M. Pueschel, T. Ben-Nun

Lecture 6: Fast practical locks, lock-free, consensus, and scalable locks

Review of last lecture

- **Memory models in practical parallel programming**
  - Synchronized programming
  - How locks synchronize processes *and* memory!
    How to code in C++ and Java

- **Proving program correctness**
  - Pre-/postconditions – sequential
  - Lifting to parallel
    How to prove locked programs correct (nearly trivial)

- **Lock implementation**
  - Proof of correctness (using read/write histories, program and visibility orders)
    With x86 memory model!
  - Peterson lock
  - Lock performance
    Simple x86 – how much does memory model correctness cost?

Recap: Spin-Locks

- **Two-Thread Locks**
  - First “Lock”
  - LockOne
  - LockTwo
  - Peterson Lock
- **N-Thread Locks**
  - Lamport’s Bakery Algorithm

```
volatile int flag

void lock() {
  while(flag);
  flag =
}

void unlock() {
  flag = 0;
}
```

No mutual exclusion

```
volatile int flag[2];

void lock() {
  int j = 1 - tid;
  flag[tid] = true;
  while (flag[j]) {} // wait
}

void unlock() {
  flag[tid] = false;
}
```

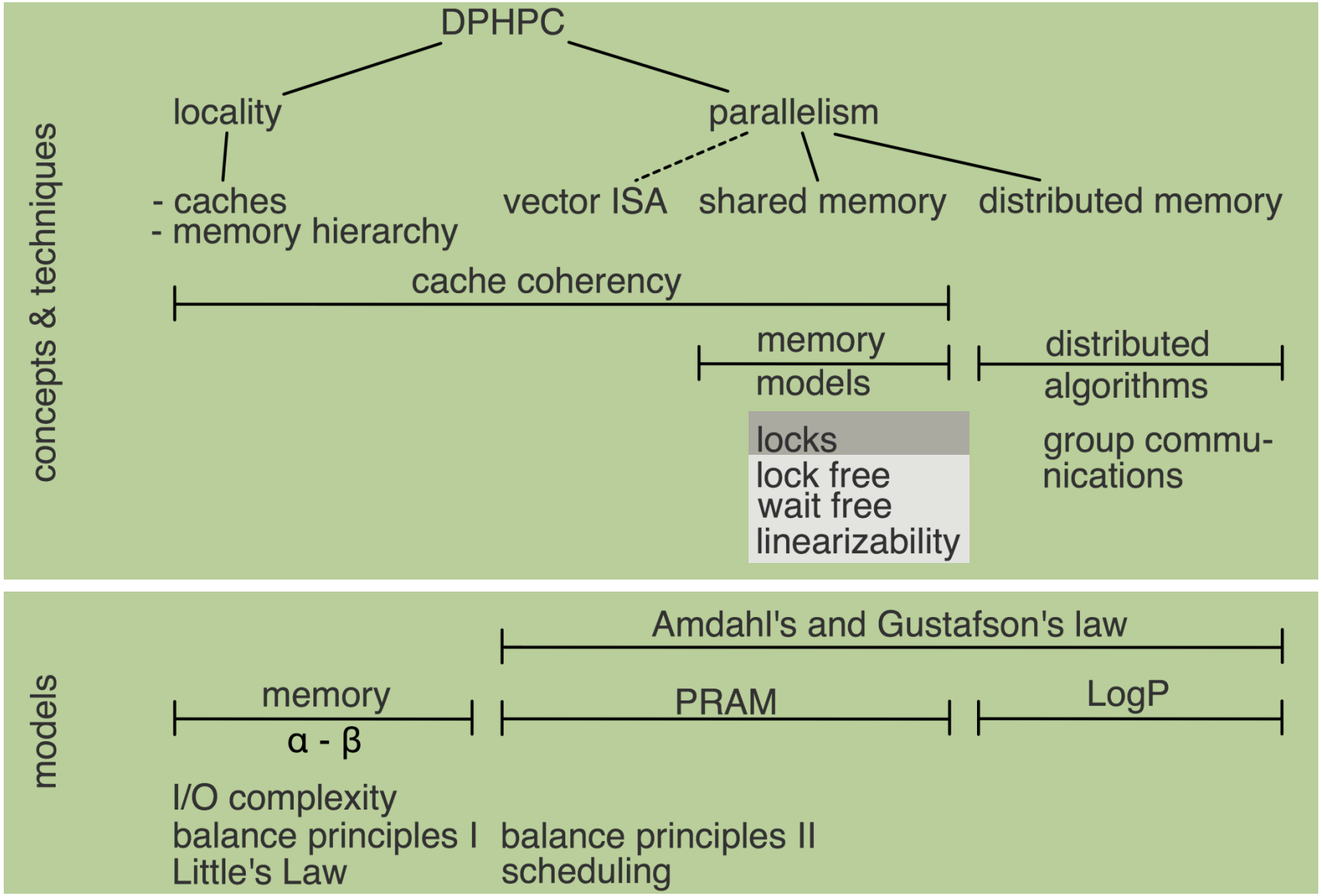
✓ Mutual exclusion
× Deadlock if two attempts overlap

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```

✓ Mutual exclusion
× Deadlock if two attempts **do not** overlap

```
volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
  while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}
public void unlock() {
  flag[tid] = 0;
}
```

DPHPC Overview



Goals of this lecture

- **Scientific Benchmarking**

- **Fast and scalable practical locks!**
  - Based on atomic operations
  - Why do we need atomic operations?

- **Recap lock-free and wait-free programming**
  - Proof that wait-free consensus is impossible without atomics

    Valence argument: a proof technique similar to showing that atomics are needed for locks

- **Locks in practical setting**
  - How to block?
  - When to block?
  - How long to block?

    Simple proof of competitiveness

Interlude: Scientific integrity – or how to report benchmark results?

1991 – the classic!

2012 – the shocking

2013 – the extension

Fooling the Masses with Performance Results: Old Classics & Some New Ideas

Gerhard Wellein[1,2], Georg Hager[2]

[1]Department for Computer Science
[2]Erlangen Regional Computing Center
Friedrich-Alexander-Universität Erlangen-Nürnberg

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Scientific Benchmarking of Parallel Computing Systems

Twelve ways to tell the masses when reporting performance results

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Roberto Belli
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
bellir@inf.ethz.ch

ABSTRACT

Measuring and reporting performance of parallel computers constitutes the basis for scientific advancement of high-performance computing (HPC). Most scientific reports show performance improvements of new techniques and are thus obliged to ensure reproducibility or at least interpretability. Our investigation of a stratified sample of 120 papers across three top conferences in the field shows that the state of the practice is lacking. For example, it is often unclear if reported improvements are deterministic or observed by chance. In addition to distilling best practices from existing work, we propose statistically sound analysis and reporting techniques and simple guidelines for experimental design in parallel computing and codify them in a portable benchmarking library. We aim to improve the standards of reporting research results and initiate a discussion in the HPC field. A wide adoption of our minimal set of rules will lead to better interpretability of performance results and improve the scientific culture in HPC.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Benchmarking, parallel computing, statistics, data analysis

1. INTRODUCTION

Correctly designing insightful experiments to measure and report performance numbers is a challenging task. Yet, there is surprisingly little agreement on standard techniques for measuring, reporting, and interpreting computer performance. For example, common questions such as "How many iterations do I have to run per measurement?", "How many measurements should I run?", "Once I have all data, how do I summarize it into a single number?", or "How do I measure time in a parallel system?" are usually answered based on intuition. While we believe that an expert's intuition is most often correct, there are cases where it fails and invalidates expensive experiments or even misleads us. Bailey [3] illustrates this in several common but misleading data reporting patterns that he and his colleagues have observed in practice.

Reproducing experiments is one of the main principles of the scientific method. It is well known that the performance of a computer program depends on the application, the input, the compiler, the runtime environment, the machine, and the measurement methodology [20, 43]. If a single one of these aspects of *experimental design* is not appropriately motivated and described, presented results can hardly be reproduced and may even be misleading or incorrect.

The complexity and uniqueness of many supercomputers makes reproducibility a hard task. For example, it is practically impossible to recreate most hero-runs that utilize the world's largest machines because these machines are often unique and their software configurations changes regularly. We introduce the notion of *interpretability*, which is weaker than reproducibility. *We call an experiment interpretable if it provides enough information to allow scientists to understand the experiment, draw own conclusions, assess their certainty, and possibly generalize results.* In other words, interpretable experiments support sound conclusions and convey precise information among scientists. Obviously, every scientific paper should be interpretable; unfortunately, many are not.

For example, reporting that an High-Performance Linpack (HPL) run on 64 nodes (N=314k) of the Piz Daint system during normal operation (cf. Section 4.1.2) achieved 77.38 Tflop/s is hard to interpret. If we add that the theoretical peak is 94.5 Tflop/s, it becomes clearer, the benchmark achieves 81.8% of peak performance. But is this true for every run or a typical run? Figure 1

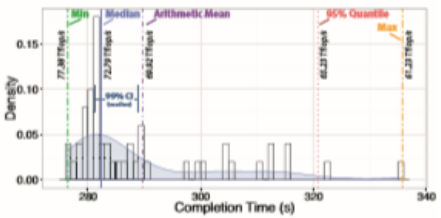
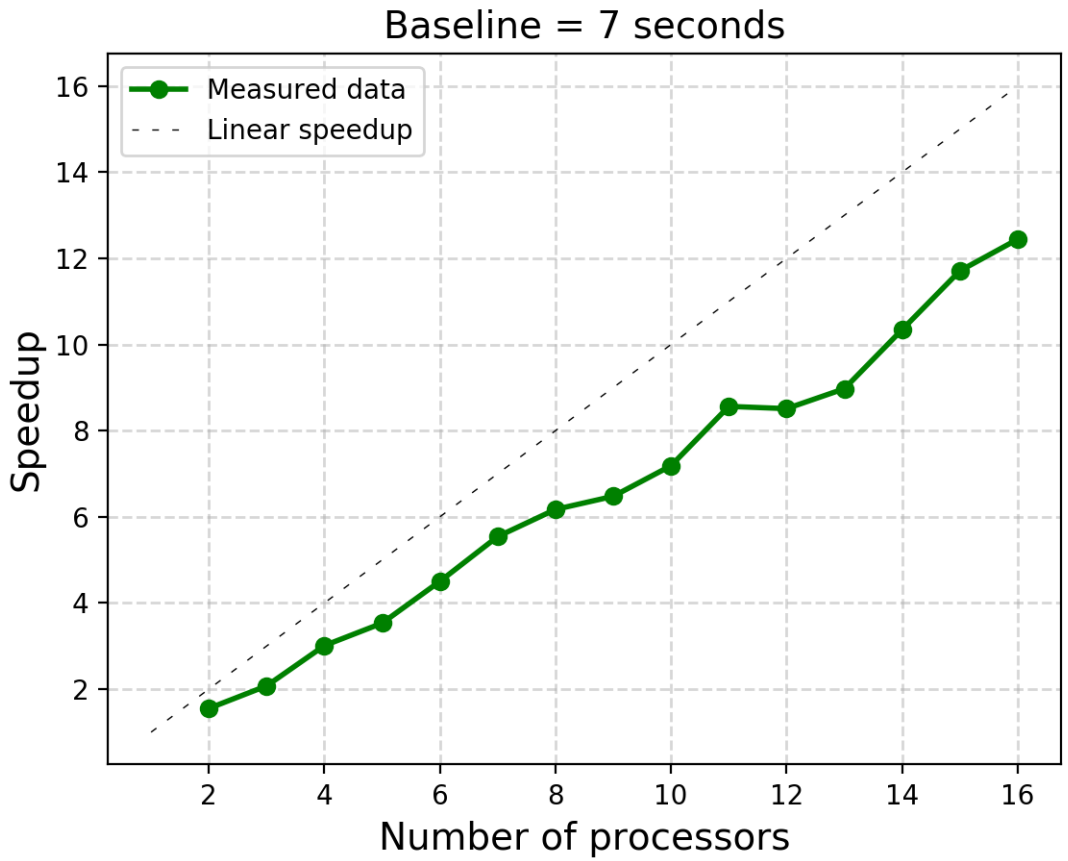


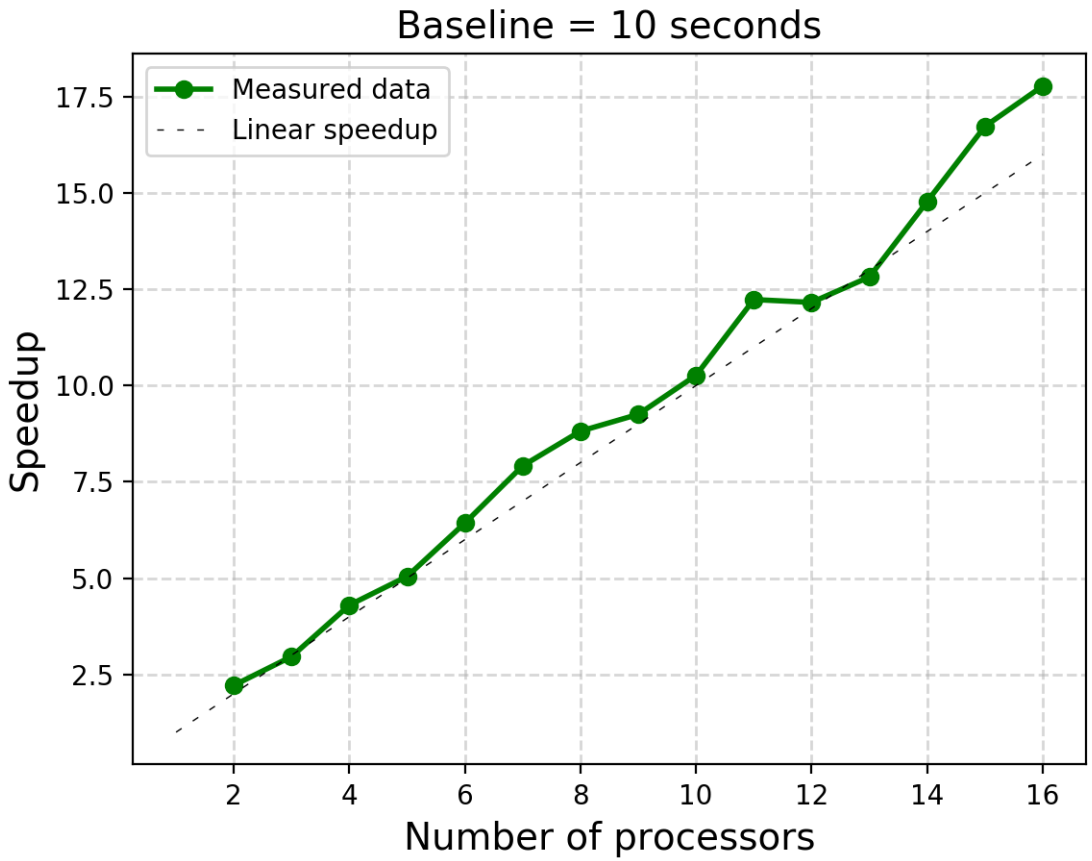
Figure 1: Distribution of completion times for 50 HPL runs.

provides a much more interpretable and informative representation of the collected runtimes of 50 executions. It shows that the variation is up to 20% and the slowest run was only 61.2 Tflop/s

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: Pitfalls of Relative Performance Reporting (Rule 1)





A. Rowstron et al.: Nobody ever got fired for using Hadoop on a cluster, HotCDP 2012

F. McSherry et al.: Scalability!

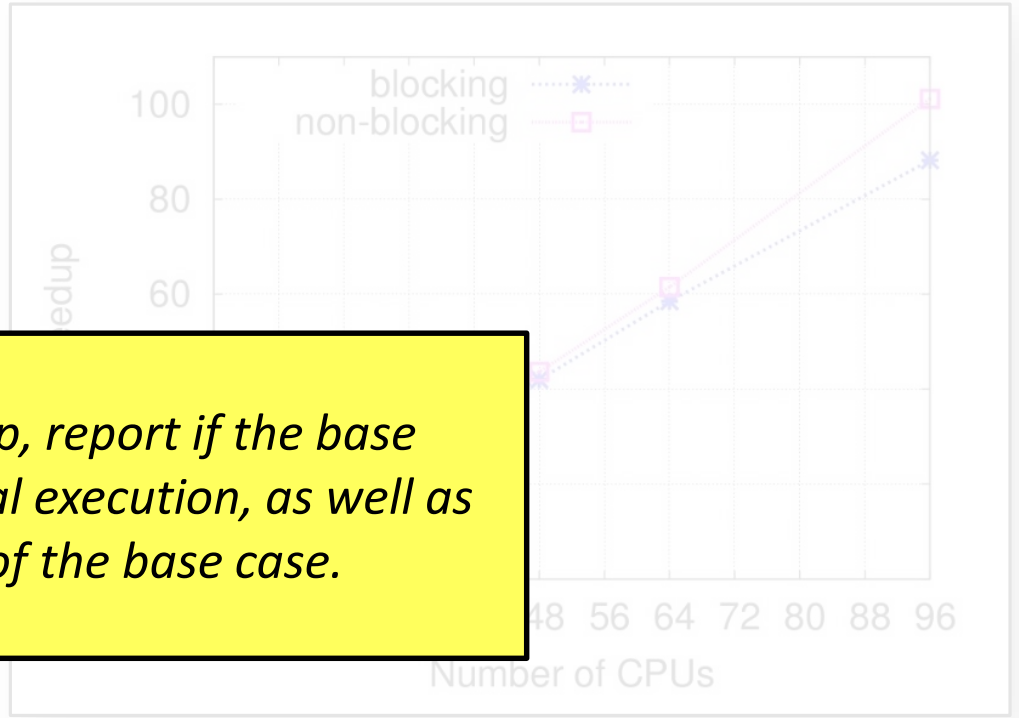
Both plots show speedups calculated from the same data.

The only difference is the baseline.

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: Pitfalls of Relative Performance Reporting (Rule 1)

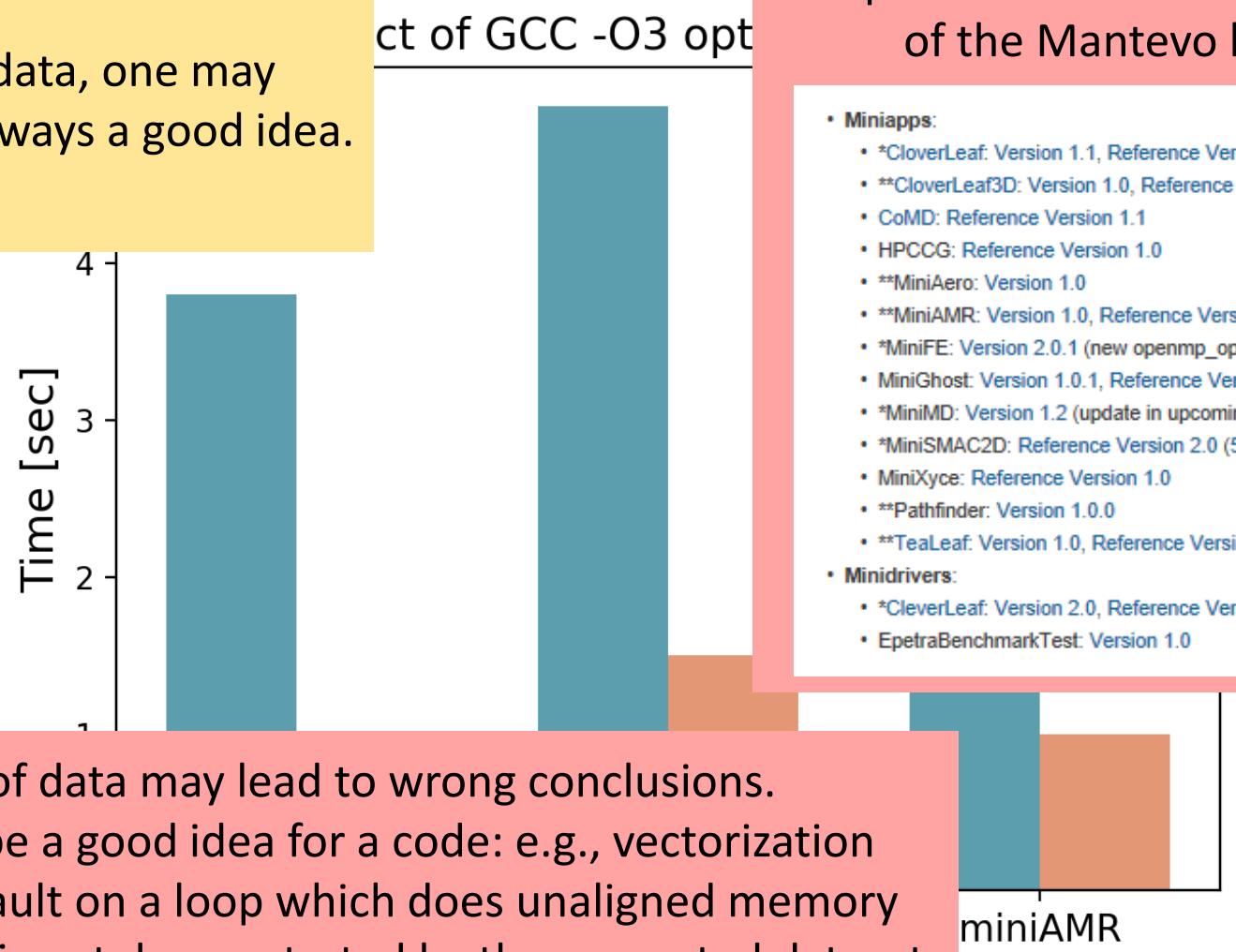
- **Most common (and oldest) problem with reporting**
  - First seen 1988 – also included in Bailey's 12 ways
  - Speedups can look arbitrarily good if it's relative to a bad baseline
  - Imagine an unop
    The optimized M
    What does this m
  - **Class question**: how could we improve the situation?
    - **A simple generalization of this rule implies that one should never report ratios without absolute values.**
  - Recently rediscovered in the "big data" universe
    A. Rowstron et al.: Nobody ever got fired for using Hadoop on a cluster, HotCDP 2012
    F. McSherry et al.: Scalability! but at what cost?, HotOS 2015



Rule 1: *When publishing parallel speedup, report if the base case is a single parallel process or best serial execution, as well as the absolute execution performance of the base case.*

Scientific Benchmarking: Benchmark Selection (Rule 2)

Based on the presented data, one may conclude that using **-O3** is always a good idea.



The presented data set contains only **a subset** of the Mantevo benchmark suite.

- **Miniapps**:
  - *CloverLeaf: Version 1.1, Reference Version 1.1
  - **CloverLeaf3D: Version 1.0, Reference Version 1.0
  - CoMD: Reference Version 1.1
  - HPCCG: Reference Version 1.0
  - **MiniAero: Version 1.0
  - **MiniAMR: Version 1.0, Reference Version 1.0
  - *MiniFE: Version 2.0.1 (new openmp_opt version), Reference Version 2.0
  - MiniGhost: Version 1.0.1, Reference Version 1.0.1
  - *MiniMD: Version 1.2 (update in upcoming minor suite release), Reference Version 2.0
  - *MiniSMAC2D: Reference Version 2.0 (5kx5k, 7kx7k test inputs)
  - MiniXyce: Reference Version 1.0
  - **Pathfinder: Version 1.0.0
  - **TeaLeaf: Version 1.0, Reference Version 1.0
- **Minidrivers**:
  - *CleverLeaf: Version 2.0, Reference Version 2.0
  - EpetraBenchmarkTest: Version 1.0

The incompleteness of data may lead to wrong conclusions. Sometimes **-O3** may not be a good idea for a code: e.g., vectorization (enabled by **-O3**) may segfault on a loop which does unaligned memory access on some x86. But this is not demonstrated by the presented dataset.

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: Benchmark Selection (Rule 2)

Based on the presented data, one may conclude that using **-O3** is always a good idea.

The presented data set contains only **a subset** of the Mantevo benchmark suite.

Rule 2: *Specify the reason for only reporting subsets of standard benchmarks or applications or not using all system resources.*

- **This implies: Show results even if your code/approach stops scaling!**

The incompleteness of data may lead to wrong conclusions. Sometimes **-O3** may not be a good idea for a code: e.g., vectorization (enabled by **-O3**) may segfault on a loop which does unaligned memory access on some x86. But this is not demonstrated by the presented dataset.

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0

RULE 3 and 4

Testcase						III
Floating-point o						20,0
Time [seconds]						**2,0**
Flop Rate [Gflop						10,0
Arithmetic Mea						
Harmonic Mean						
Flop Rate by Dividing Totals [Gflop/s]	11,3			10,2		

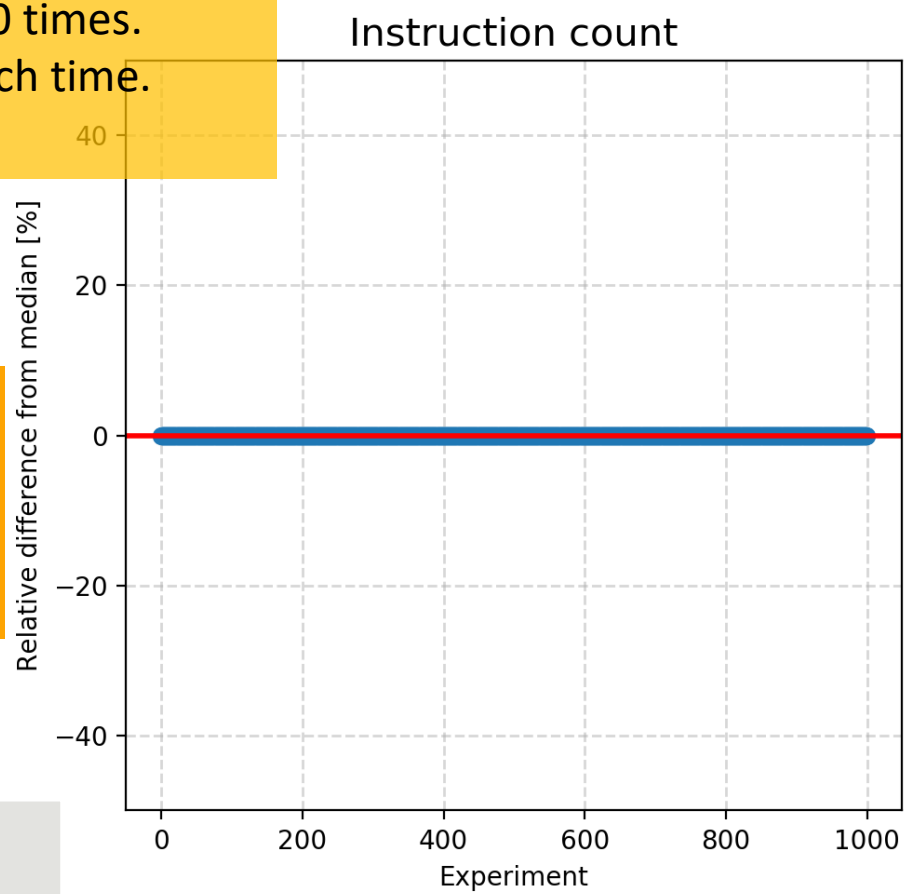
Rule 3: *Use the arithmetic mean only for summarizing costs. Use the harmonic mean for summarizing rates.*

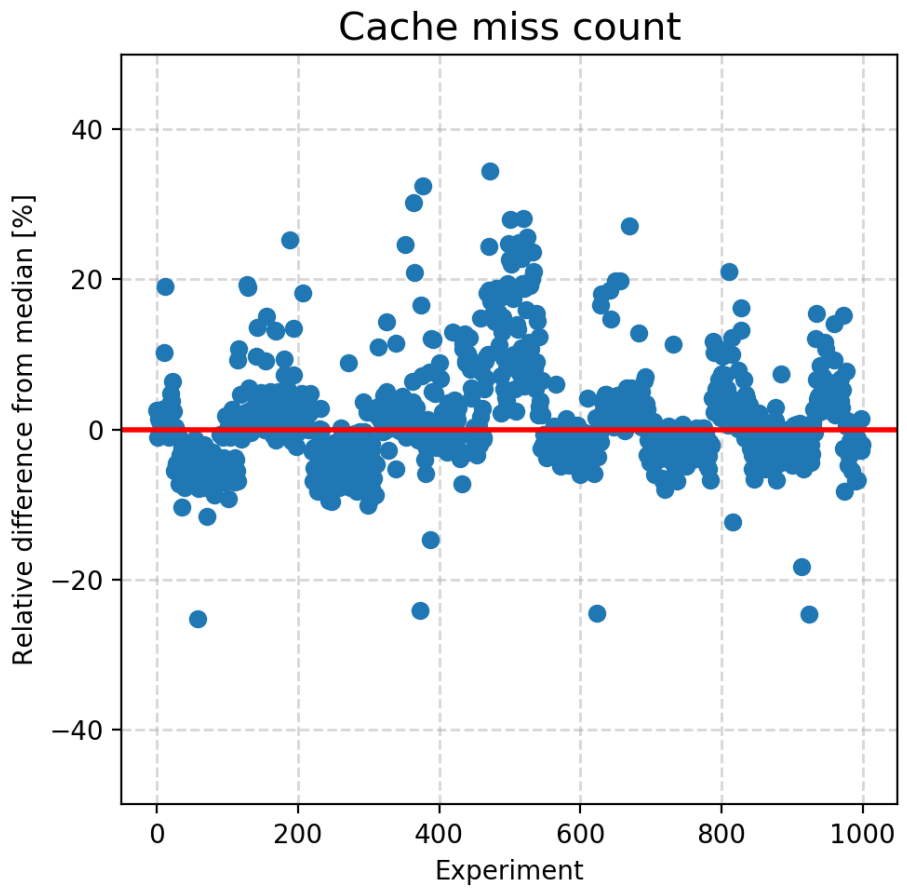
Rule 4: *Avoid summarizing ratios (e.g., speedup); summarize the costs or rates that the ratios base on instead. Only if these are not available use the geometric mean for summarizing ratios.*

Nondeterminism in [most] performance measurements!

Same code executed 1000 times.
Two metrics measured each time.

How do we report measurements showing high variation?





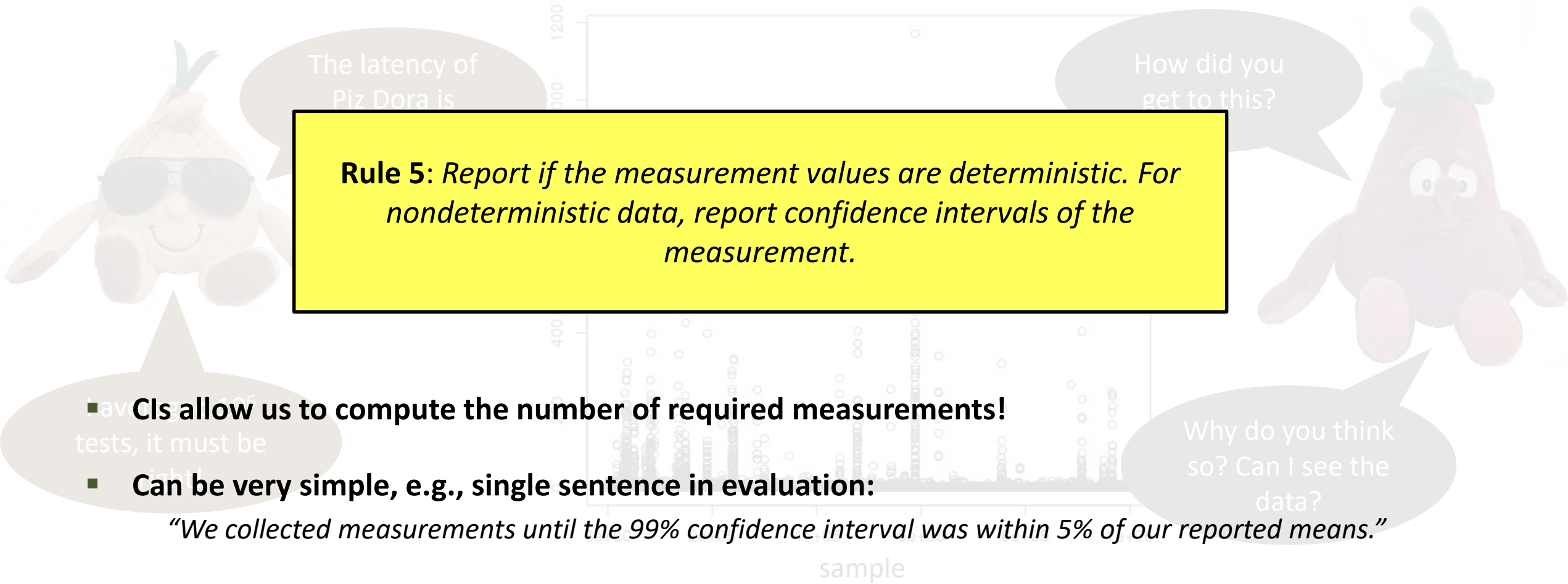

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

One is amazingly stable.

The other—not at all!

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

The simplest networking question: ping pong latency!

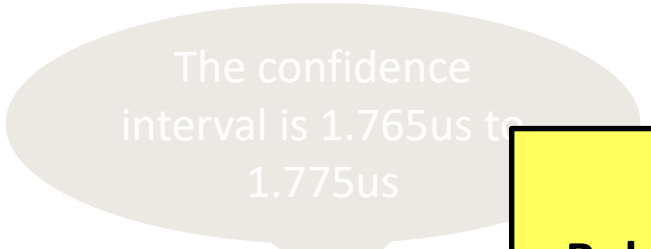


Rule 5: *Report if the measurement values are deterministic. For nondeterministic data, report confidence intervals of the measurement.*

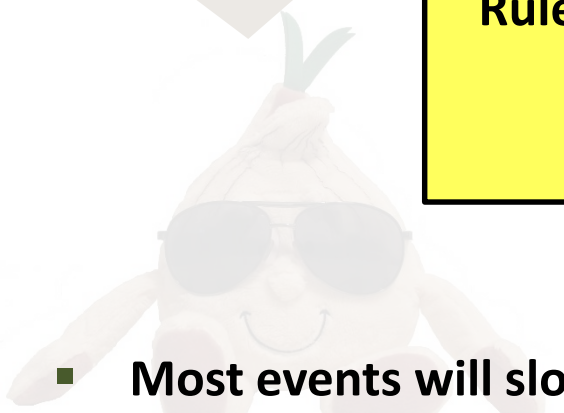
- **CIs allow us to compute the number of required measurements!**
- **Can be very simple, e.g., single sentence in evaluation:**

  "We collected measurements until the 99% confidence interval was within 5% of our reported means."

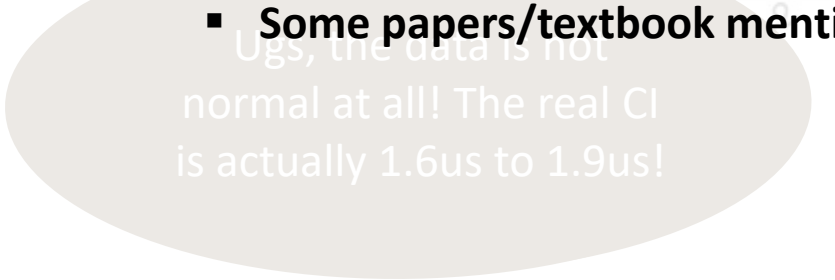
Thou shalt not trust your average textbook!



Rule 6: *Do not assume normality of collected data (e.g., based on the number of samples) without diagnostic checking.*



- **Most events will slow down performance**
  - **Heavy right-tailed distributions**
- **The Central Limit Theorem only applies asymptotically**
  - **Some papers/textbook mention "30-40 samples", don't trust them!**



Thou shalt not trust your system!

Look what data I got!

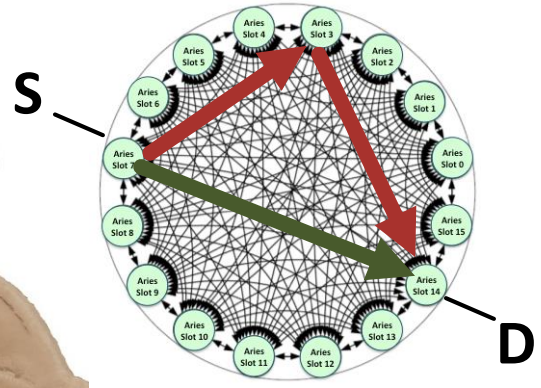
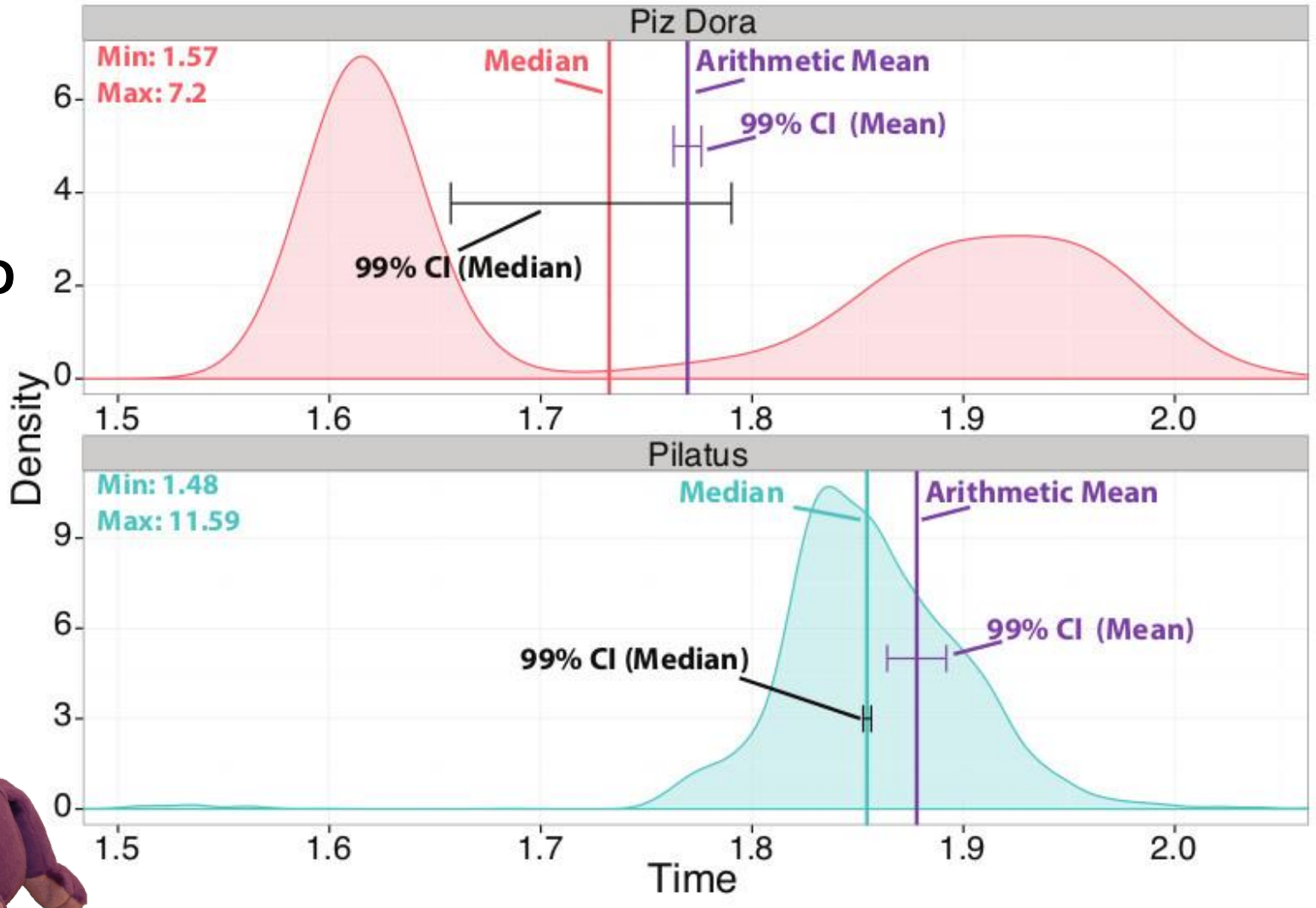
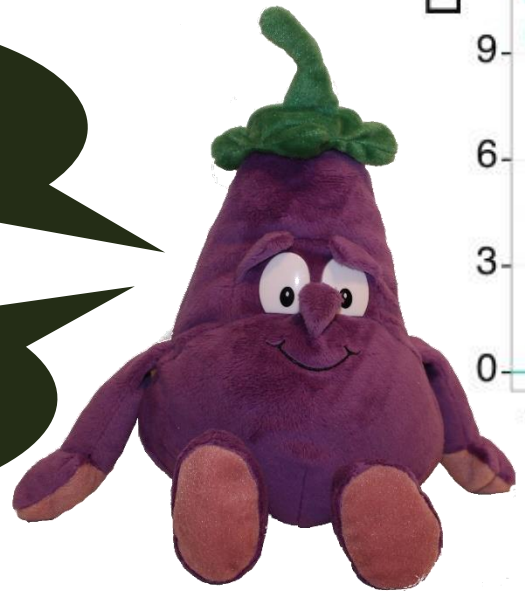


Image credit: nersc.gov



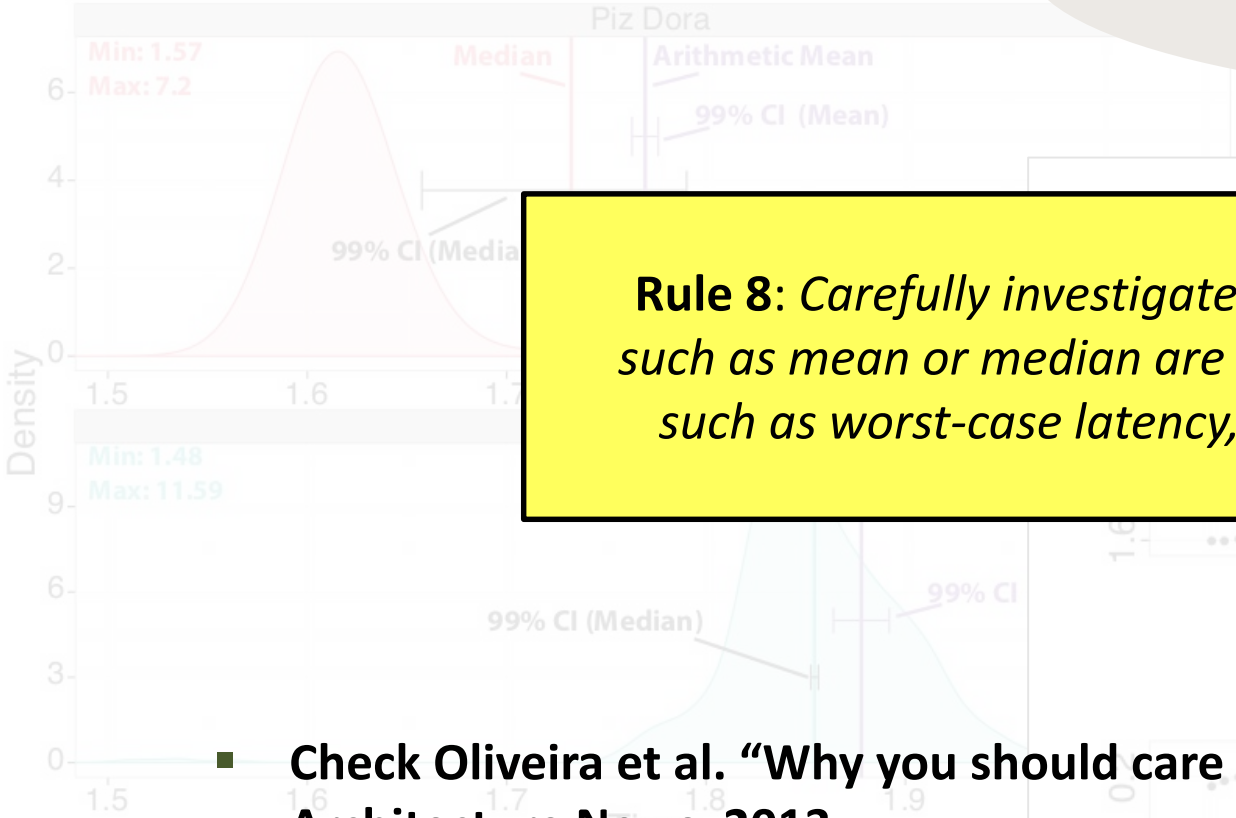
Clearly, the mean/median are not sufficient!

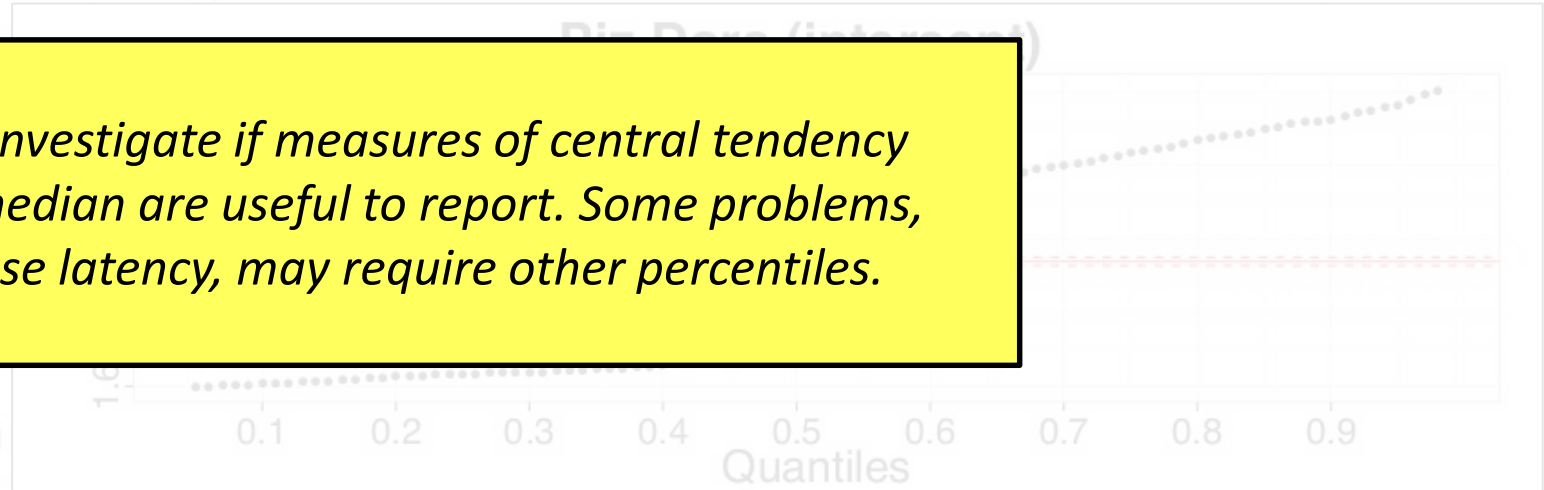
Try quantile regression!

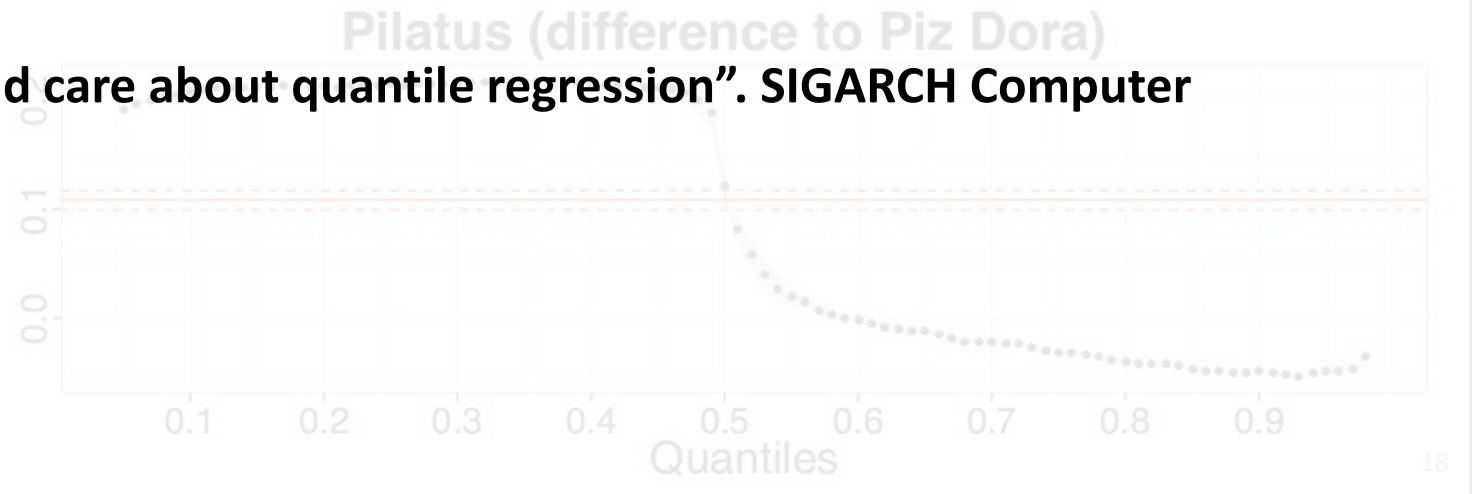


Quantile Regression

Wow, so Pilatus is better for (worst-case) latency-critical workloads even though Dora is expected to be faster







Rule 8: *Carefully investigate if measures of central tendency such as mean or median are useful to report. Some problems, such as worst-case latency, may require other percentiles.*

- **Check Oliveira et al. "Why you should care about quantile regression". SIGARCH Computer Architecture News, 2013.**

How many measurements are needed?

- **Measurements can be expensive!**
  - Yet necessary to reach certain confidence

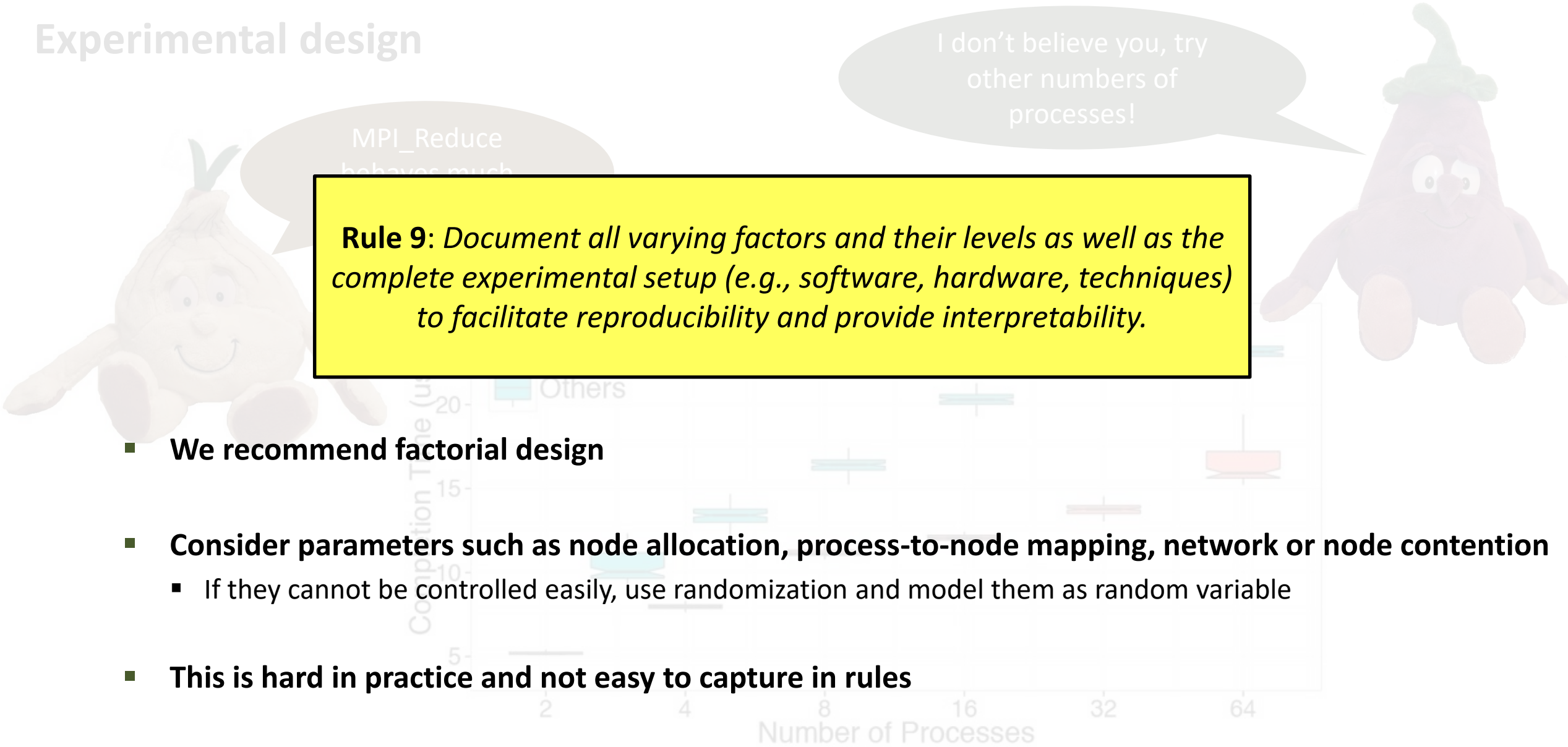
- **How to determine the minimal number of measurements?**
  - Measure until the confidence interval has a certain acceptable width
  - For example, measure until the 95% CI is within 5% of the mean/median
  - Can be computed analytically assuming normal data
  - Compute iteratively for nonparametric statistics

- **Often heard: "we cannot afford more than a single measurement"**
  - E.g., Gordon Bell runs
  - Well, then one cannot say anything about the variance

    Even 3-4 measurement can provide very tight CI (assuming normality)

    Can also exploit repetitive nature of many applications

Experimental design



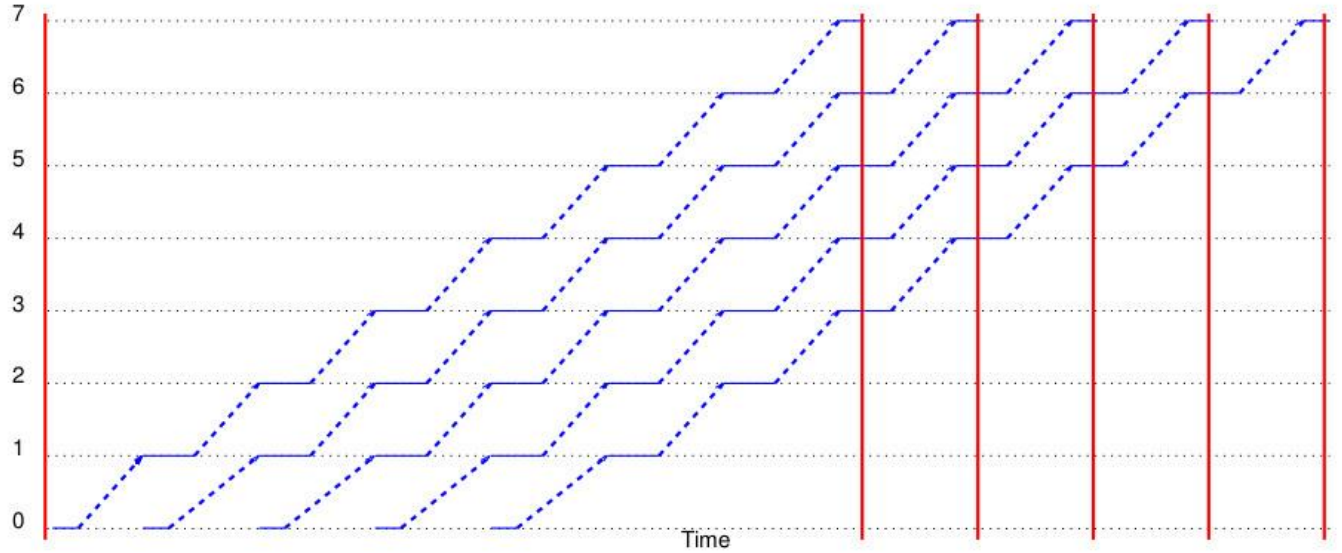
Rule 9: *Document all varying factors and their levels as well as the complete experimental setup (e.g., software, hardware, techniques) to facilitate reproducibility and provide interpretability.*

- **We recommend factorial design**
- **Consider parameters such as node allocation, process-to-node mapping, network or node contention**
  - If they cannot be controlled easily, use randomization and model them as random variable
- **This is hard in practice and not easy to capture in rules**

Time in parallel systems

My simple broadcast takes only one latency!

But I measured it so it must be true!



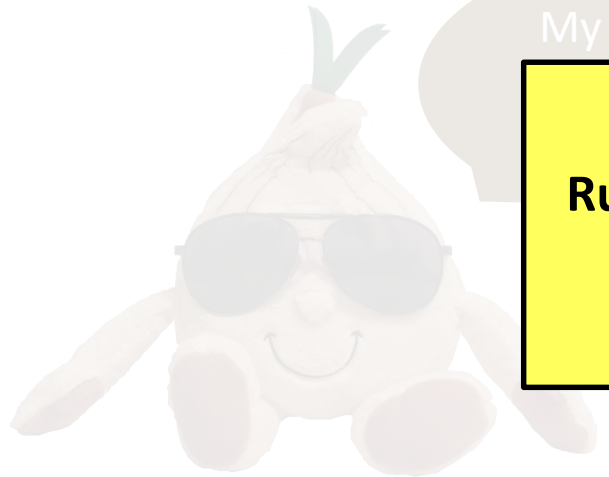
That's nonsense!

...

Measure each operation separately!

```
t = -MPI_Wtime();
for(i=0; i<1000; i++) {
  MPI_Bcast(…);
}
t += MPI_Wtime();
t /= 1000;
```

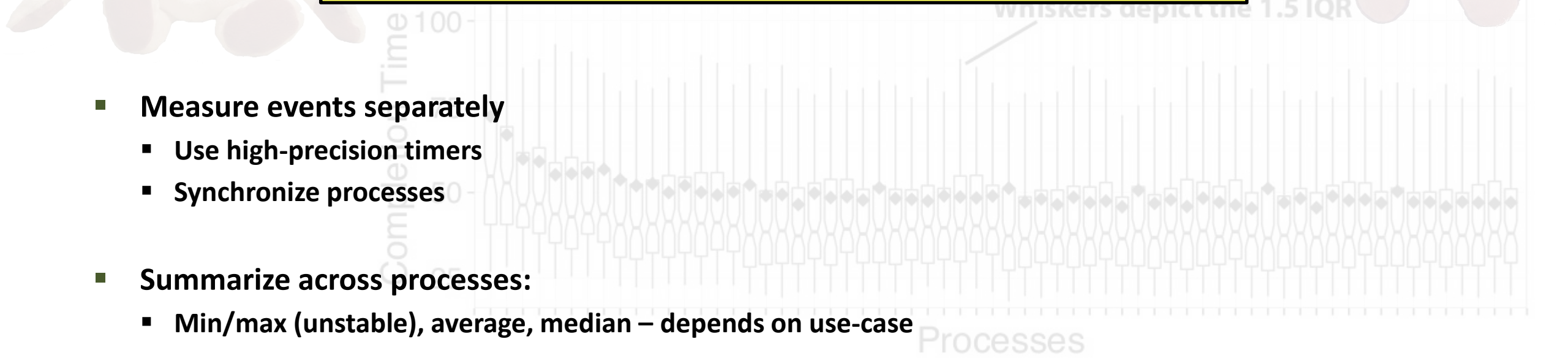
Summarizing times in parallel systems!



My new reduce

Come on, show me the data!

Rule 10: *For parallel time measurements, report all measurement, (optional) synchronization, and summarization techniques.*



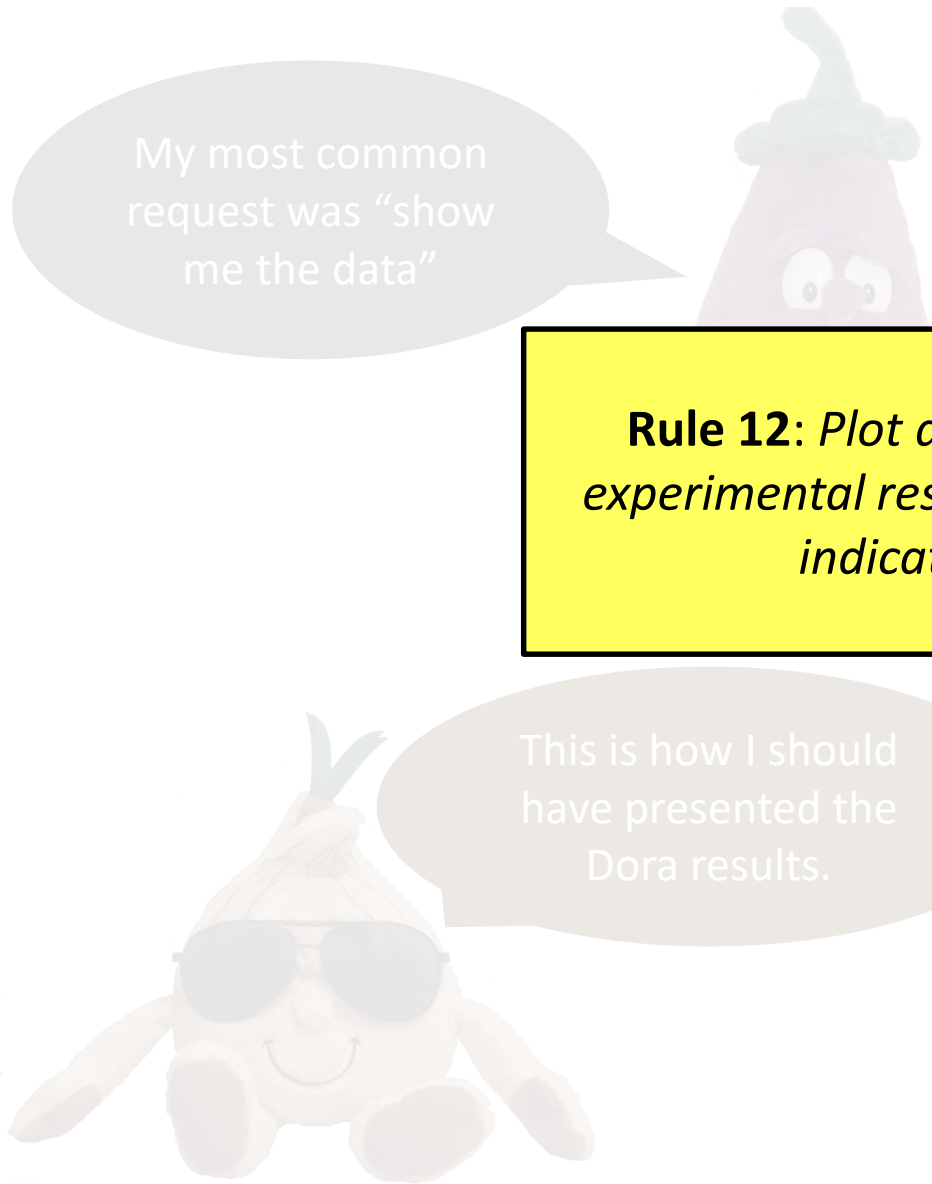
- **Measure events separately**
  - **Use high-precision timers**
  - **Synchronize processes**
- **Summarize across processes:**
  - **Min/max (unstable), average, median – depends on use-case**

Give times a meaning!

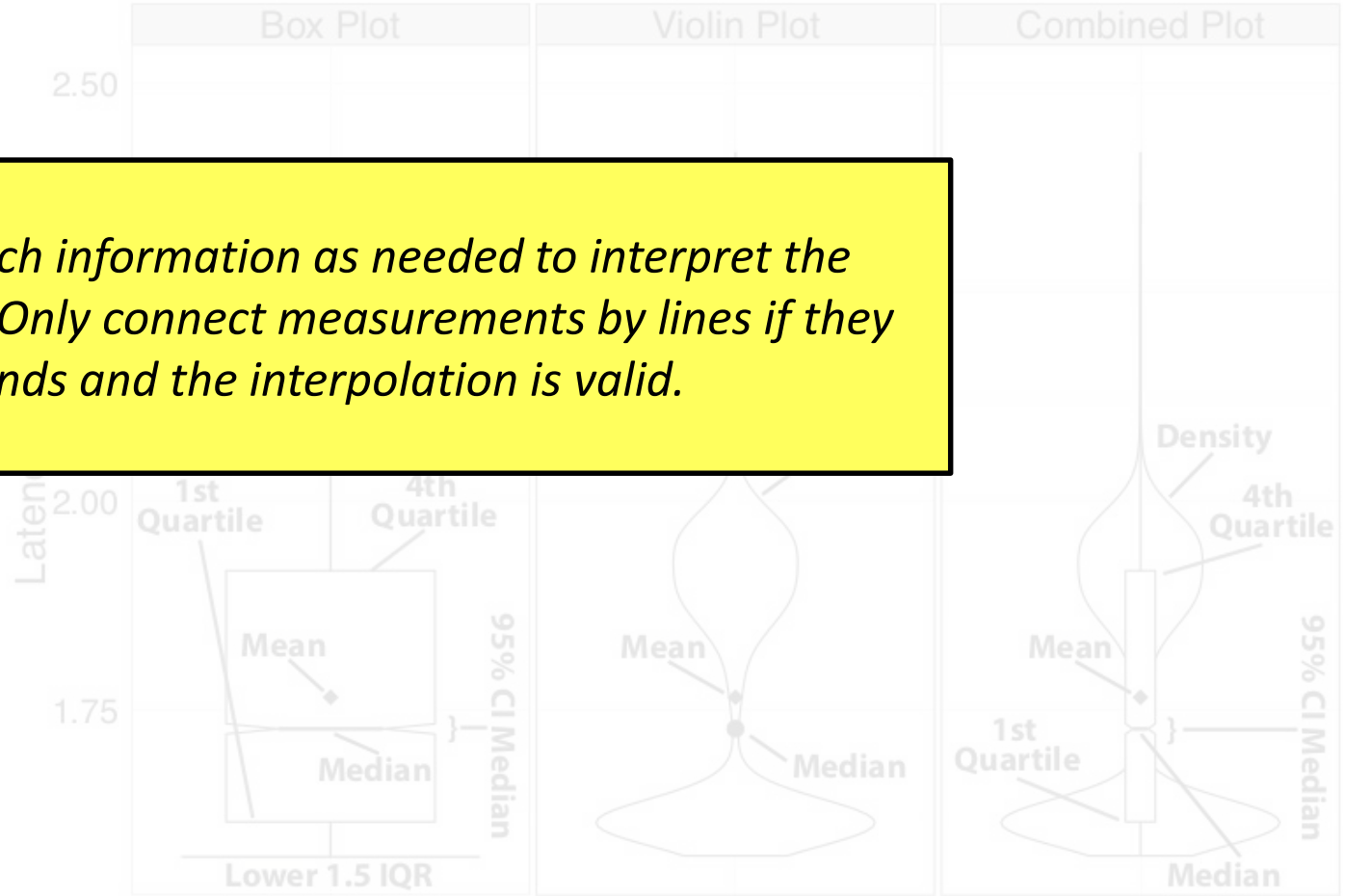
Rule 11: *If possible, show upper performance bounds to facilitate interpretability of the measured results.*

- **Model computer system as k-dimensional space**
  - Each dimension represents a capability
    Floating point, Integer, memory bandwidth, cache bandwidth, etc.
  - Features are typical rates
  - Determine maximum rate for each dimension
    E.g., from documentation or benchmarks
- **Can be used to prove optimality of implementation**
  - If the requirements of the bottleneck dimension are minimal

Plot as much information as possible!

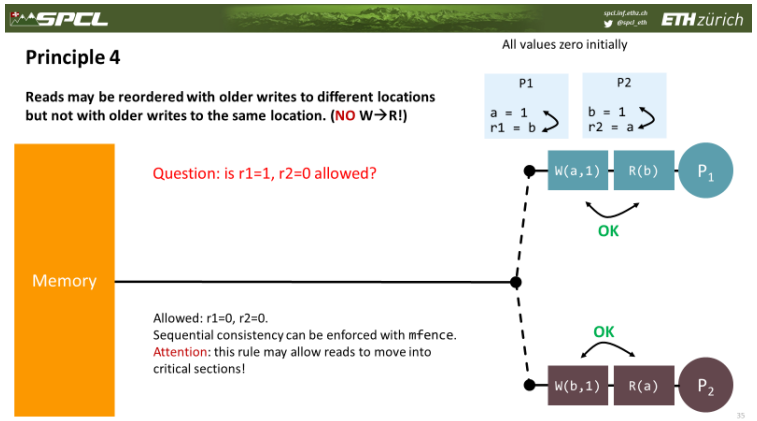


Rule 12: *Plot as much information as needed to interpret the experimental results. Only connect measurements by lines if they indicate trends and the interpolation is valid.*



Back to Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?




```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  asm("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])
  - Still $1.3 \cdot 10^{-6}$%

    Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  asm("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])
  - Still $1.3 \cdot 10^{-6}$%

    Why?

    Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  asm("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm("mfence");
  flag[tid] = 0; // I'm not interested
}
```

The compiler may inline this function ☺

Correct Peterson Lock on x86

- **Unoptimized (naïve sprinkling of mfences)**
- **Performance:**
  - No mfence
    375ns
  - mfence in lock
    379ns
  - mfence in unlock
    404ns
  - Two mfence
    427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  asm("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm("mfence");
  flag[tid] = 0; // I'm not interested
}
```

Hardware Support?

- **Hardware atomic operations:**
  - Test&Set

    Write const to memory while returning the old value
  - Atomic swap

    Atomically exchange memory and register
  - Fetch&Op

    Get value and apply operation to memory location
  - Compare&Swap

    Compare two values and swap memory with register if equal
  - Load-linked/Store-Conditional LL/SC (or load-acquire (LDA) store-release (STL) on ARM)

    Loads value from memory, allows operations, commits only if no other updates committed → mini-TM
  - Intel TSX (transactional synchronization extensions)

    Hardware-TM (roll your own atomic operations)

```
bool TestAndSet (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
```

```
movl    $1, %eax
xchg    %eax, (%ebx)
```

```
__global__ void Sum (float *in, float *out, int N) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  float previous_value = atomicAdd(out, in[tid]);
  // do something with previous_value
}
```

```
bool CompareAndSwap (T *value, T old, T new) {
  if (*value != old) return false;
  *value = new;
  return true;
} // all atomic!
```

```
movl                    $1, %eax
xacquire lock xchg      %eax, (%ebx)
...
xrelease movl           $0, (%ebx)
```

Relative Power of Synchronization

- **Design-Problem I: Multi-core Processor**
  - Which atomic operations are useful?
- **Design-Problem II: Complex Application**
  - What atomic should I use?
- **Generally hard to answer ☹**
  - Depends on too many system and application details (access patterns, CC implementation, contention, algorithm …)
- **Concept of "consensus number" C: if a primitive can be used to solve the "consensus problem" in a finite number of steps (even if threads stop)**
  - atomic registers have C=1 (thus locks have C=1!)
  - TAS, Swap, Fetch&Op have C=2
  - CAS, LL/SC, TM have C=∞

Test-and-Set Locks

- **Test-and-Set semantics**
  - Memoize old value
  - Set fixed value TASval (true)
  - Return old value
- **After execution:**
  - Post-condition is a fixed (constant) value!

```
bool TestAndSet (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
```

Test-and-Set Locks

- **Assume TASval indicates “locked”**
- **Write something else to indicate “unlocked”**
- **TAS until return value is != TASval (1 in this example)**

```
bool TestAndSet (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
```

- **Questions:**
  - When will the lock be granted?
  - Does this work well in practice?

  Is spinning in lock (TAS) a good idea?

```
volatile int lck = 0;

void lock() {
  while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

Cacheline contention (or: MESI and friends return)

- **On x86, the XCHG instruction is used to implement TAS**
  - x86 lock is implicit in xchg!
- **Cacheline is read and written**
  - Ends up in exclusive state, invalidates other copies
  - Cacheline is “thrown” around uselessly
  - High load on memory subsystem

  x86 lock is essentially a full memory barrier ☹

```
movl    $1, %eax
xchg    %eax, (%ebx)
```

Test-and-Test-and-Set (TATAS) Locks

- **Spinning in TAS is not a good idea**
- **Spin on cache line in shared state**
  - All threads at the same time, no cache coherency/memory traffic

- **Danger!**
  - Efficient but use with great care!
  - Generalizations are very dangerous

```
volatile int lck = 0;

void lock() {
  do {
     while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

Warning: Even experts get it wrong!

- **Example: Double-Checked Locking**

1997

Double-Checked Locking

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

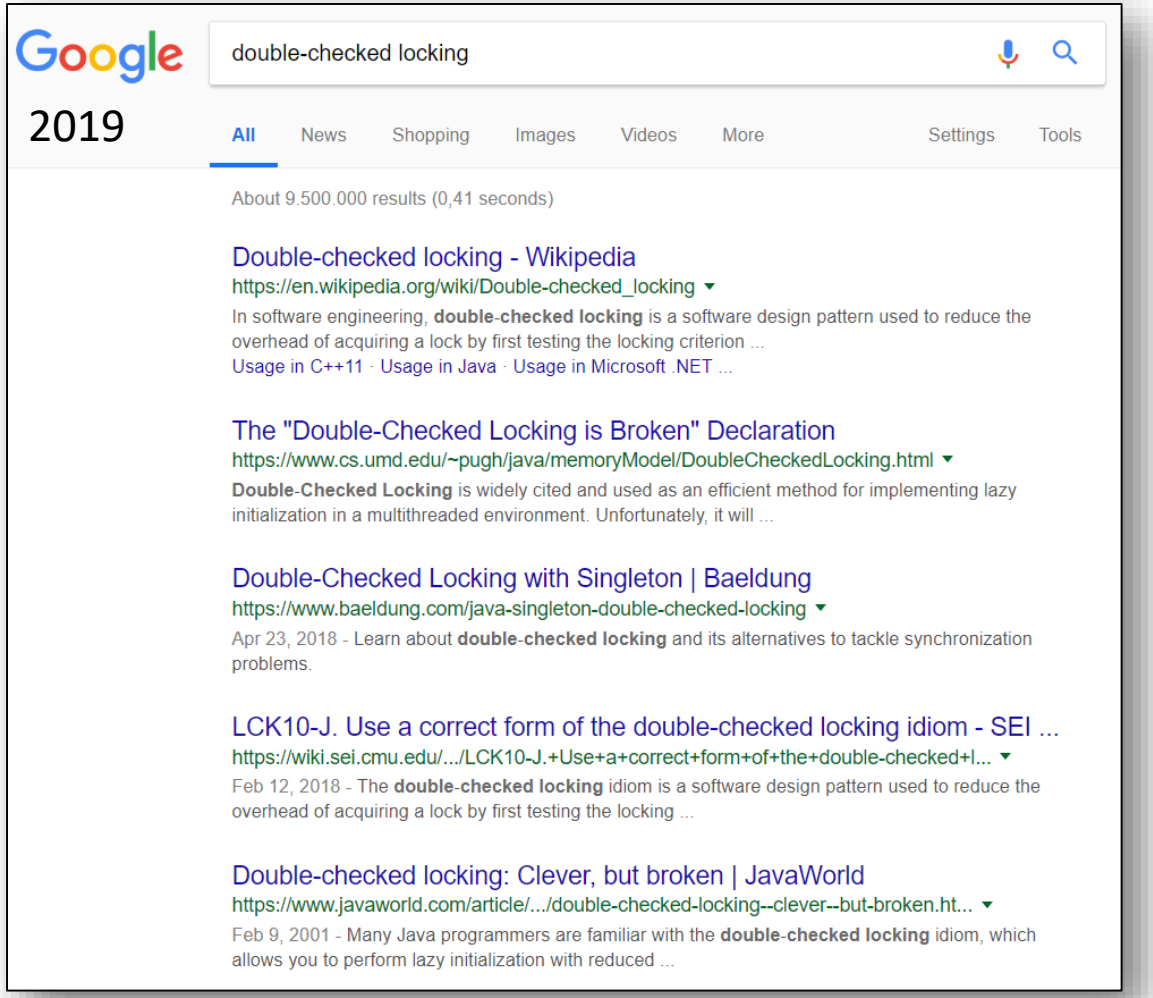
context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    if (instance_ == 0)
      // Critical section.
      instance_ = new Singleton;

    return instance_;
  }
```

2019



Problem: Memory ordering leads to race-conditions!

Contention?

- **Do TATAS locks still have contention?**
- **When lock is released, k threads fight for cache line ownership**
  - One gets the lock, all get the CL exclusively (serially!)
  - What would be a good solution?

    think "collision avoidance"

```
volatile int lck = 0;

void lock() {
  do {
    while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

TAS Lock with Exponential Backoff

- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    How can we make it even less likely?

```
volatile int lck = 0;

void lock() {
  while (TestAndSet(&lck) == 1) {
    wait(time);
    time *= 2; // double waiting time
  }
}

void unlock() {
  lck = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

TAS Lock with Exponential Backoff

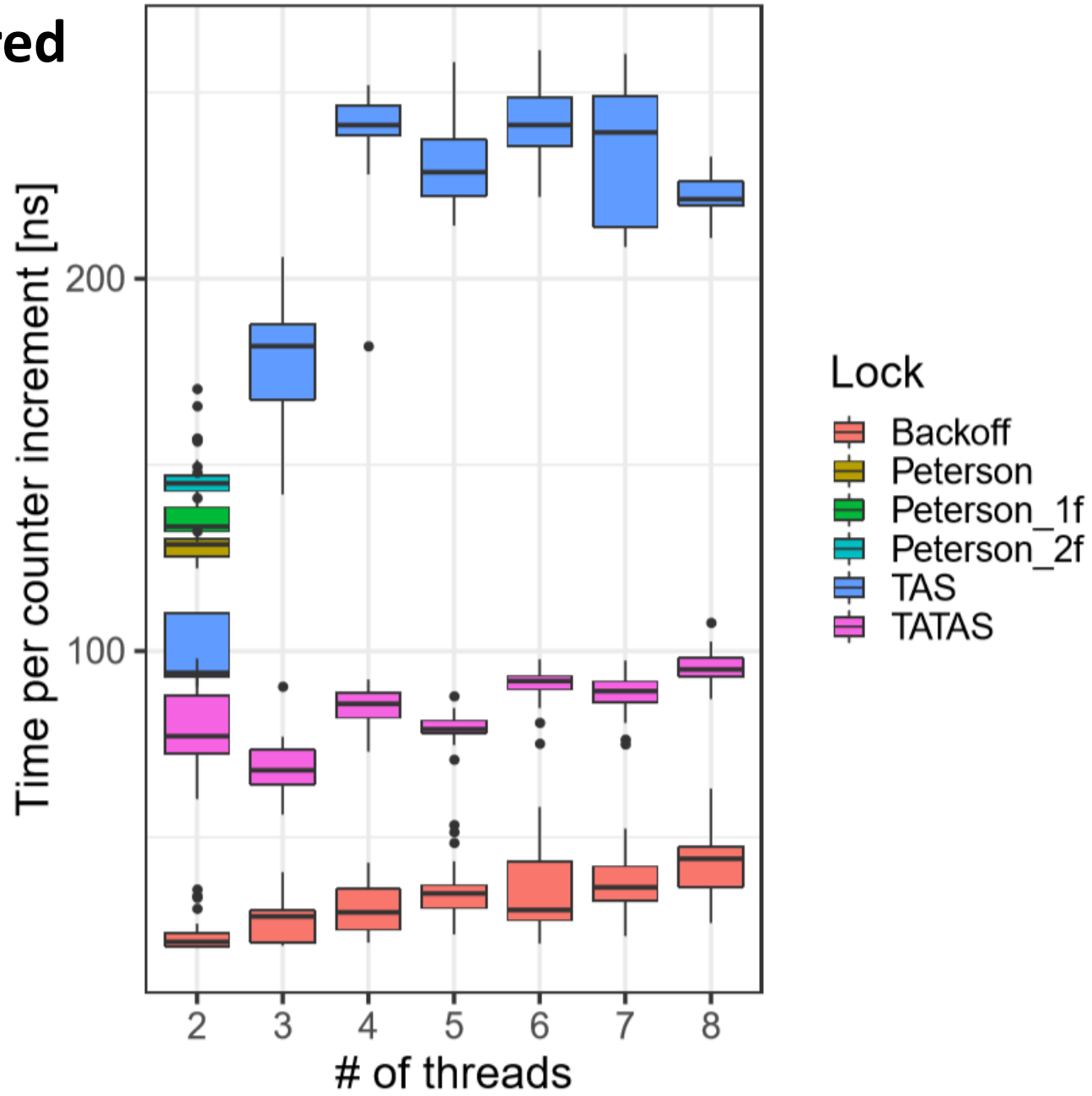
- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    Maximum waiting time makes it less likely

```
volatile int lck = 0;
const int maxtime=1000;

void lock() {
  while (TestAndSet(&lck) == 1) {
    wait(time);
    time = min(time * 2, maxtime);
  }
}

void unlock() {
  lck = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

Performance of our locks compared



Improvements?

- **Are TAS locks perfect?**
  - What are the two biggest issues?
    - Cache coherency traffic (contending on same location with expensive atomics)

      -- or --
    - Critical section underutilization (waiting for backoff times will delay entry to CR)
- **What would be a fix for that?**
  - How is this solved at airports and shops (often at least)?

- **Queue locks -- Threads enqueue**
  - Learn from predecessor if it's their turn
  - Each threads spins at a different location
  - FIFO fairness

Array Queue Lock

- **Array to implement queue**
  - Tail-pointer shows next free queue position
  - Each thread spins on own location

    CL padding!
  - index[] array can be put in TLS

- **So are we done now?**
  - What's wrong?
  - Synchronizing M objects requires Θ(NM) storage
  - What do we do now?

```
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
index[tid] = FetchAndInc(tail) % n;
while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
array[index[tid]] = 0; // I release my lock
array[(index[tid] + 1) % n] = 1; // next one
}
```

CLH Lock (1993)

- **List-based (same queue principle)**
  - Discovered twice by Craig, Landin, Hagersten 1993/94
- **2N+3M words**
  - N threads, M locks
- **Requires thread-local qnode pointer**
  - Can be hidden!

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock
```

```
void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

CLH Lock (1993)

- **Qnode objects represent thread state!**
  - succ_blocked == 1 if waiting or acquired lock
  - succ_blocked == 0 if released lock
- **List is implicit!**
  - One node per thread
  - Spin location changes

    NUMA issues (cacheless)
- **Can we do better?**

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock
```

```
void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

MCS Lock (1991)

- **Make queue explicit**
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- **Similar advantages as CLH but**
  - Only 2N + M words
  - Spinning position is fixed!

    Benefits cache-less NUMA
- **What are the issues?**
  - Releasing lock spins
  - More atomics!

```
typedef struct qnode {
  struct qnode *next;
  int locked;
} qnode;

qnode *lck = NULL;
```

```
void lock(qnode **lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(*lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
  }
}

void unlock(qnode **lck, qnode *qn) {
  if(qn->next == NULL) { // if we're the last waiter
    if(CAS(*lck, qn, NULL)) return;
    while(qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}
```

Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Lessons Learned!

- **Key Lesson:**
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)

    Remember how the coherence protocols work (can lead to better protocols!)

- **MCS: 2006 Dijkstra Prize in distributed computing**
  - "*an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade*"
  - "*probably the most influential practical mutual exclusion algorithm ever*"
  - "*vastly superior to all previous mutual exclusion algorithms*"
  - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- **Down to memory complexity of 2N+M**
  - Probably close to optimal
- **Only local spinning**
  - Several variants with low expected contention
- **But: we assumed sequential consistency ☹**
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Open research on minimally-synching algorithms!

More Practical Optimizations

- **Let's step back to "data race"**
  - (recap) two operations A and B on the same memory cause a data race if one of them is a write ("conflicting access") and neither A→B nor B→A
  - So we put conflicting accesses into a CR and lock it!

    Remember: this also guarantees memory consistency in C++/Java!

- **Let's say you implement a web-based encyclopedia**
  - Consider the "average two accesses" – do they conflict?

Number of edits (2007-11/27/2017): 921,644,695
Average views per day: ~200,000,000

→ 0.12% write rate

Reader-Writer Locks

- **Allows multiple concurrent reads**
  - Multiple reader locks concurrently in CR
  - Guarantees mutual exclusion between writer and writer locks and reader and writer locks

- **Syntax:**
  - read_(un)lock()
  - write_(un)lock()

A Simple and Fast RW Lock

- **Seems efficient!?**
  - Is it? What's wrong?
  - Polling CAS!
- **Is it fair?**
  - Readers are preferred!
  - Can always delay writers (again and again and again)

```
const W = 1;
const R = 2;
volatile int lock=0; // LSB is writer flag!

void read_lock(lock_t lock) {
  AtomicAdd(lock, R);
  while(lock & W);
}

void write_lock(lock_t lock) {
  while(!CAS(lock, 0, W));
}

void read_unlock(lock_t lock) {
  AtomicAdd(lock, -R);
}

void write_unlock(lock_t lock) {
  AtomicAdd(lock, -W);
}
```

Fighting CPU waste: Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
  - Other threads can get them back on the queue!

- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**

- **Wait and signal are OS calls**
  - Often expensive, which one is more expensive?

    Wait, because it has to perform a full context switch

When to Spin and When to Block?

- **Spinning consumes CPU cycles but is cheap**
  - "Steals" CPU from other threads

- **Blocking has high one-time cost and is then free**
  - Often hundreds of cycles (trap, save TCB …)
  - Wakeup is also expensive (latency)

    Also cache-pollution

- **Strategy:**
  - Poll for a while and then block

    But what is a "while"??

When to Spin and When to Block?

- **Optimal time depends on the future**
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule

    Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?
  - Actual problem is an online problem

- **Competitive algorithms**
  - An algorithm is c-competitive if for a sequence of actions x and a constant a holds:

    $C(x) \leq c * C_{opt}(x) + a$
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- **If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!**
  - Karlin, Manasse, McGeoch, Owicki: “Competitive Randomized Algorithms for Non-Uniform Problems”, SODA 1989

- **If randomized algorithms are used, then e/(e-1)-competitiveness (~1.58) can be achieved**
  - See paper above!

Remember: lock-free vs. wait-free

- **A locked method**
  - May deadlock (methods may never finish)
- **A lock-free method**
  - Guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
  - Guarantees that **each** method call finishes in a finite number of steps (implies lock-free)
- **Synchronization instructions are not equally powerful!**
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.

Concept: Consensus Number

- **Each level of the hierarchy has a "consensus number" assigned.**
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem

- **The consensus problem:**
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:
    consistency: all threads get the same value
    validity: the value is some thread's input
  - Simplification: binary consensus (inputs in {0,1})

Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
  - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
  - The protocol has to be wait-free (bounded number of steps per thread)
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple …

- **Binary consensus with two threads (A, B)!**
  - Each thread moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree

    Example with two threads and two moves each!

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - i.e., they cannot be used to solve even two-thread consensus! Really?
- **Proof outline:**
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) *Any thread reads (other thread runs solo until end)*
    2) *Threads write to different registers (order doesn't matter)*
    3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
  - → We need hardware atomics or Transactional Memory!
- **Proof technique borrowed from:**

Impossibility of distributed consensus with one ... - ACM Digital Library
https://dl.acm.org/citation.cfm?id=214121
by MJ Fischer - 1985 - Cited by 4669 - Related articles
Sep 4, 2012 - Michael J. Fischer , Nancy A. Lynch , Michael S. Paterson, **Impossibility of distributed consensus** with one faulty process, Proceedings of the ...

- **Very influential paper, always worth a read!**
  - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)

- **CAS and TM have consensus number ∞**
  - Constructive proof!

Compare and Set/Swap Consensus

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return v; // I won!
  else
    return proposed[thread]; // thread won
}
```

- **CAS provides an infinite consensus number**
  - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
  - I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything ☺

- **Not really … ;-)**
  - We'll argue more about **performance** now!
- **But you have all the tools for:**
  - Efficient locks
  - Efficient lock-based algorithms
  - Reasoning about parallelism!
- **What now?**
  - Now you understand practice and will appreciate theory
    Wasn't that all too messy ☺?
  - Focus on (parallel) performance, techniques, and algorithms