SCOOP performance and implementation

Scott West

ETH Zürich

February 19, 2013

Examining SCOOP performance

Since the research retreat:

- Added the missing benchmarks that were discussed
- Added 2 more languages: Scala (Akka) actors and Java (mutex/condition variables)
- Started to investigate SCOOP performance and solutions

Examining SCOOP performance

Since the research retreat:

- Added the missing benchmarks that were discussed
- Added 2 more languages: Scala (Akka) actors and Java (mutex/condition variables)
- Started to investigate SCOOP performance and solutions

Examining SCOOP performance

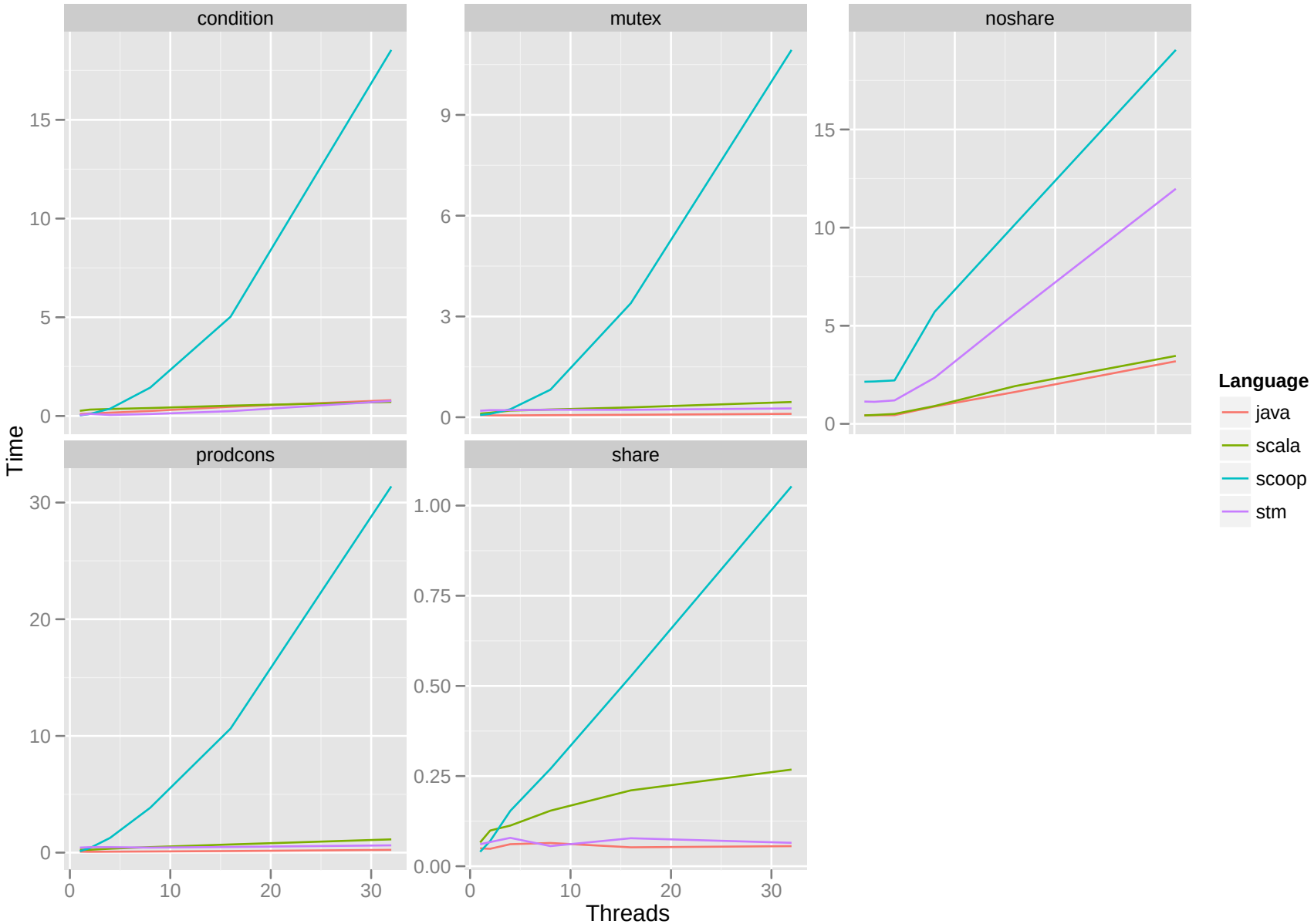
Since the research retreat:

- Added the missing benchmarks that were discussed
- Added 2 more languages: Scala (Akka) actors and Java (mutex/condition variables)
- Started to investigate SCOOP performance and solutions

Benchmark Overview

condition	two groups of threads, each increments a shared variable when it is odd/even.
mutex	multiple threads contend for a critical section.
no share	threads do not share memory or coordinate, perform a computationally bound task.
prodcons	multiple producers and consumers work on a single queue.
share	multiple threads all threads share memory, but do not guard access to it (if possible)

Benchmark results



Next steps

Theme from last time: measure then cut.

- SCOOP currently slows down superlinearly in the amount of work :(
- How do we find what we should cut?
- Profiling and code inspection.

Next steps

Theme from last time: measure then cut.

- SCOOP currently slows down superlinearly in the amount of work :(
- How do we find what we should cut?
- Profiling and code inspection.

Next steps

Theme from last time: measure then cut.

- SCOOP currently slows down superlinearly in the amount of work :(
- How do we find what we should cut?
- Profiling and code inspection.

SCOOP RUNTIME: AIMED AT SPEED

- Lots of atomic operations (compare and swap, atomic increment, etc.)
- Implemented records using arrays and offset indices to avoid the object overhead.
- Avoids synchronization primitives (locks, semaphores).

Resource acquisition

```
from
   acquired := False
until
   acquired
loop
   old_val :=
      cas (resrc, free, this)
   if old_val = free then
      acquired := True
   else
      yield
   end
end
```

- This code avoids a mutex!
- Mutexes are slow because they may trigger context switch...
- ... but *yield* also triggers a context switch.
- This code is basically a mutex but never sleeps and eats a core (it spins until it acquires the resource).
- Replacing it with a traditional mutex helps.

Resource acquisition

```
from
   acquired := False
until
   acquired
loop
   old_val :=
      cas (resrc, free, this)
   if old_val = free then
      acquired := True
   else
      yield
   end
end
```

- This code avoids a mutex!
- Mutexes are slow because they may trigger context switch...
- ... but *yield* also triggers a context switch.
- This code is basically a mutex but never sleeps and eats a core (it spins until it acquires the resource).
- Replacing it with a traditional mutex helps.

Resource acquisition

```
from
   acquired := False
until
   acquired
loop
   old_val :=
      cas (resrc, free, this)
   if old_val = free then
      acquired := True
   else
      yield
   end
end
```

- This code avoids a mutex!
- Mutexes are slow because they may trigger context switch...
- ... but *yield* also triggers a context switch.
- This code is basically a mutex but never sleeps and eats a core (it spins until it acquires the resource).
- Replacing it with a traditional mutex helps.

Resource acquisition

```
from
   acquired := False
until
   acquired
loop
   old_val :=
      cas (resrc, free, this)
   if old_val = free then
      acquired := True
   else
      yield
   end
end
```

- This code avoids a mutex!
- Mutexes are slow because they may trigger context switch...
- ... but *yield* also triggers a context switch.
- This code is basically a mutex but never sleeps and eats a core (it spins until it acquires the resource).
- Replacing it with a traditional mutex helps.

Resource acquisition

```
from
   acquired := False
until
   acquired
loop
   old_val :=
      cas (resrc, free, this)
   if old_val = free then
      acquired := True
   else
      yield
   end
end
```

- This code avoids a mutex!
- Mutexes are slow because they may trigger context switch...
- ... but *yield* also triggers a context switch.
- This code is basically a mutex but never sleeps and eats a core (it spins until it acquires the resource).
- Replacing it with a traditional mutex helps.

Moving forward

- I started to rewrite the slow parts as best I could; these did lead to some improvements.
- However the changes are untestable: the code currently deadlocks and has some other bugs so I have no way to figure out if it's more or less broken after my changes.
- Develop a "laboratory" where I can try out ideas: proof-of-concept in C++.

Moving forward

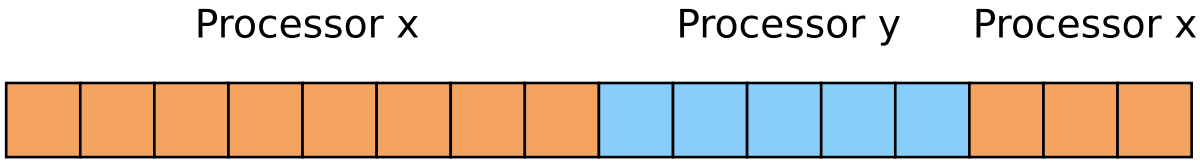
- I started to rewrite the slow parts as best I could; these did lead to some improvements.
- However the changes are untestable: the code currently deadlocks and has some other bugs so I have no way to figure out if it's more or less broken after my changes.
- Develop a "laboratory" where I can try out ideas: proof-of-concept in C++.

Moving forward

- I started to rewrite the slow parts as best I could; these did lead to some improvements.
- However the changes are untestable: the code currently deadlocks and has some other bugs so I have no way to figure out if it's more or less broken after my changes.
- Develop a "laboratory" where I can try out ideas: proof-of-concept in C++.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

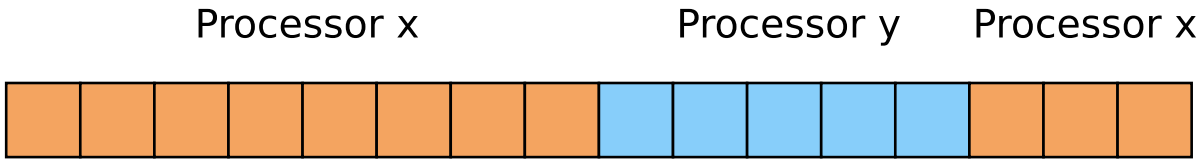


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

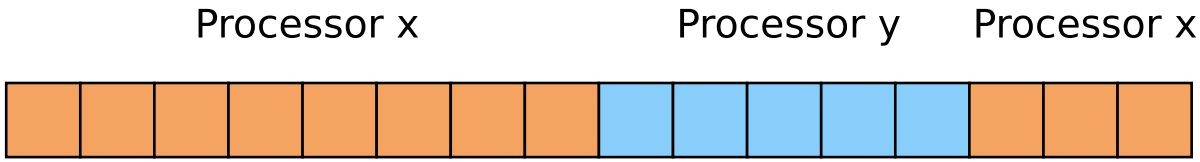


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

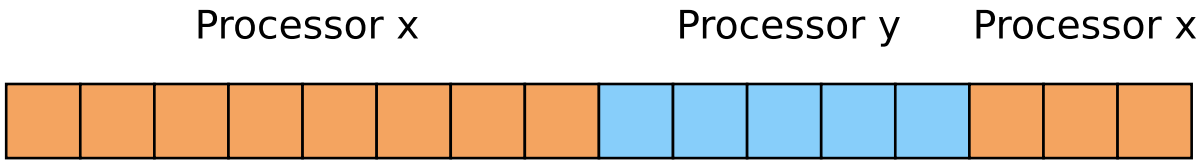


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

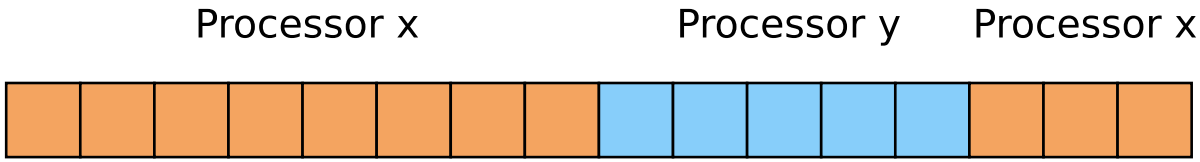


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

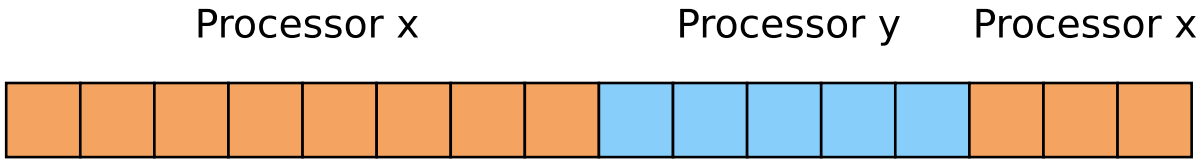


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model

A SCOOP processor is basically a work-queue which other processors have exclusive access to.

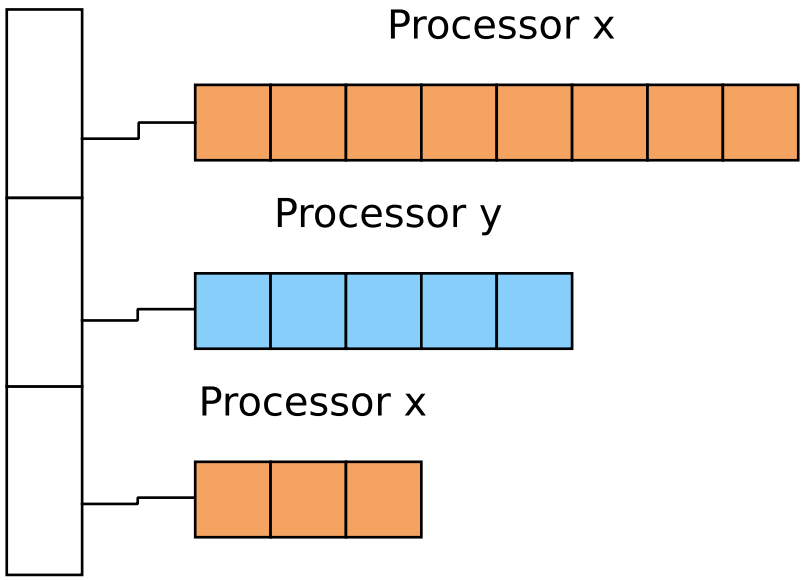


Naïve implementation STL queue wrapped in a monitor: 404s
Non-blocking TBB queue: 180s
Blocking TBB queue: 108s

Is this any faster than what we have now?
Sort of, EiffelStudio 7.1: 33s but EiffelStudio 7.2: 155s.

SCOOP implementation model 2

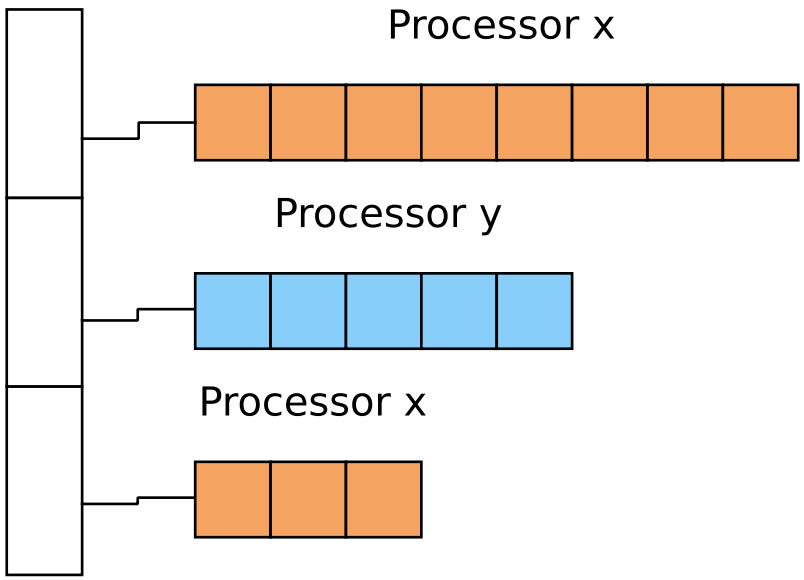
Lock contention can be reduced if there is no memory to contend over: use a queue of queues



Queue of queues: 15s.
With limited dequeue spinning: 3.7s
Don't use the queue for "second" queries: 2.0s

SCOOP implementation model 2

Lock contention can be reduced if there is no memory to contend over: use a queue of queues



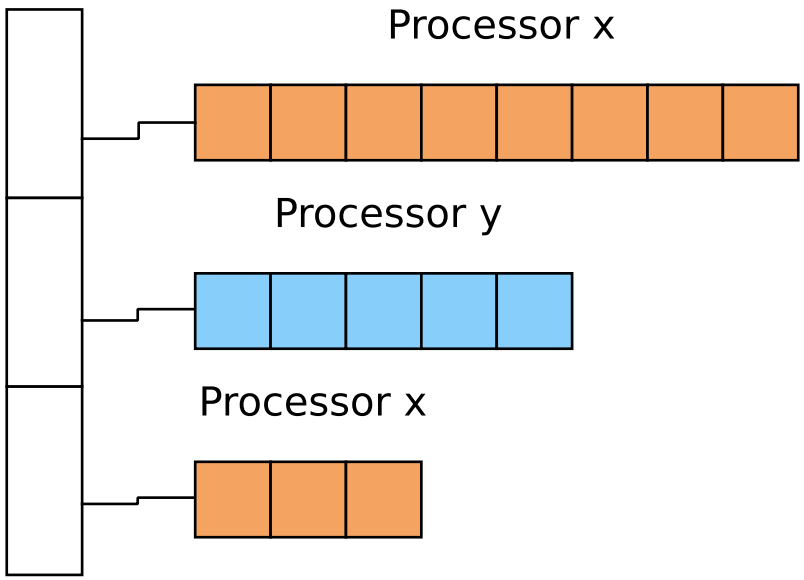
Queue of queues: 15s.

With limited dequeue spinning: 3.7s

Don't use the queue for "second" queries: 2.0s

SCOOP implementation model 2

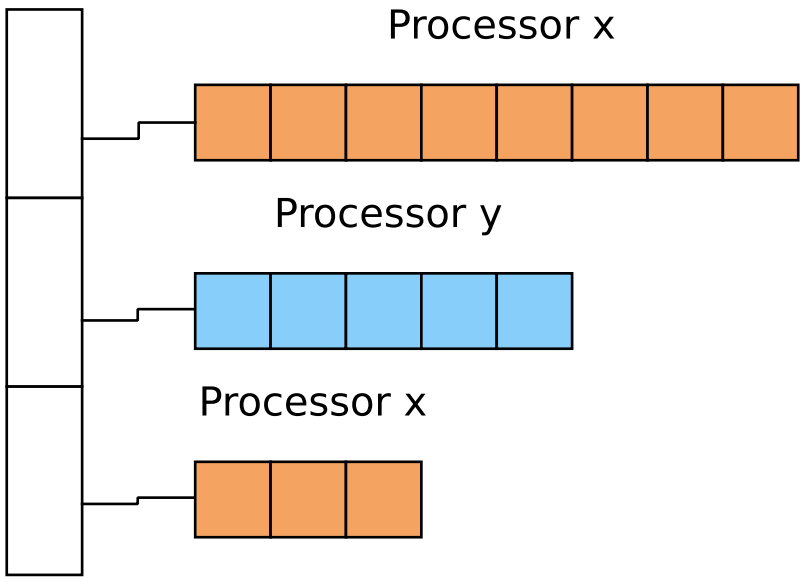
Lock contention can be reduced if there is no memory to contend over: use a queue of queues



Queue of queues: 15s.
With limited dequeue spinning: 3.7s
Don't use the queue for "second" queries: 2.0s

SCOOP implementation model 2

Lock contention can be reduced if there is no memory to contend over: use a queue of queues

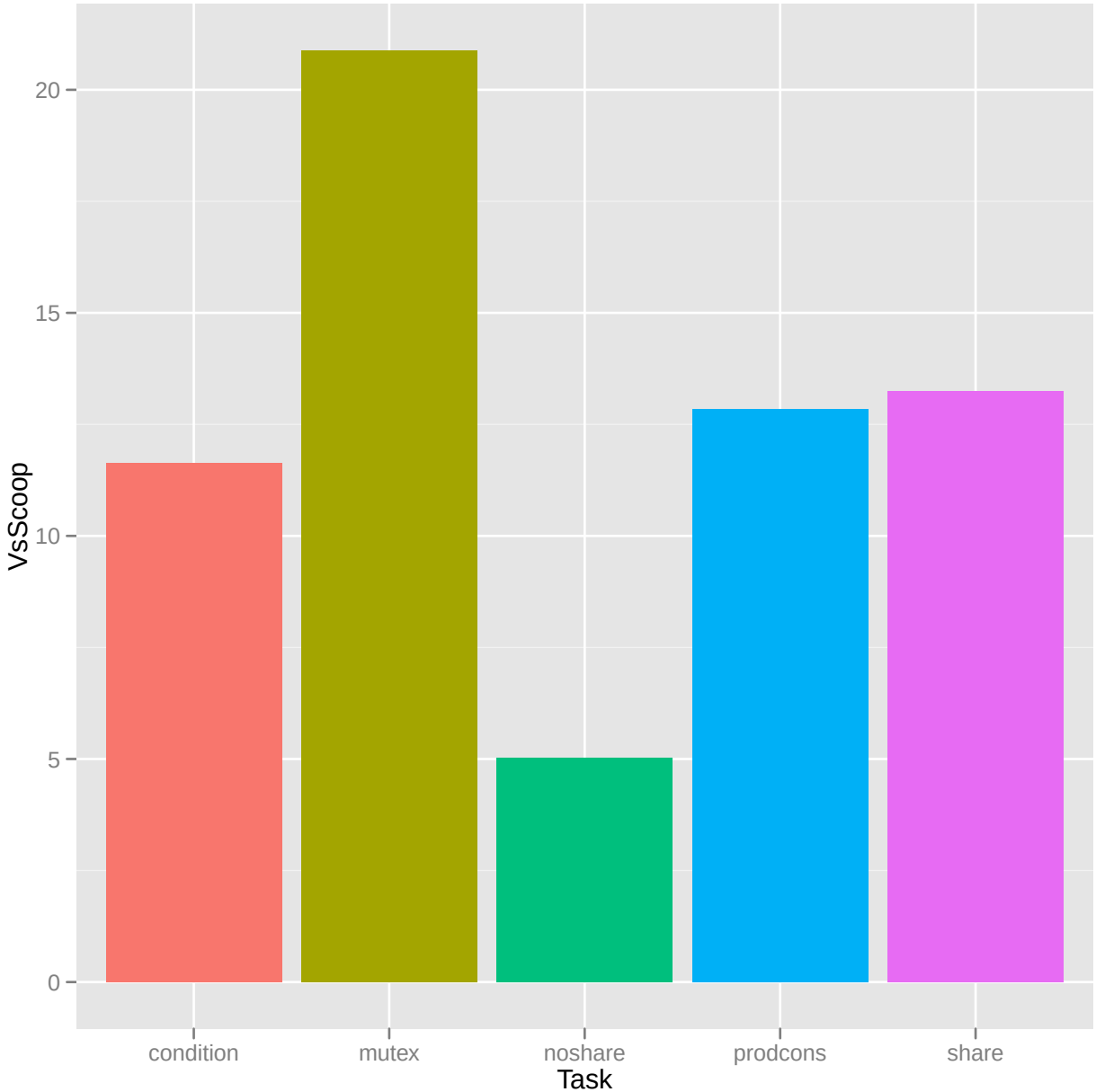


Queue of queues: 15s.
With limited dequeue spinning: 3.7s
Don't use the queue for "second" queries: 2.0s

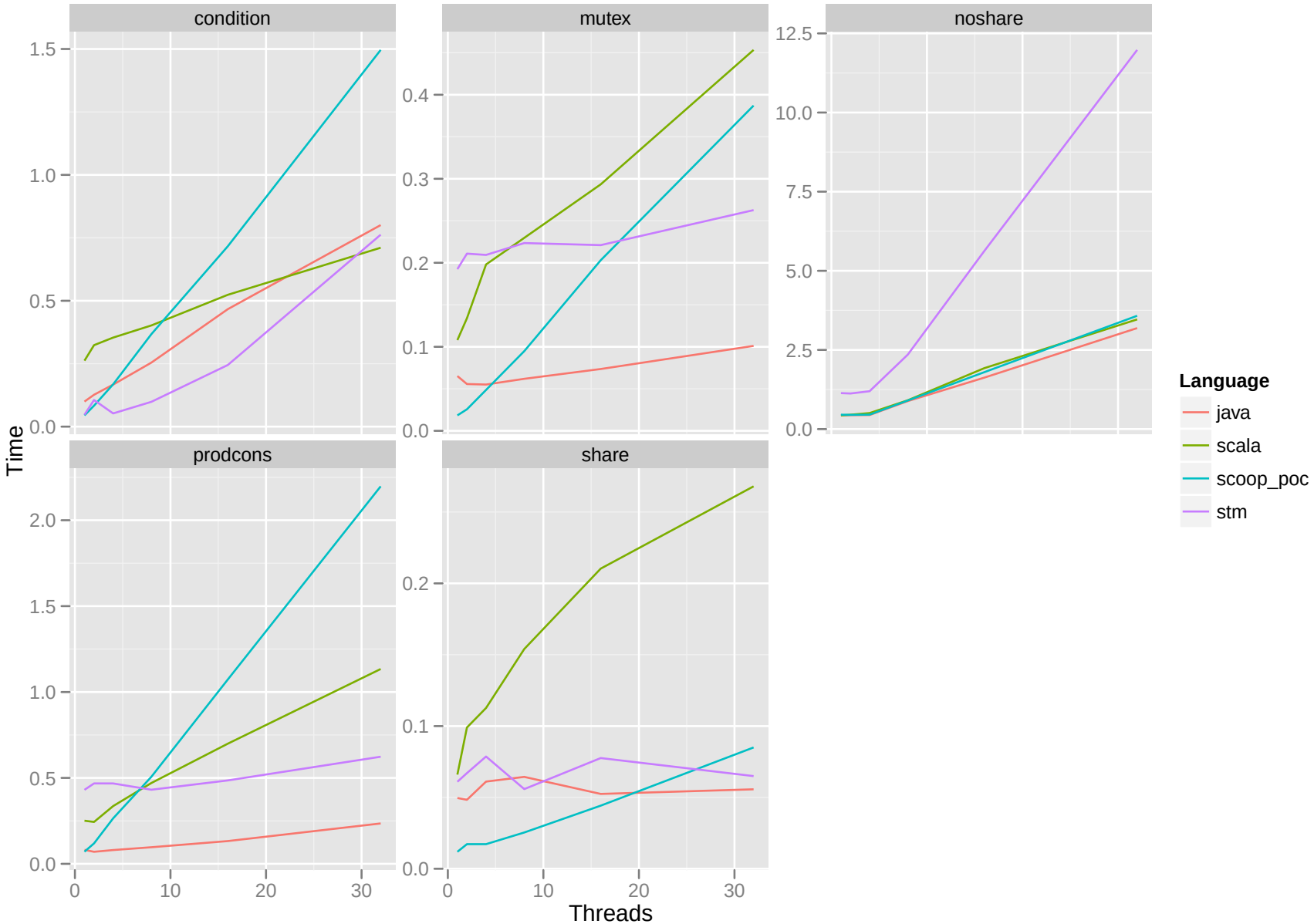
Putting them all together

Approach	Time
Blocking (STL wrapper)	404s
TBB Non-blocking (sleep)	180s
ES 7.2	155s
TBB Blocking	108s
ES 7.1	33s
Queue of queues	15s
QoQ + Spinning	3.7s
QoQ + Local queries	2.0s

EiffelStudio 7.1 vs. proof of concept



New benchmark results



Implementation comments

- The processor "runtime" is small: about 150 lines of C++.
- Seems to be the right target abstraction; almost no usage of explicit synchronization.
- Only one dependency, TBB.
- Probably still some room to fine tune, i.e. use specialized queues where possible, like single-producer/consumer queues.