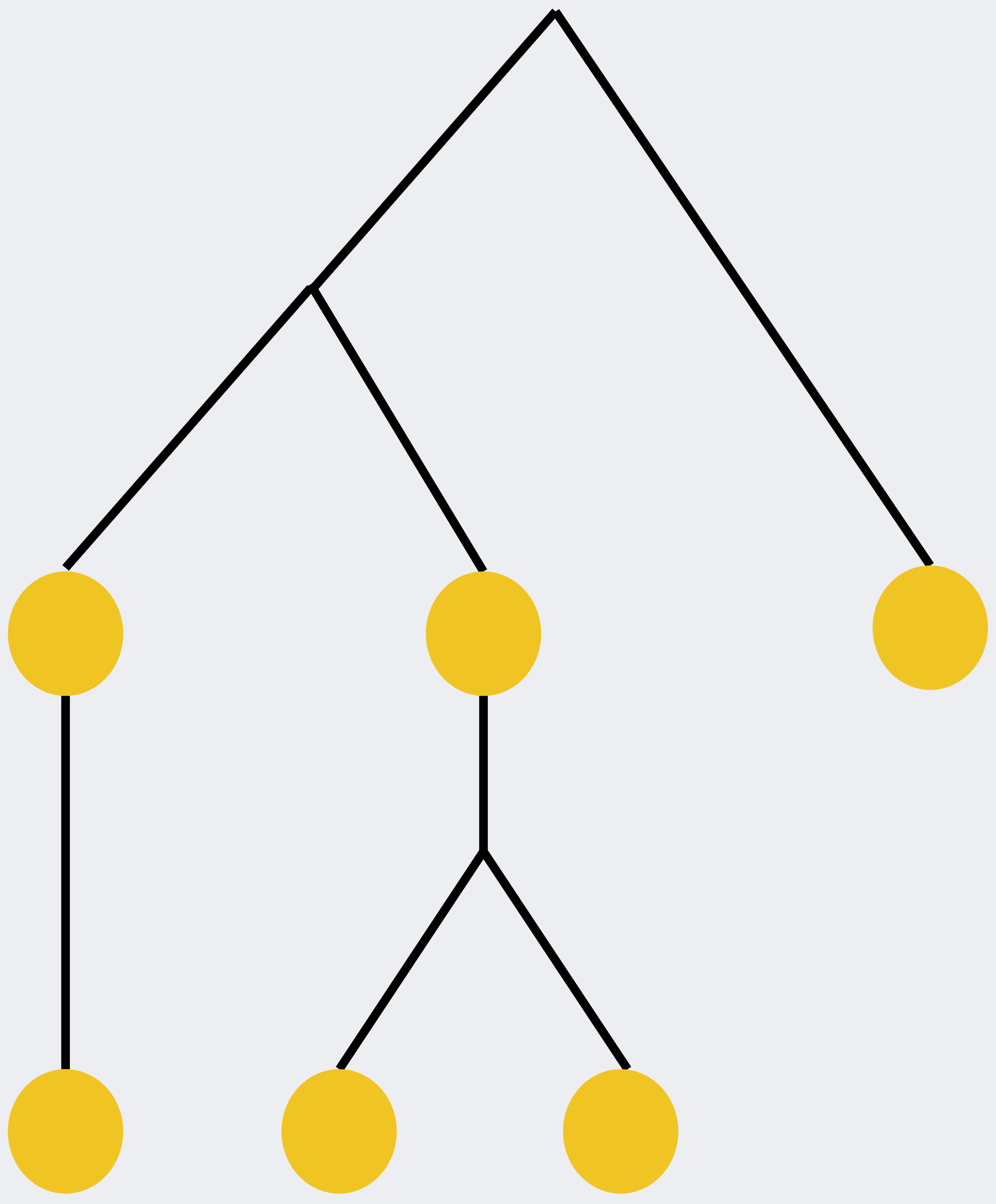
Concurrency matters

- But explicit concurrency is hard
  - Need to spot where we can use it
  - Clutters the code with operational details
  - Refactoring becomes harder, and is likely to get the concurrency wrong

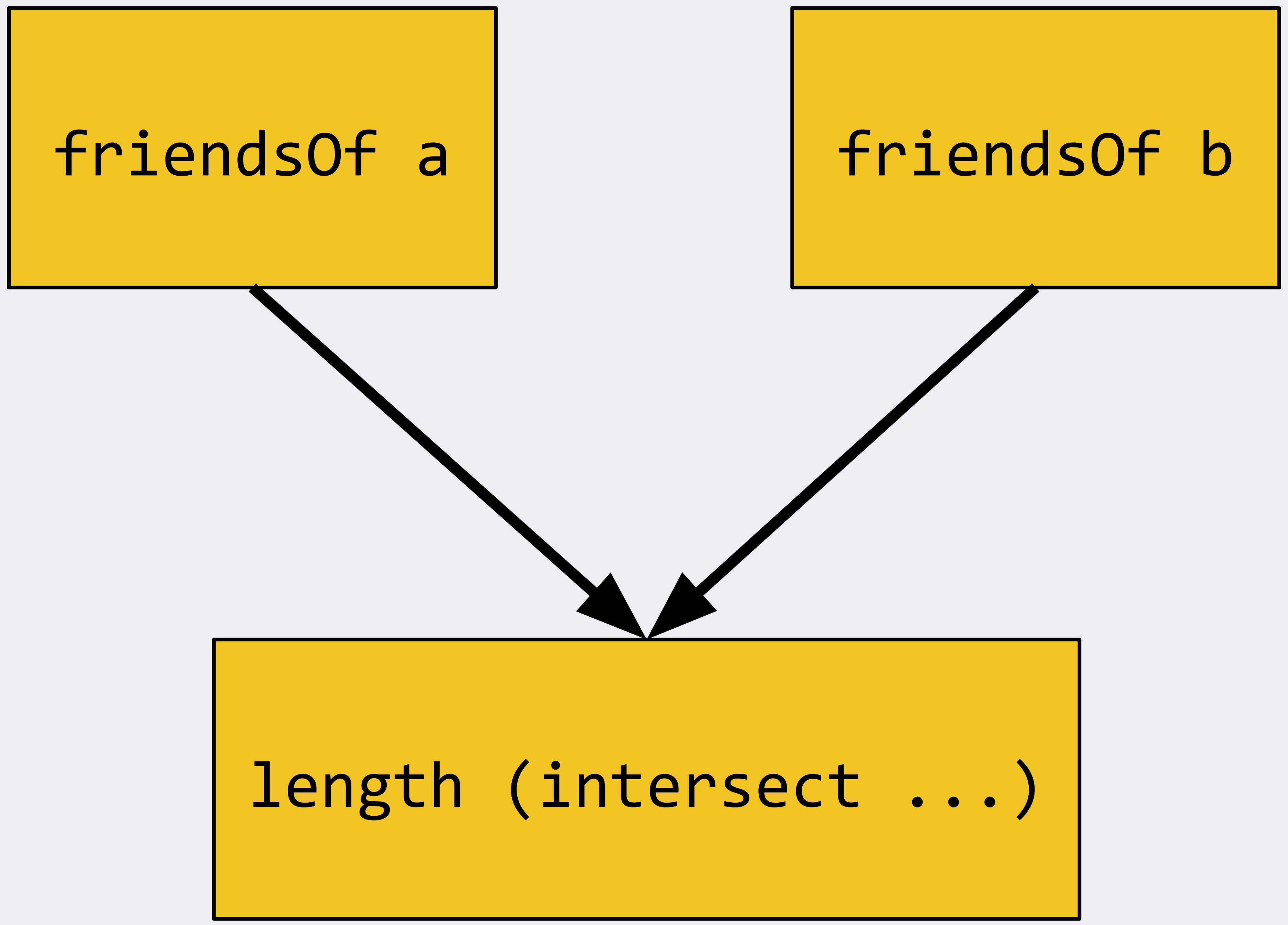


Concurrency matters

- What if we flip the assumption?
- **Assume that there are no side effects**
- Fetching data is just a function
- Now we are free to exploit concurrency as far as data dependencies allow.
- Enforce “no side-effects” with the type system and module system.



```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```



FP with remote data access

- Treat data-fetching as a function

```
friendsOf :: Id -> Haxl [Id]
```

- Implemented as a (cached) data-fetch
- Might be performed concurrently or batched with other data fetches
- From the user's point of view, "friendsOf x" always has the same value for a given x.

Why `friendsOf :: Id -> Haxl [Id]` ?

- Data-fetches can fail
  - Haxl includes exceptions
  - Exceptions must not prevent concurrency (not EitherT)
- Haxl monad is where we implement concurrency
  - otherwise it would have to be in the compiler

How does concurrency in Haxl work?

- By exploiting Applicative:

```
(>>=) :: Monad m         => m a           → (a → m b) → m b
```

dependency

```
(<*>) :: Applicative f => f (a → b) → f a           → f b
```

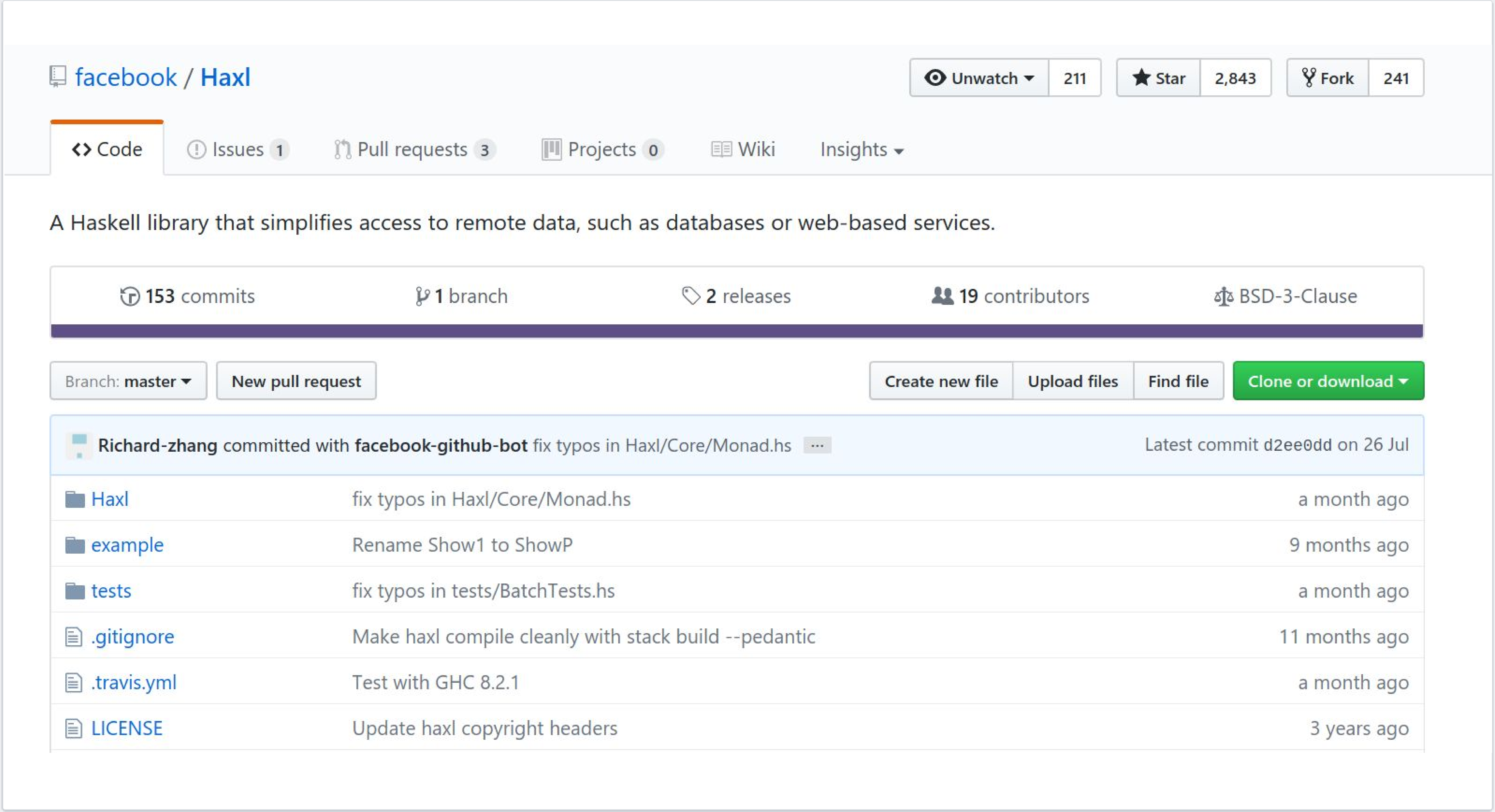
independent

Applicative concurrency

- Applicative instance for Haxl allows data-fetches in both arguments to be performed concurrently
- Things defined using Applicative are automatically concurrent, e.g. `mapM`:

```
friendsOfFriends :: Id -> Haxl [Id]
friendsOfFriends x = concat <$> mapM friendsOf x
```

- (details in Marlow et. al. ICFP'14)


facebook / Haxl
Unwatch 211
Star 2,843
Fork 241
Code
Issues 1
Pull requests 3
Projects 0
Wiki
Insights
A Haskell library that simplifies access to remote data, such as databases or web-based services.
153 commits
1 branch
2 releases
19 contributors
BSD-3-Clause
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
Richard-zhang committed with facebook-github-bot fix typos in Haxl/Core/Monad.hs
Latest commit d2ee0dd on 26 Jul
Haxl
fix typos in Haxl/Core/Monad.hs
a month ago
example
Rename Show1 to ShowP
9 months ago
tests
fix typos in tests/BatchTests.hs
a month ago
.gitignore
Make haxl compile cleanly with stack build --pedantic
11 months ago
.travis.yml
Test with GHC 8.2.1
a month ago
LICENSE
Update haxl copyright headers
3 years ago

Clones!

- Stitch (Scala; @Twitter; not open source)
- clump (Scala; open source clone of Stitch)
- Fetch (Scala; open source)
- Fetch (PureScript; open source)
- muse (Clojure; open source)
- urania (Clojure; open source; based on muse)
- HaxlSharp (C#; open source)
- fraxl (Haskell; using Free Applicatives)

Haxl solves half of the problem

- What about this?

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

- Should we force the user to write

```
numCommonFriends a b =
  (length . intersect)
      <$> friendsOf a
      <*> friendsOf b
```

- Maybe small examples are OK, but this gets really hard to do in more complex cases

```
do x1 ← a
   x2 ← b x1
   x3 ← c
   x4 ← d x3
   x5 ← e x1 x4
   return (x2,x4,x5)
```

⇒

```
do ((x1,x2),x4) <- (,)
     <$> (do x1 <- a
             x2 <- b x1
             return (x1,x2))
     <*> (do x3 <- c; d x3)
    x5 <- e x1 x4
    return (x2,x4,x5)
```

- And after all, our goal was to derive the concurrency automatically from data dependencies

{-# LANGUAGE ApplicativeDo #-}

- Have the compiler analyse the **do** statements
- Translate into Applicative wherever data dependencies allow it

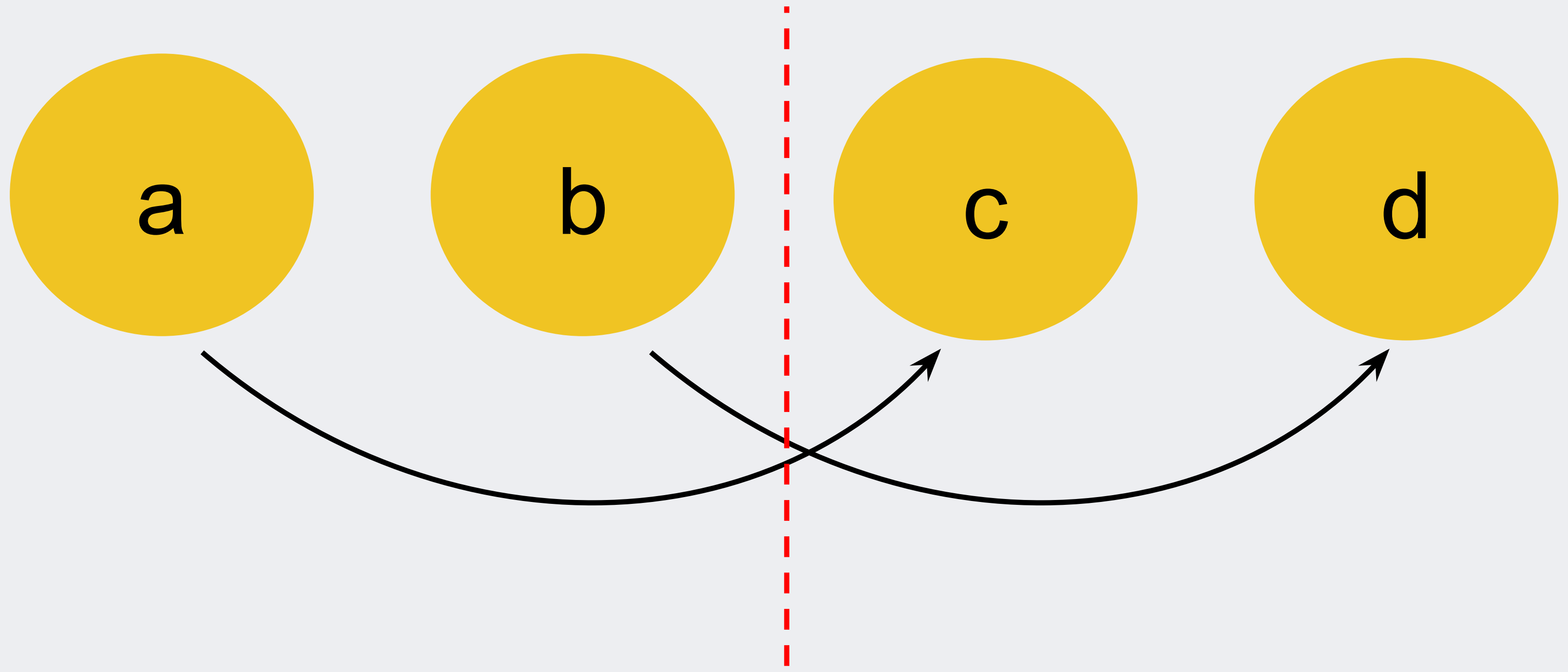
```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

```
numCommonFriends a b =
  (length . intersect)
      <$> friendsOf a
      <*> friendsOf b
```

One design decision

How should we translate this?

```
do x1 <- a
   x2 <- b
   x3 <- c x1
   x4 <- d x2
   return (x3,x4)
```




```
((,) <$> A <*> B) >>= \(x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

(A | B) ; (C | D)

```
(,) <$> (A >>= \x1 -> C[x1])
    <*> (B >>= \x2 -> D[x2])
```

(A ; C) | (B ; D)

Which is best?

```
((,) <$> A <*> B) >>= \(x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

(A | B) ; (C | D)

```
(,) <$> (A >>= \x1 -> C[x1])
    <*> (B >>= \x2 -> D[x2])
```

(A ; C) | (B ; D)

More concurrency

What laws do we assume?

```
((,) <$> A <*> B) >>= \(x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

valid for any law-abiding Monad

```
(,) <$> (A >>= \x1 -> C[x1])
    <*> (B >>= \x2 -> D[x2])
```

only valid for commutative Monads

- We chose to assume law-abiding Monads only
- This sometimes restricts the available concurrency
- If the user writes this instead, they get a better result:

```
do x1 <- a
   x3 <- c x1
   x2 <- b
   x4 <- d x2
   return (x3,x4)
```

- ApplicativeDo is ultimately a **heuristic compiler optimisation**, there are many ways to defeat it.

Should concurrency be the compiler's job?

- When there are no (or few) side effects, implicit concurrency is a better default
  - More concise code
  - Less brittle
  - Easier to refactor
  - Can still use explicit concurrency
    - (via Applicative, mapM etc.)

Should concurrency be the compiler's job?

- Against:
  - IT'S INVISIBLE MAGIC
  - Can miss opportunities
  - Easy to go wrong when there are side-effects

What about side effects?

- In Sigma we cleanly separate effects
  - Rules return actions to perform
- Even if you have a few side effects, explicit ordering is possible, turn off ApplicativeDo or use >>=

```
myFunction =
  writeSomeData >>= \_ ->
  readSomeData ...
```

Caching & memoization

All data fetches are cached

- Cache lives for the request only
- So “friendsOf x” always returns the same result in a given request
- This is liberating!
  - never need to pass around fetched data
  - just fetch it wherever you need it
  - caching reduces coupling, increases modularity
- Cache enables record + replay for testing

Taking caching further

```
memo :: Key -> Haxl a -> Haxl a
```

- memoize an arbitrary “Haxl a” computation
  - (again, within a request)
- Even more liberating!
  - profile to find duplicate work, add memo
  - no need to pass results around
  - great for modularity

Performance at the runtime level

Scheduling

- GHC uses an N/M threading model:
  - N *capabilities* (think: OS thread)
  - M Haskell threads (lightweight, or bound to OS thread)
  - runtime scheduler attempts to load-balance M onto N
- Maximum real parallelism = N

Competing concerns

- N should be large enough to max out the CPU
  - including Hyperthreaded cores (~30% of CPU)
- If GHC doesn't schedule our M workers perfectly onto the N capabilities, we waste some CPU
- Easiest way to fix this is to make N larger
  - (give the scheduling problem to the OS)
- But...

Garbage Collection

- GHC uses parallel stop-the-world GC
- Running on the same N threads

- Problem: parallel GC degrades badly if N > #cores
  - due to work-stealing
- So increasing N to counteract scheduling imperfection causes GC to slow down

Solution: let GC use <N threads

- We added a new option, `+RTS -qn`*n*
- Limits the number of GC threads to *n*
- Picks dynamically at runtime which threads to use
  - use busy threads for GC, leave idle threads asleep
- e.g. on a 16-core box we could use

  `+RTS -N48 -qn16`

  and easily max out the CPU

  provided we have enough worker threads

-qn is the default

- This worked so well, that I enabled -qn by default to counteract the slowdown when N > #cores
- Benchmarks: -N8 -qn4 on 4-core laptop:

Program	Size	Allocs	Runtime	Elapsed	TotalMem
blackscholes	+0.0%	+0.0%	-72.5%	-72.0%	+9.5%
coins	+0.0%	-0.0%	-73.7%	-72.2%	-0.8%
mandel	+0.0%	+0.0%	-76.4%	-75.4%	+3.3%
matmult	+0.0%	+15.5%	-26.8%	-33.4%	+1.0%
nbody	+0.0%	+2.4%	+0.7%	0.076	0.0%
parfib	+0.0%	-8.5%	-33.2%	-31.5%	+2.0%
partree	+0.0%	-0.0%	-60.4%	-56.8%	+5.7%
prsa	+0.0%	-0.0%	-65.4%	-60.4%	0.0%
queens	+0.0%	+0.2%	-58.8%	-58.8%	-1.5%
ray	+0.0%	-1.5%	-88.7%	-85.6%	-3.6%
sumeuler	+0.0%	-0.0%	-47.8%	-46.9%	0.0%

Aside: multiple processes?

- Could we run N processes instead?
  - Avoids GC sync issues
  - But sharing is much harder
    - The server process has shared caches and process-level state which would be harder to manage
    - Monitoring, debugging etc. are easier with one process

Multiple heaps?

- aka the Erlang model
- Again, managing shared caches becomes harder
- But having local independently-collected heaps in some form is the way forwards

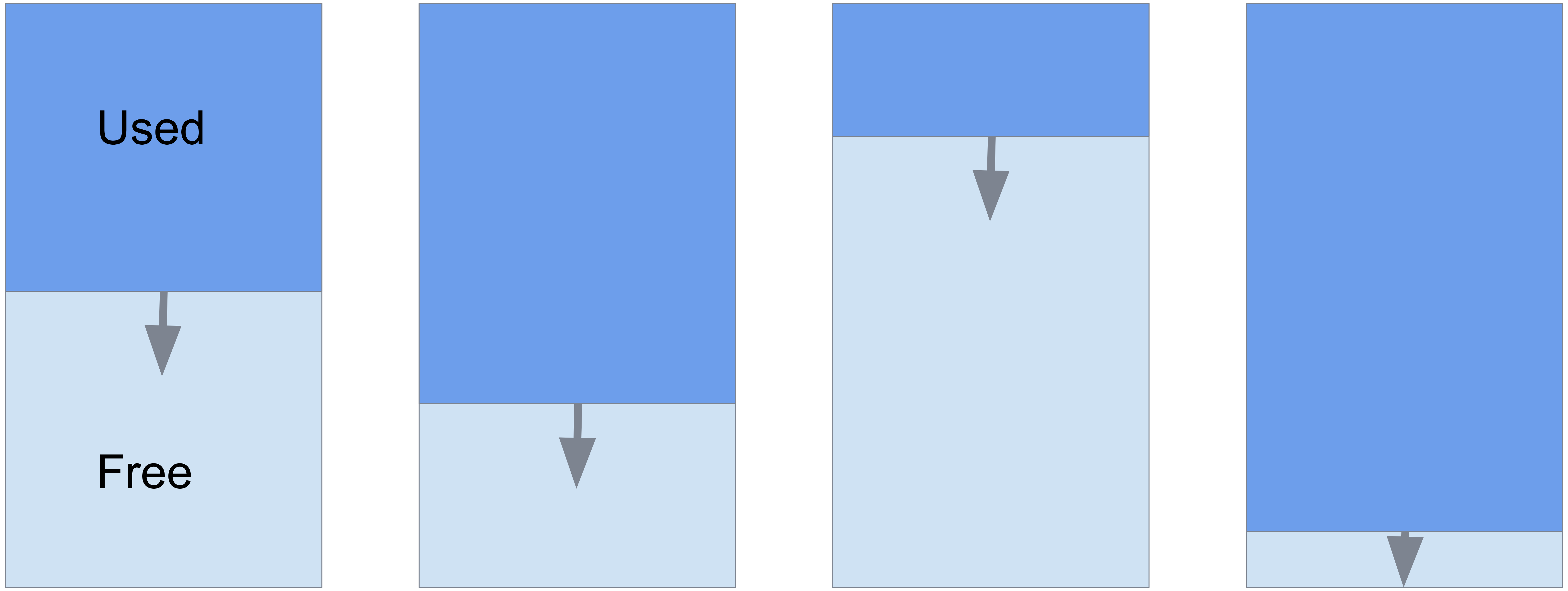
Let's talk about… GC

- GHC has a parallel, generational, stop-the-world copying collector
  - Allocate like crazy, then stop and copy everything live
  - We have to worry about:
    - overall throughput
    - pause time
    - synchronising threads to stop-the-world

Improving throughput

- GC is a space/time tradeoff
  - We improve throughput by using more memory
  - More memory = fewer GCs
- But how is the memory divided up?
  - By default, GHC divides nursery size evenly by N capabilities
  - This was fine for small nurseries (L2 cache sized)
  - But we want a multi-GB nursery

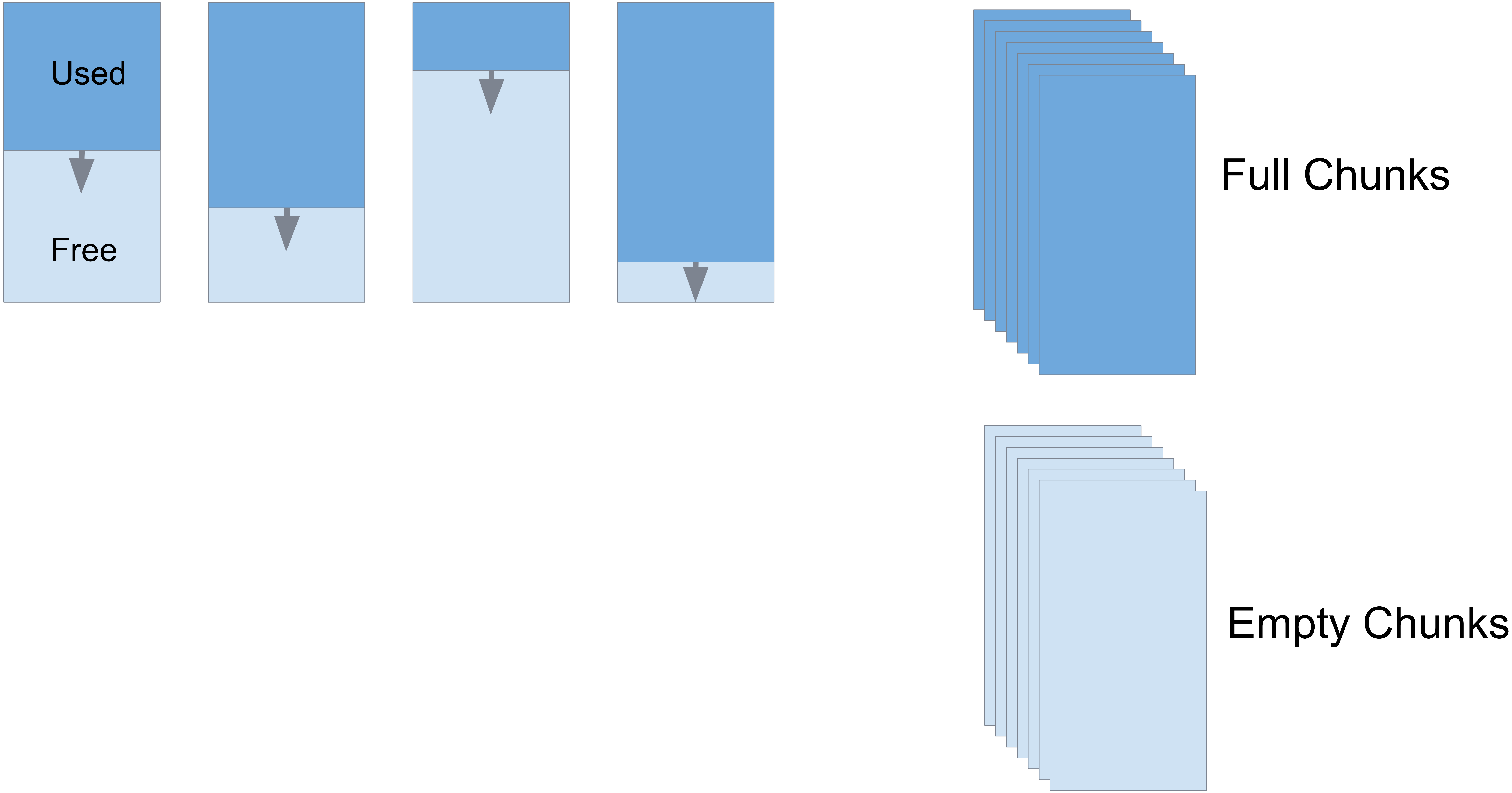
Nurseries



Problem: capabilities allocate at different rates, so we GC before we have filled all the memory

Solution: nursery chunks

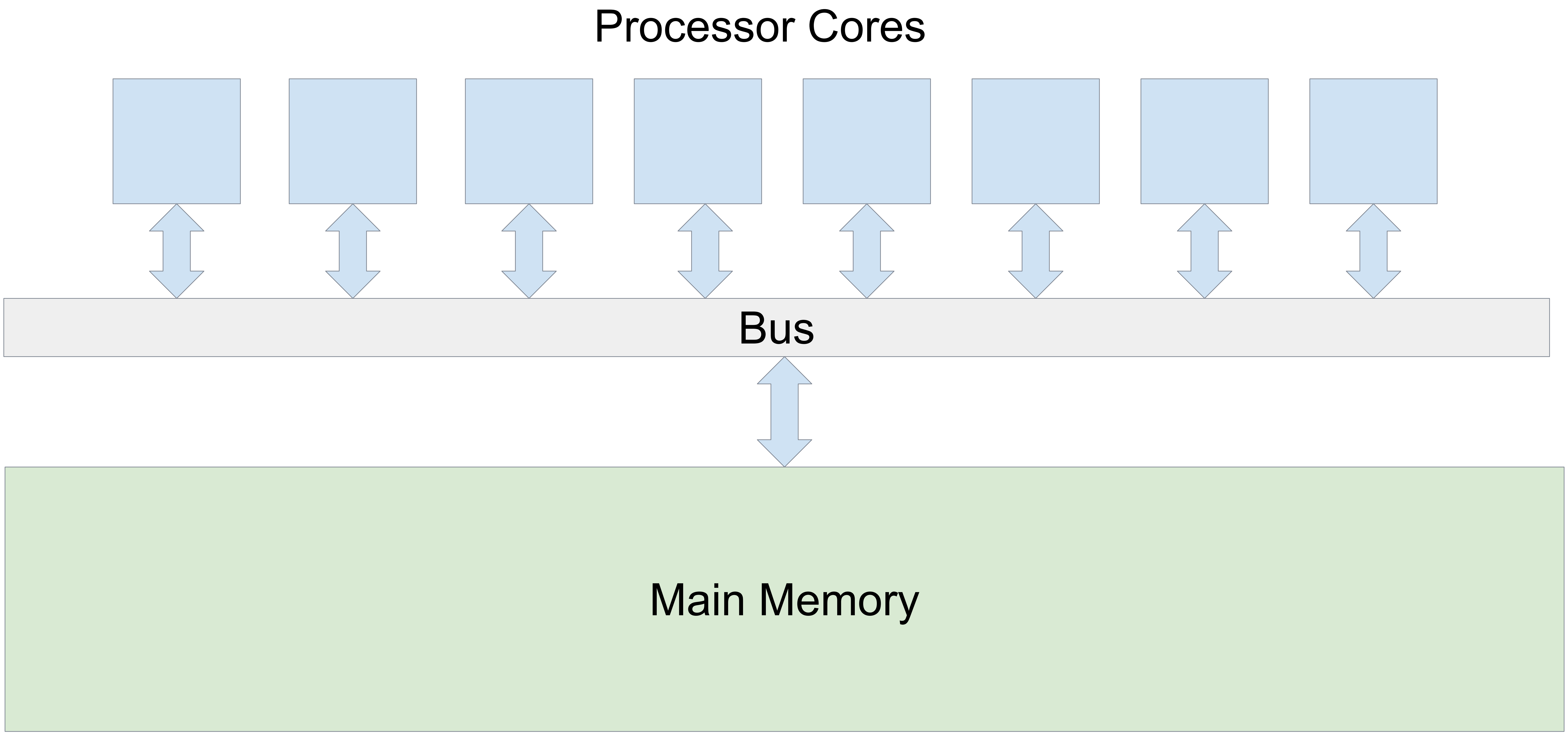
- Divide the nursery into fixed-size chunks
  - e.g. 4MB

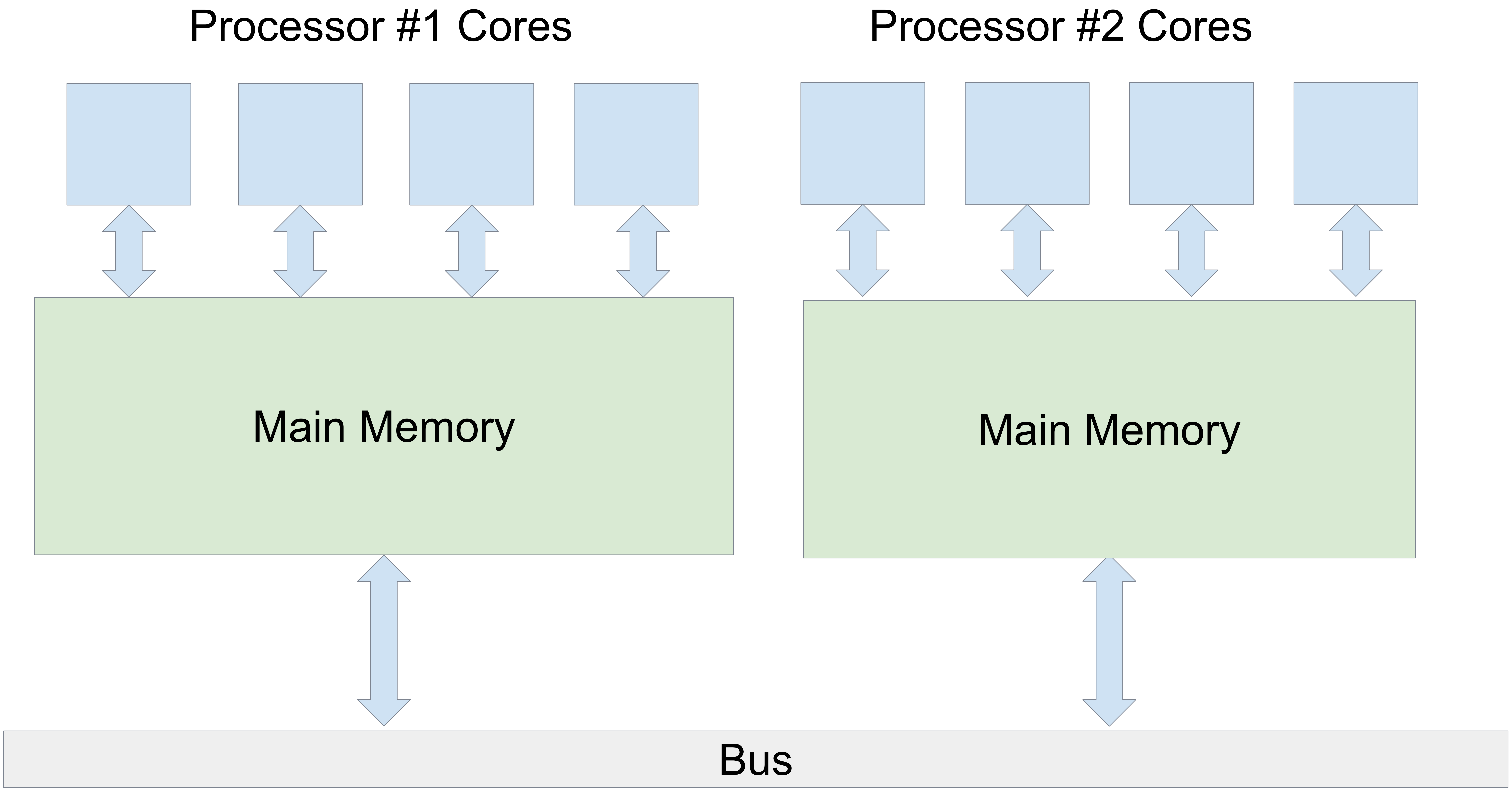

Used
Free
Full Chunks
Empty Chunks

Nursery chunks

- GC when all the chunks are full
- Very little wastage
- Significantly reduced GC overhead

- We can optimise memory access further...


Processor Cores
Bus
Main Memory


Processor #1 Cores
Processor #2 Cores
Main Memory
Main Memory
Bus

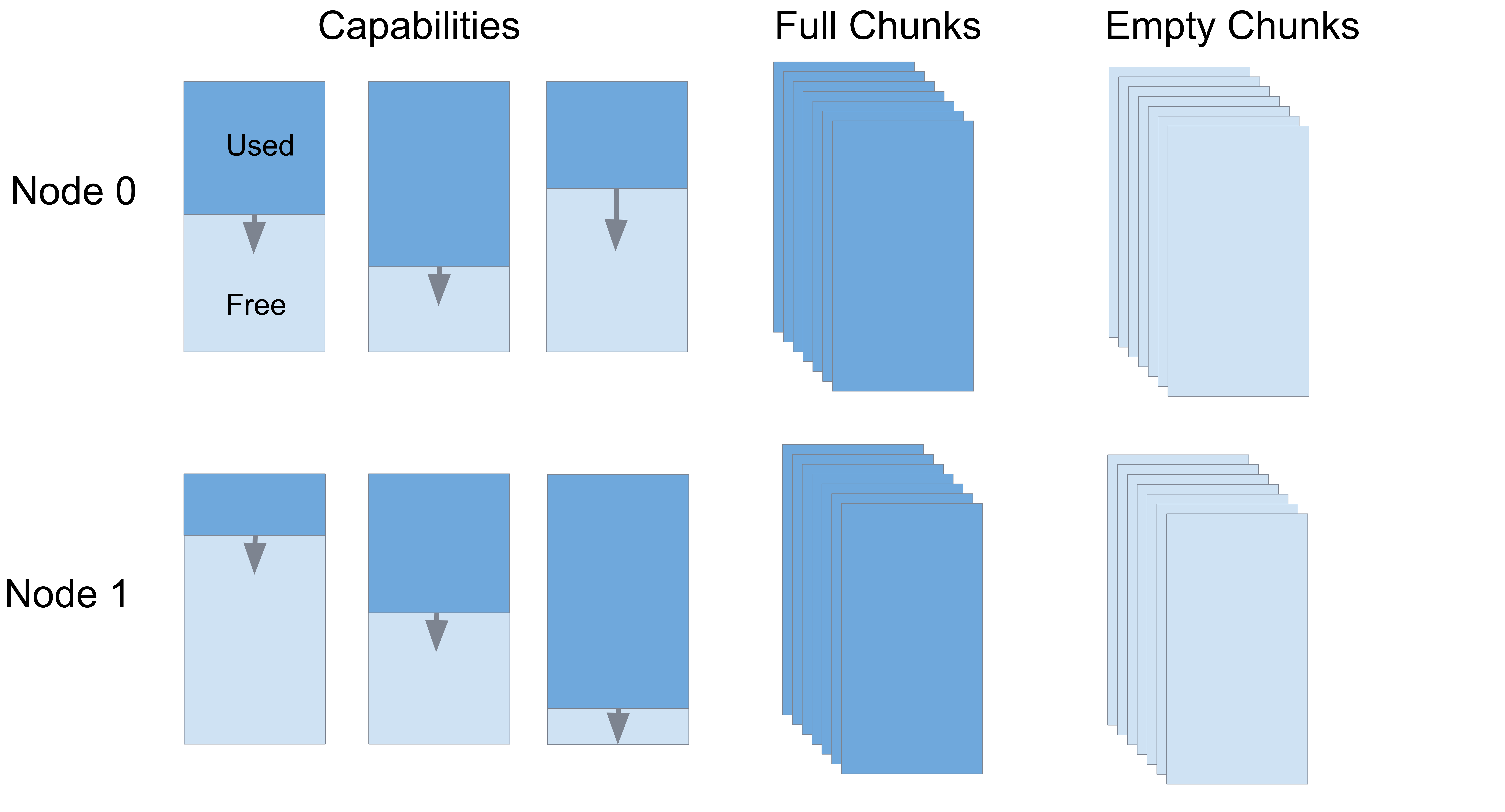
Non-Uniform Memory Access (NUMA)

- Machine divided into *nodes*
- Accessing memory on the local node is faster (e.g. 2x)
- In the absence of any hints, the OS allocates memory randomly, so we'll get ~50% remote access

Observation

- *Most memory access is to the nursery*
  - Since our nursery is much larger than the cache
  - Most memory access is to recently allocated objects

- Opportunity:
  - Ensure that nursery memory accesses are local


Capabilities
Full Chunks
Empty Chunks
Node 0
Used
Free
Node 1

Does it help?

- Higher percentage of local memory access
- Could be better
  - Where are the rest of the remote accesses?
- Tradeoff
  - when the pool is empty, do we steal from the other node, or run the GC?

Reducing pause times

- Some fraction of the heap data is *mostly static*
- In Sigma, it's static configuration data
  - needs to be cached, for fast access
  - but rarely changes
- No point in having the GC copy this data on every (major) collection

New in GHC 8.2: compact regions!

```
compact :: a -> IO (Compact a)
getCompact :: Compact a -> a
```

- The compact value is treated as a single object by the GC, so O(1)
- `compact` is O(n), similar overhead to GC

takes an arbitrary value and copies it into a consecutive region of memory

returns a reference to the compacted value

Compact unlocks new use cases

- Now we can have an arbitrary amount of Haskell data in the heap, with zero GC overhead
- Some caveats:
  - Data can’t contain functions, mutable things, ByteString
  - Pay O(n) to update the data
- Why no functions?
  - Functions might refer to CAFs
- Why no ByteString?
  - Pinned memory :(

Optimising FFI calls

- A source of pain: callbacks from C/C++
- How can you implement an efficient Haskell wrapper for a C++ API like this

```
void sendRequest(
 Request &req,
 std::function<void(Response&)> callback
);
```

The usual way

```
type HaskellCallback = Ptr Response -> IO ()

foreign import ccall “wrapper”
  mkCallback :: HaskellCallback
             -> IO (FunPtr HaskellCallback)

sendRequest :: Request -> IO (MVar Response)
sendRequest req = do
  mvar <- newEmptyMVar
  callback <- mkCallback $ \responsePtr -> do
    r <- unmarshal responsePtr
    putMVar r
  -- send the request, passing the callback
```

But this is slow...

- mkCallback has to generate some code
  - and we have to free it later
- When C++ calls the callback
  - Creates a new Haskell thread and runs it
  - Will block if the GC is currently running
  - Calls into Haskell are heavyweight

Faster async callbacks

- GHC exposes a new C API:

```
void hs_try_putmvar (
  int capability,
  HsStablePtr sp
);
```

Hint (int capability)

StablePtr (MVar ()) (HsStablePtr sp)

- Behaves just like

```
tryPutMVar :: MVar () -> IO ()
```

- But called from C/C++

How to use it

```
receive :: MVar () -> Ptr Response -> IO Response
receive m p = do
  takeMVar m
  peek p
```

- We need a callback wrapper on the C side to call hs_try_putmvar()
- Memory to store the result can be Haskell-allocated and GC'd, no need to free

Furthermore...

- `hs_try_putmvar()` is **non-blocking**
- If it can do the `putMVar` immediately, it does
- If GC is in progress, or the capability is running, it sends a message
- Callbacks blocking or failing is a source of problems: `hs_try_putmvar()` avoids all that
- We saw some nice speed and scalability improvements from this

Performance at the service level

Performance tradeoffs

- For best throughput:
  - Handle as many concurrent requests as we can fit in the memory
  - Defer GC as long as possible
- But these will negatively affect latency
- We found we can get better throughput by tuning

How to exploit this?

- We have a latency SLA (Service Level Agreement)
- Multiple instances of the service, with different SLAs
  - Throughput-tuned SLA is weaker
- Find applications that can accept the weaker SLA, and move them to the throughput-tuned service
- The weaker the SLA, the more flexibility we have to time-shift requests

Summary

Messages

- Abstract away from concurrency (Haxl + ApplicativeDo)
- Help users care about perf, and give them the tools to understand it
- Exploit latency-insensitivity in clients
- Runtime tricks:
  - GC scheduling, nursery chunks, NUMA, hs_try_putmvar, Compact