Postgres PROFESSIONAL

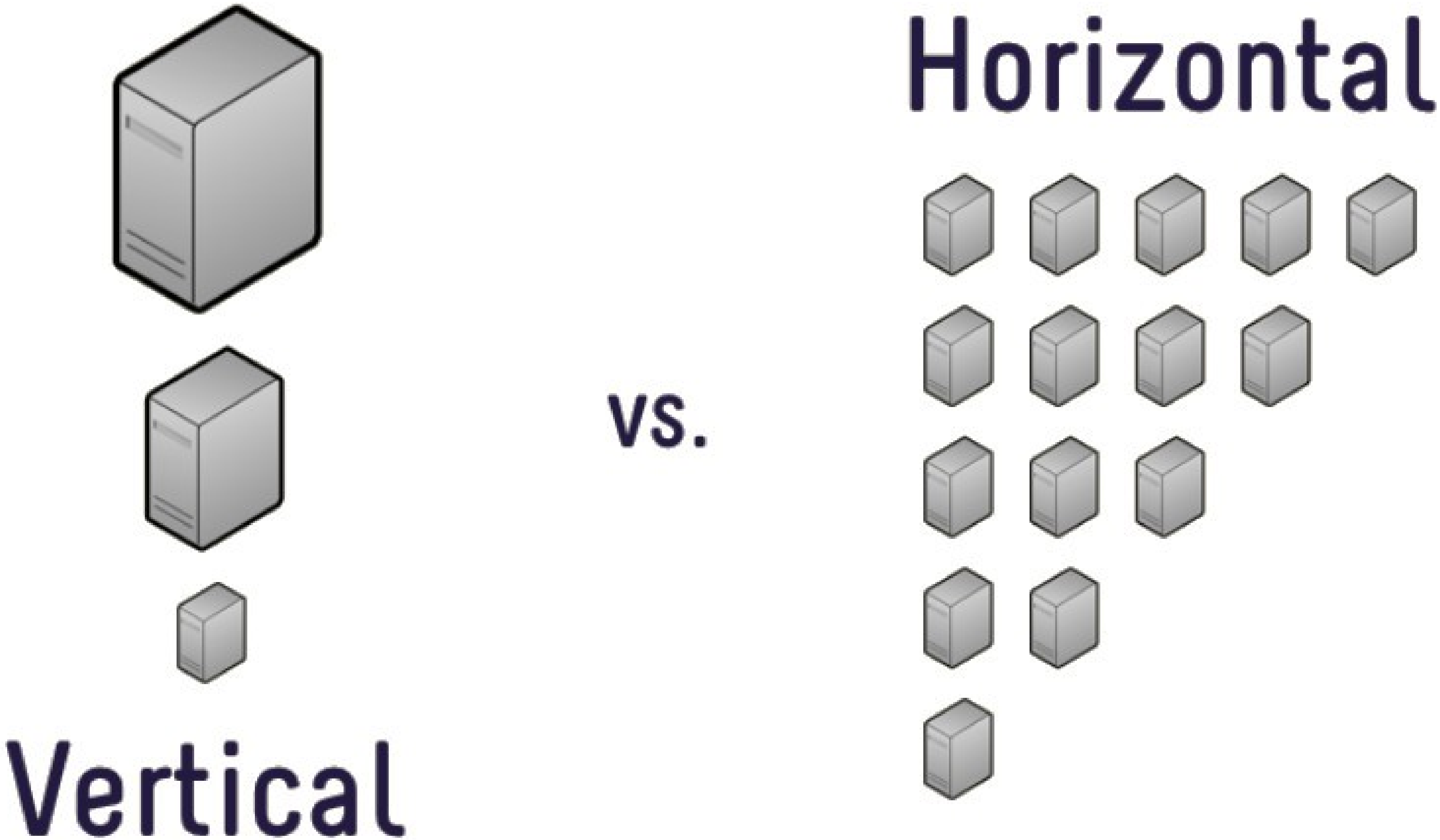
PostgreSQL Scalability

Dmitry Vasilyev

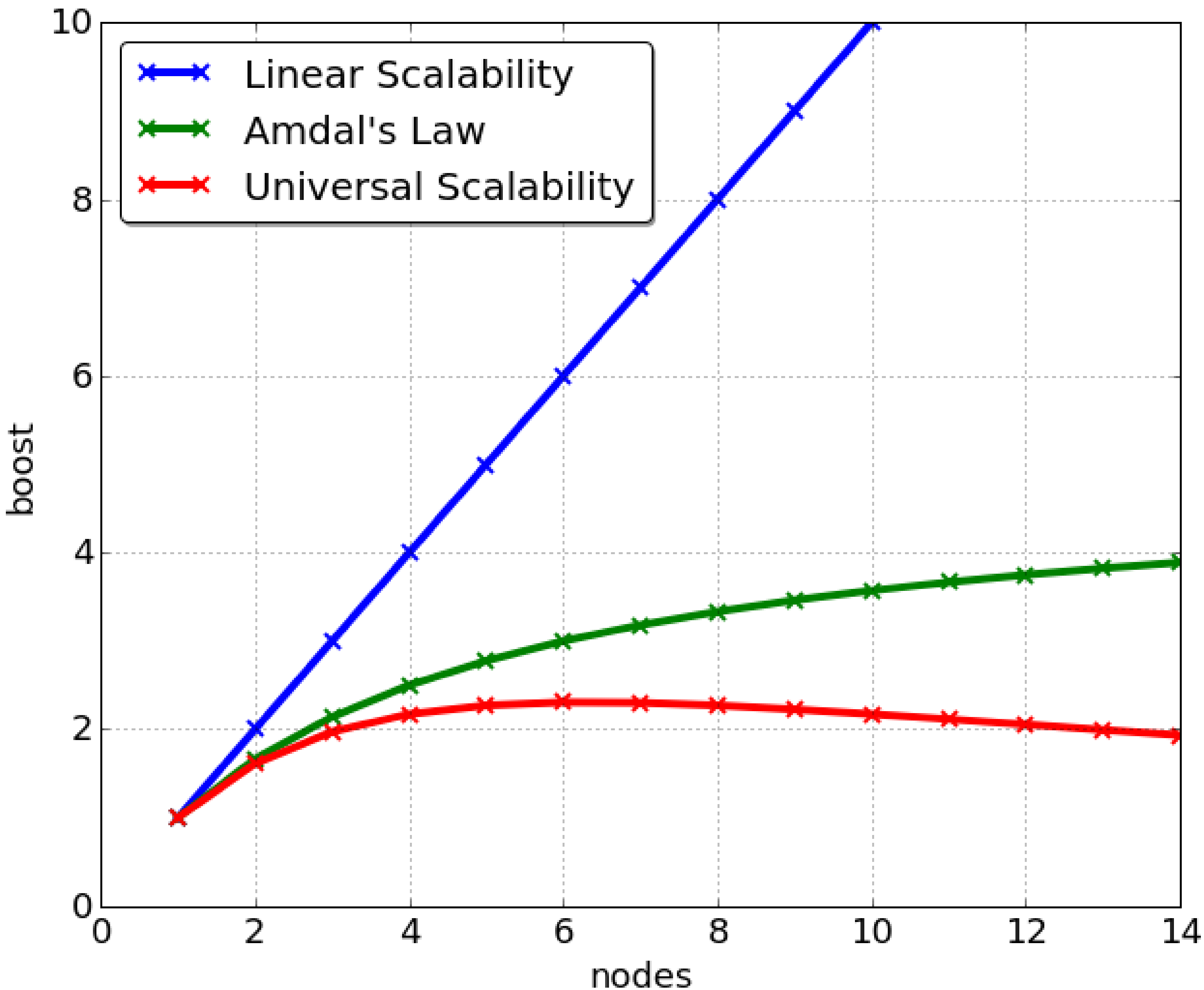
Scalability

- ability to increase perfomance by adding resources.

Scalability



Scalability Laws



Linear Scalability

$$B(N) = c * N$$

B — Boost

N — Number of processors

Amdahl's Law

$$B(N) = \frac{N}{1 + a * (N - 1)}$$

a - fraction that can be parallelized

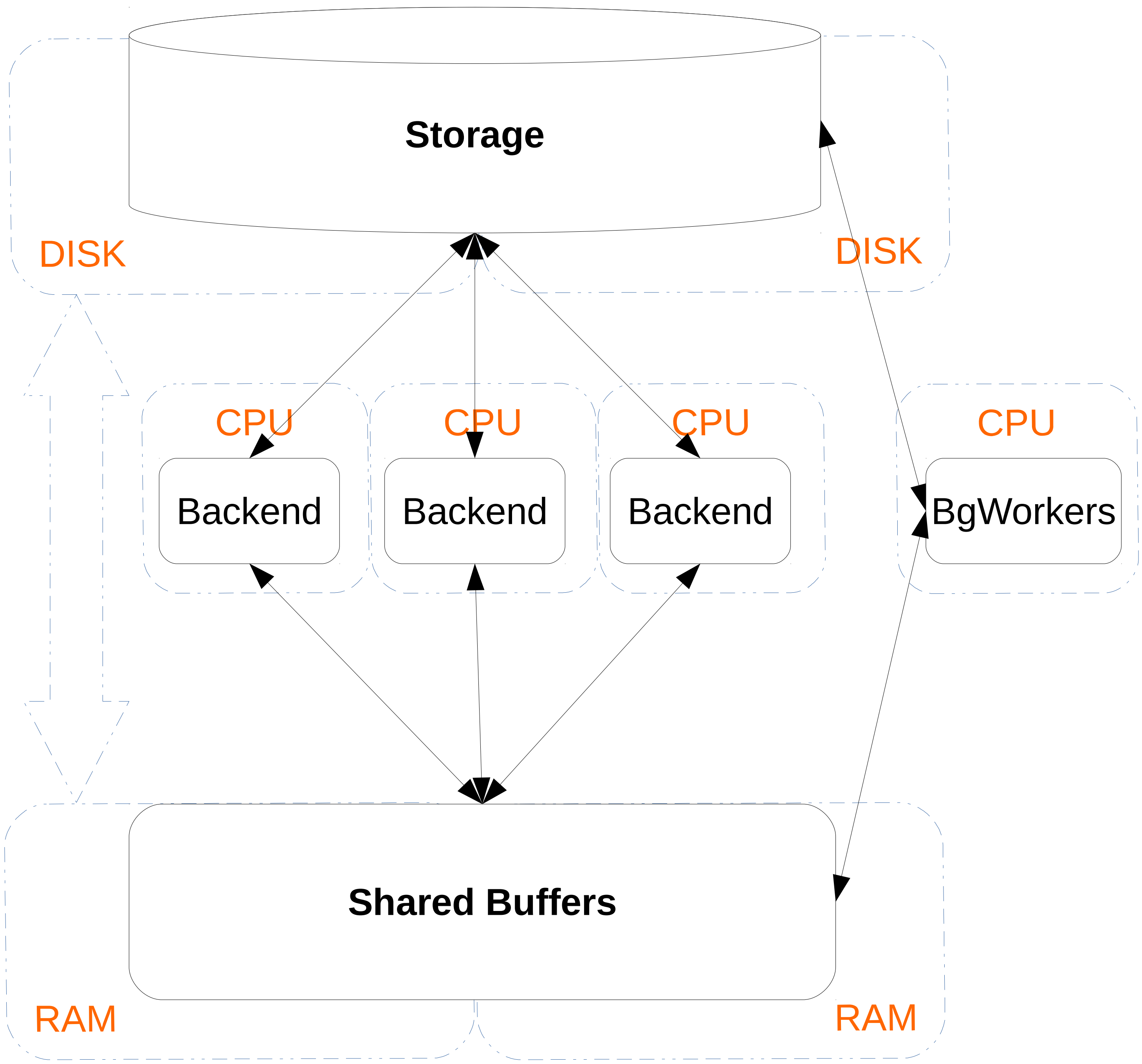
Universal Scalability Law

$$B(N) = \frac{N}{1 + a*(N-1) + b*N*(N-1)}$$

a - fraction that can be parallelized

b - synchronized fraction

PostgreSQL


Storage
DISK
DISK
CPU
CPU
CPU
CPU
Backend
Backend
Backend
BgWorkers
Shared Buffers
RAM
RAM

One instance of PostgreSQL

N = CPU

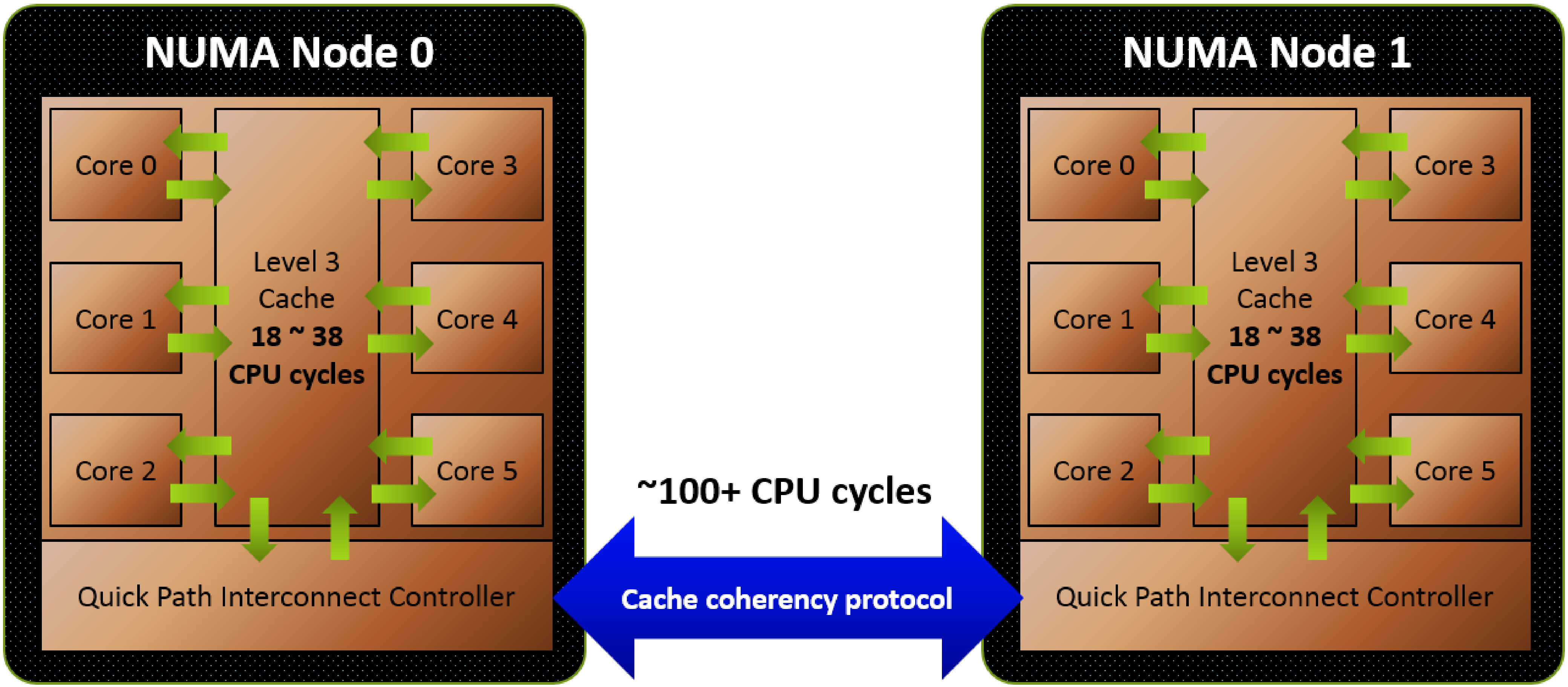
a = waiting or queueing for shared resources

b = delay for data to become consistent

$1 \gg a,b > 0$

Waiting for shared resources

Delay for data to become consisten



Fraction that can be parallelized

Critical section

- runtime code part making exclusive access to sharing resource (data or device)

PostgreSQL Locks

- SpinLock
- Lightweight Lock (LWLock)
- Heavyweight lock (HWLock)

Also: Row-Level, Predicate, Advisory Locks

SpinLock

- Fast (very short locks)
- Exclusive only
- No deadlock checks
- No queuing

LWLocks

- Fast
- Shared / Exclusive locks
- No deadlock checks
- Mostly uses SpinLocks before PG 9.5

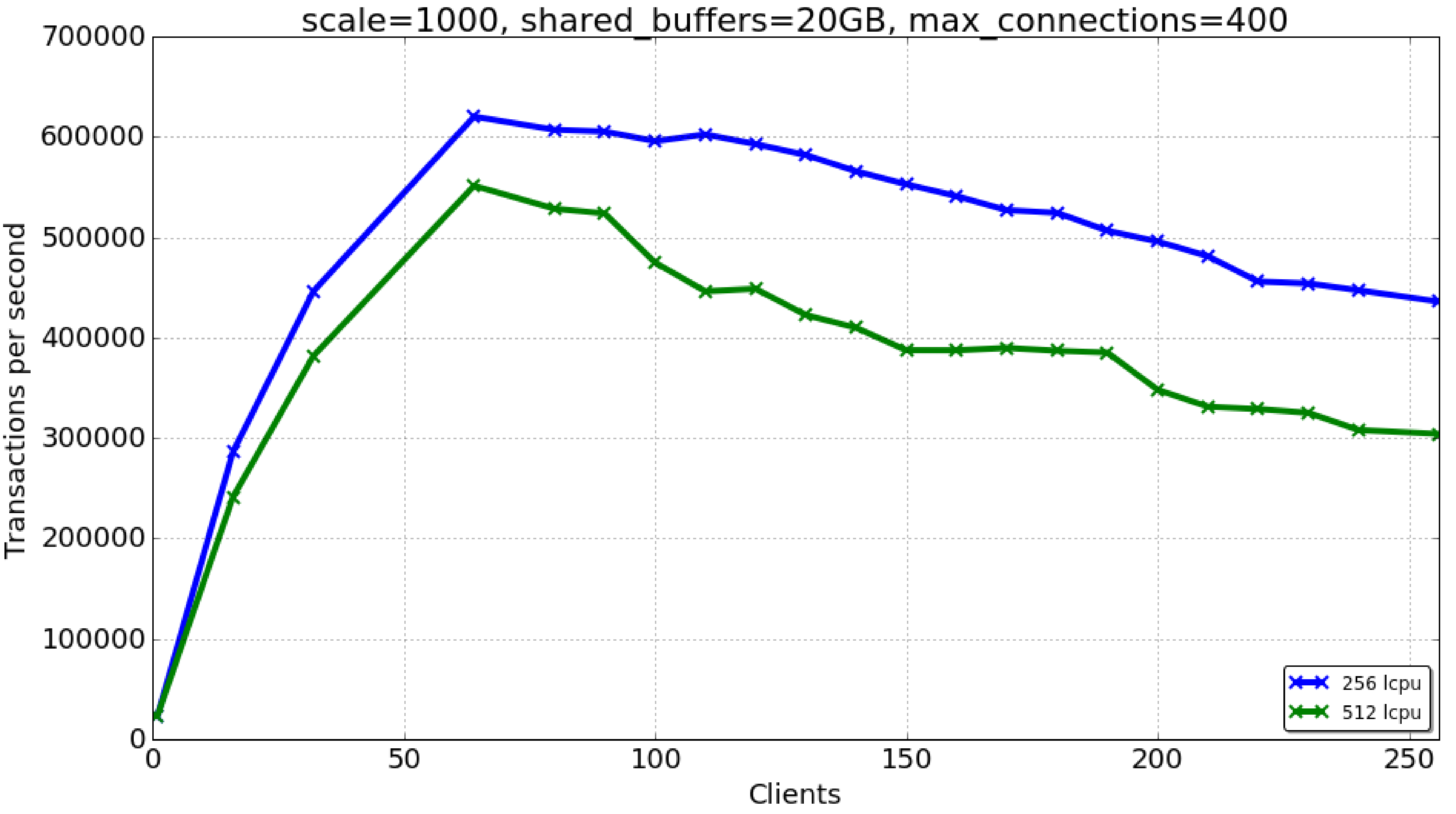
HWLocks

- Works with objects in DB
- Complex locking modes
- Deadlock checks

IBM E880

- 2 x 2U Nodes
- Nodes connected via «HyperConnect»
- One Node: 4 Socket
- One Socket: 8 CORE
- One CORE: 8 SMT

256 vs 512 LCPU



perf top *

- 33.48% postgres [.] **s_lock**
- 2.51% postgres [.] GetSnapshotData
- 1.82% postgres [.] PinBuffer

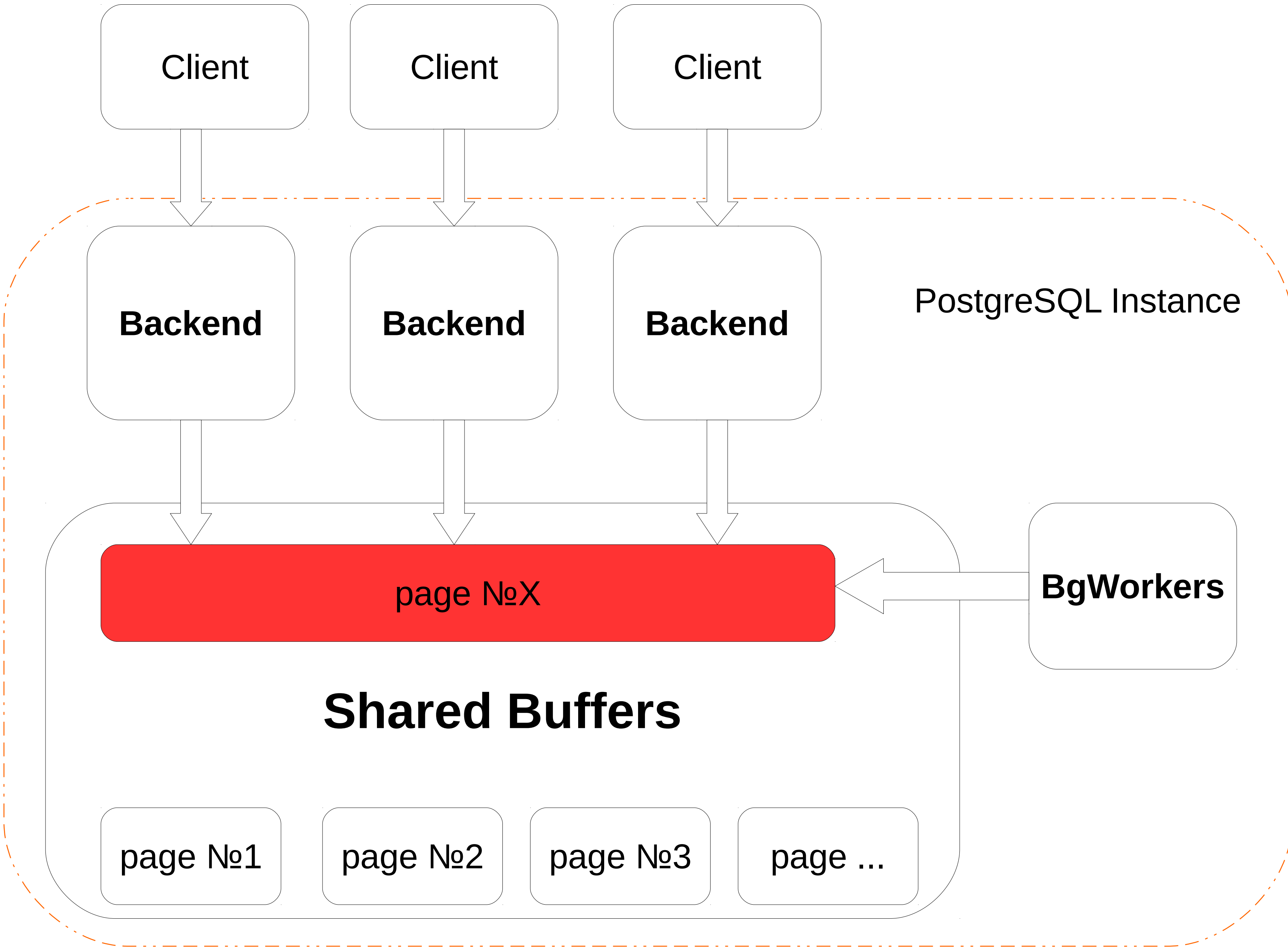
* Hanging out in critical session increasing with growing number of clients and CPU

gdb *

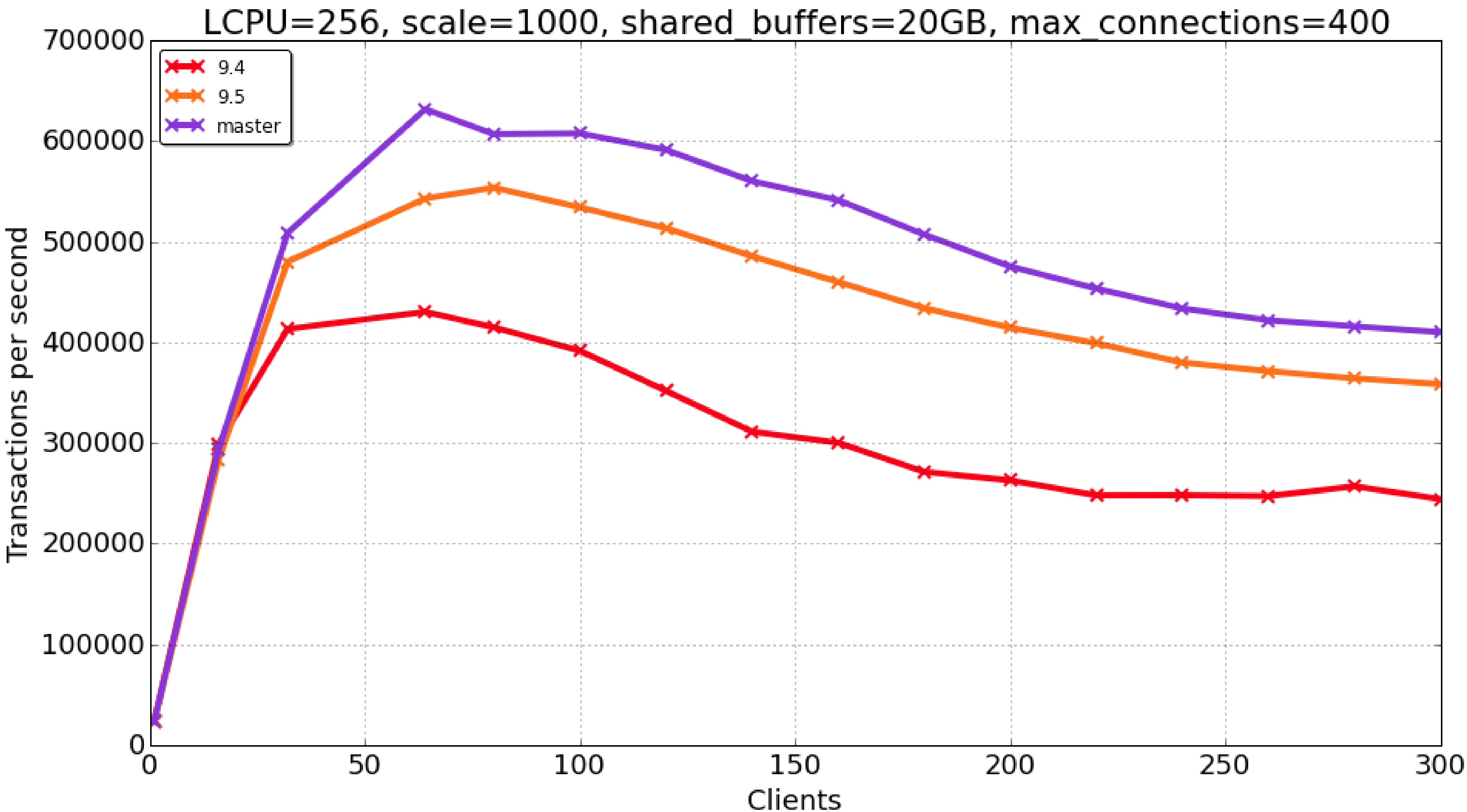
- #0 0x00003fffac40a858 in ___newselect_nocancel () from /lib64/power8/libc.so.6
- #1 0x00000000106105f0 in pg_usleep (microsec=<optimized out>) at pgsleep.c:53
- #2 0x00000000103e5f18 in s_lock (lock=0x3fe607980be0, file=0x10718398 "bufmgr.c", line=<optimized out>) at s_lock.c:110
- #3 0x00000000103aea10 in **UnpinBuffer** (buf=0x3fe607980bc0, fixOwner=1 '\001') at bufmgr.c:1540
- #4 0x00000000103b4910 in ReleaseAndReadBuffer (buffer=<optimized out>, relation=0x3fe6067073e0, blockNum=<optimized out>) at bufmgr.c:1401

* One threaded perf performance is insufficient

Yet another look at an architecture

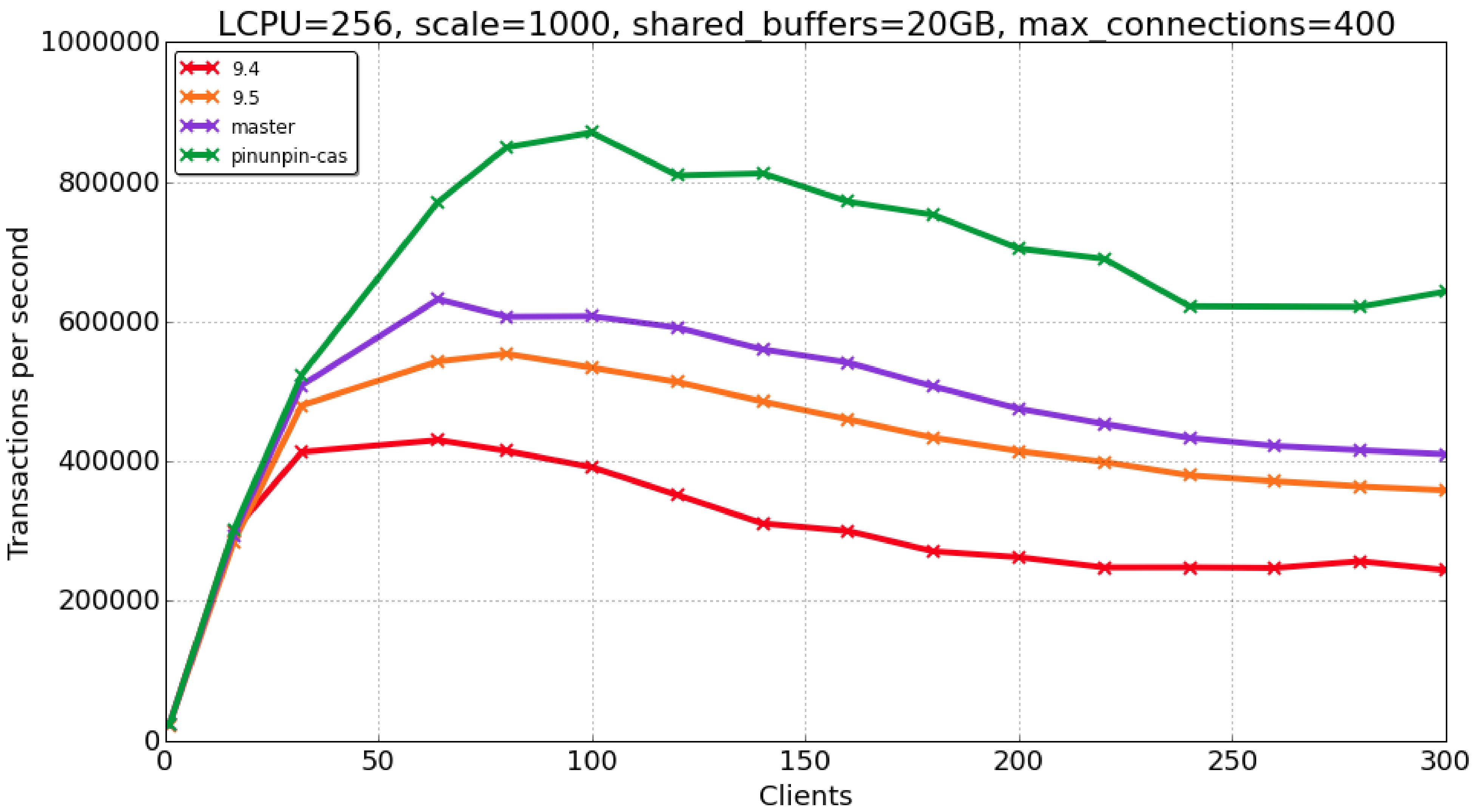


PostgreSQL 9.4 vs 9.5 vs master



9.4 LWLocks uses SpinLock
>= 9.5 LWLocks uses Atomic

Andres Freund patches



PinBuffer uses CAS
UnPinBuffer uses Atomic

Results of optimisation

After:

- 33.48% postgres [.] s_lock
- 2.51% postgres [.] GetSnapshotData
- 1.82% postgres [.] PinBuffer

Before:

- 13.75% postgres [.] GetSnapshotData
- 4.88% postgres [.] AllocSetAlloc
- 2.47% postgres [.] LWLockAcquire

Show Me Asm!

Compare-And-Set:

```
# Input parameters:
#  r3 – old, r4 – new value
#  r5 – atomic variable address

.L1: lwarx 9,0,5
     cmpw 0,9,3
     bne- 0,.L2
     stwcx. 4,0,5
     bne- 0,.L1
.L2: isync
```

Atomic Add:

```
# Input parameters:
#  r3 – increment
#  r5 – atomic variable address

.L1: lwarx 9,0,5
     add 9,9,3
     stwcx. 9,0,5
     bne- 0,.L1
     isync
```

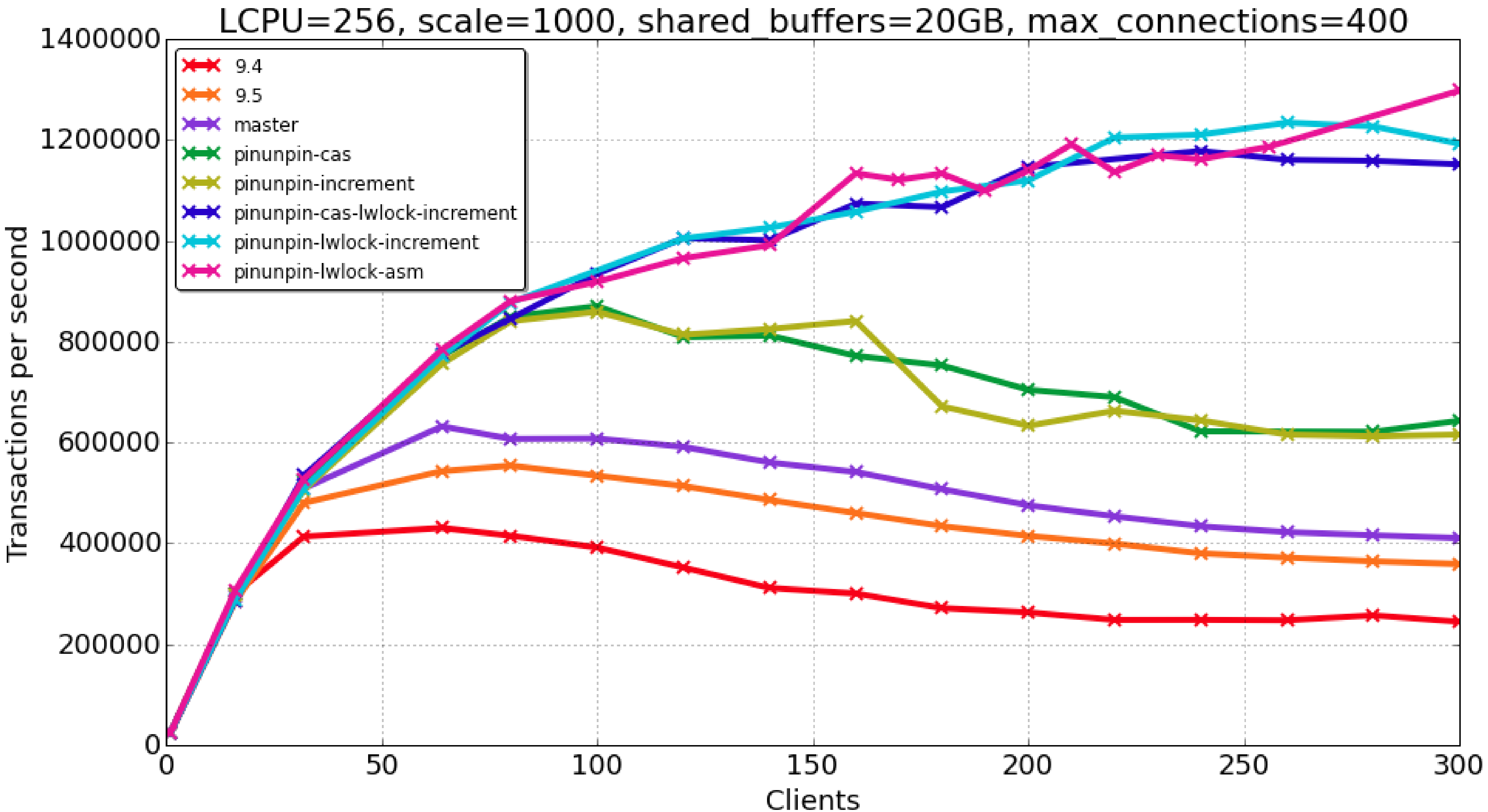
What have we done?

Try two ideas:

1. Develop performance critical functions such as PinBuffer and LWLockAttemptLock with ASM inlines.

2. Alternative idea is to use atomic increment and develop “optimistic atomic lock”. In PinBuffer and LWLockAttemptLock is possible to do atomic increment of “state variable” and then to check had we rights for such change or had not for actual set value. In case we had no rights to change state in this way — cancel atomic increment change operation.

PostgresPro patches



pinunpin-cas – PinBuffer uses CAS
pinunpin-increment – PinBuffer optimistic Atomic
pinunpin-cas-lwlock-increment – PinBuffer uses CAS, LWLockAttemptLock uses optimistic Atomic
pinunpin-lwlock-increment – PinBuffer and LWLockAttemptLock optimistic Atomic
pinunpin-lwlock-asm – PinBuffer and LWLockAttemptLock written on asm

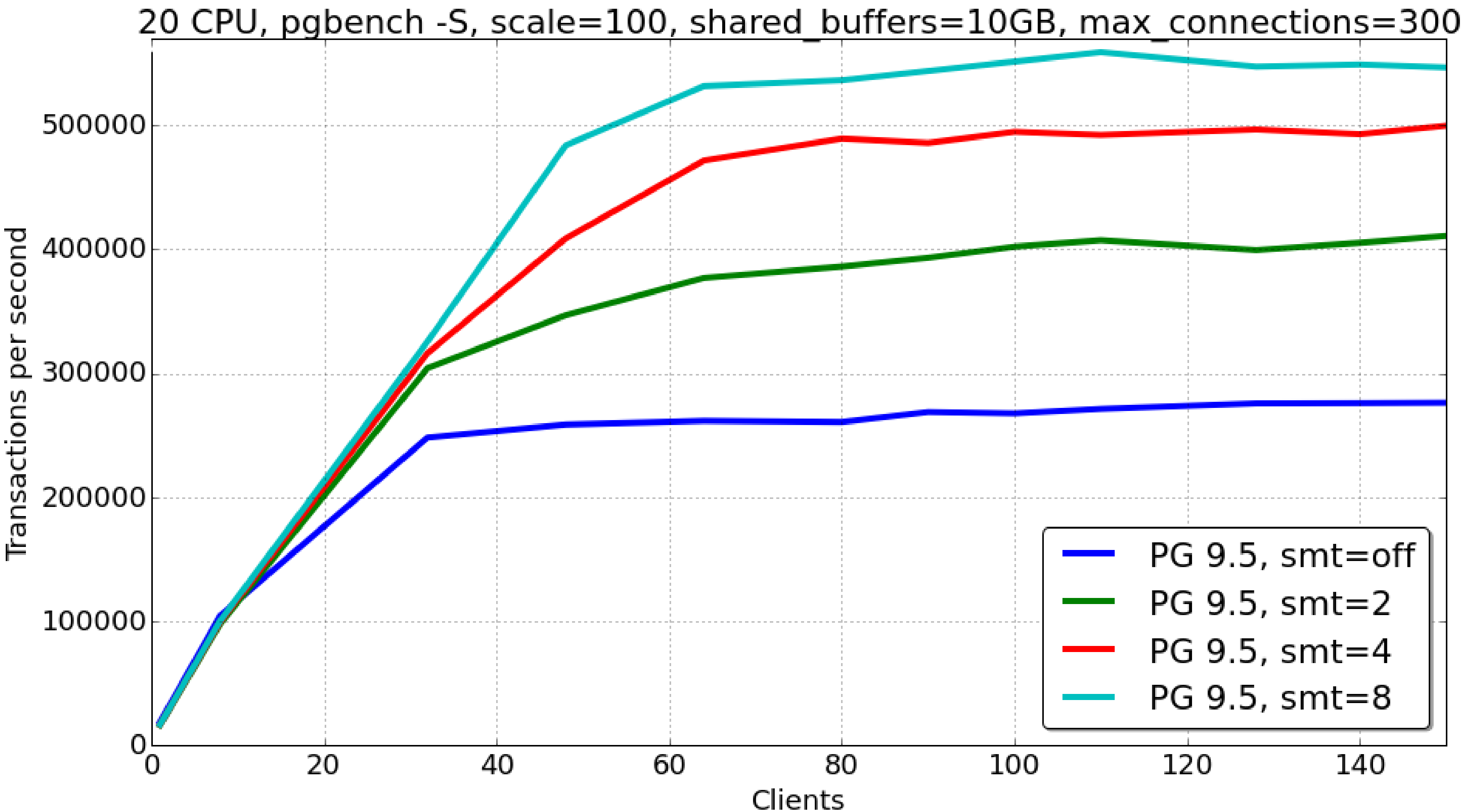
PostgresPro patches

- Replacement of low-level locks system in PostgreSQL was produced in experimental mode so it approximately doubles the scalability of the number of processor cores and thus increase the availability of the PostgreSQL effective use on a very large Power8 servers.
- Last ASM patch allowed us to reach 2 million tps total performance on 48 cores Power8. This was achieved on the two copies of PostgreSQL running concurrently on the different compute nodes.

A - Fraction that can be parallelized

PostgreSQL 9.4	4.3 %
PostgreSQL 9.6	2.3 %
PostgresPro patches	1.6 %

PostgreSQL and Linux on Power



PostgreSQL and Linux on Power

- GCC Advanced Toolchain showed better performance then GCC
- XLC + PostgreSQL - optimisations are not working.

LWLock monitoring

pg_stat_wait:

- Profiling
- History
- Tracing query

Try this!

https://github.com/postgrespro/postgres

* monitoring for 9.4: waits_monitoring_94

Thank you for your time!

Questions

Contacts:

info@postgrespro.ru

See you later!

Links

http://habrahabr.ru/company/postgrespro/blog/

https://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf

http://www.postgresql.eu/events/sessions/pgconfeu2015/session/1080-scaling-up-postgresql/

http://www.slideshare.net/chris1adkin/super-scaling-singleton-inserts-53947279

http://habrahabr.ru/post/190862/

http://www.slideshare.net/rivitli/waits-monitoring-in-postgresql