Computational Reproducibility

by example of phylogenetic inference

Alexandros Stamatakis

Focus of This Lecture

- **Computational Reproducibility**

  → If I run a program with the same parameters *n* times will I always get the same results?

- We will not cover topics such as
  - Archiving, storing, and sharing the data
  - Providing scripts for reproducing results and figures
- I will tell you a story of all the things that have gone wrong over the years → Murphy's law

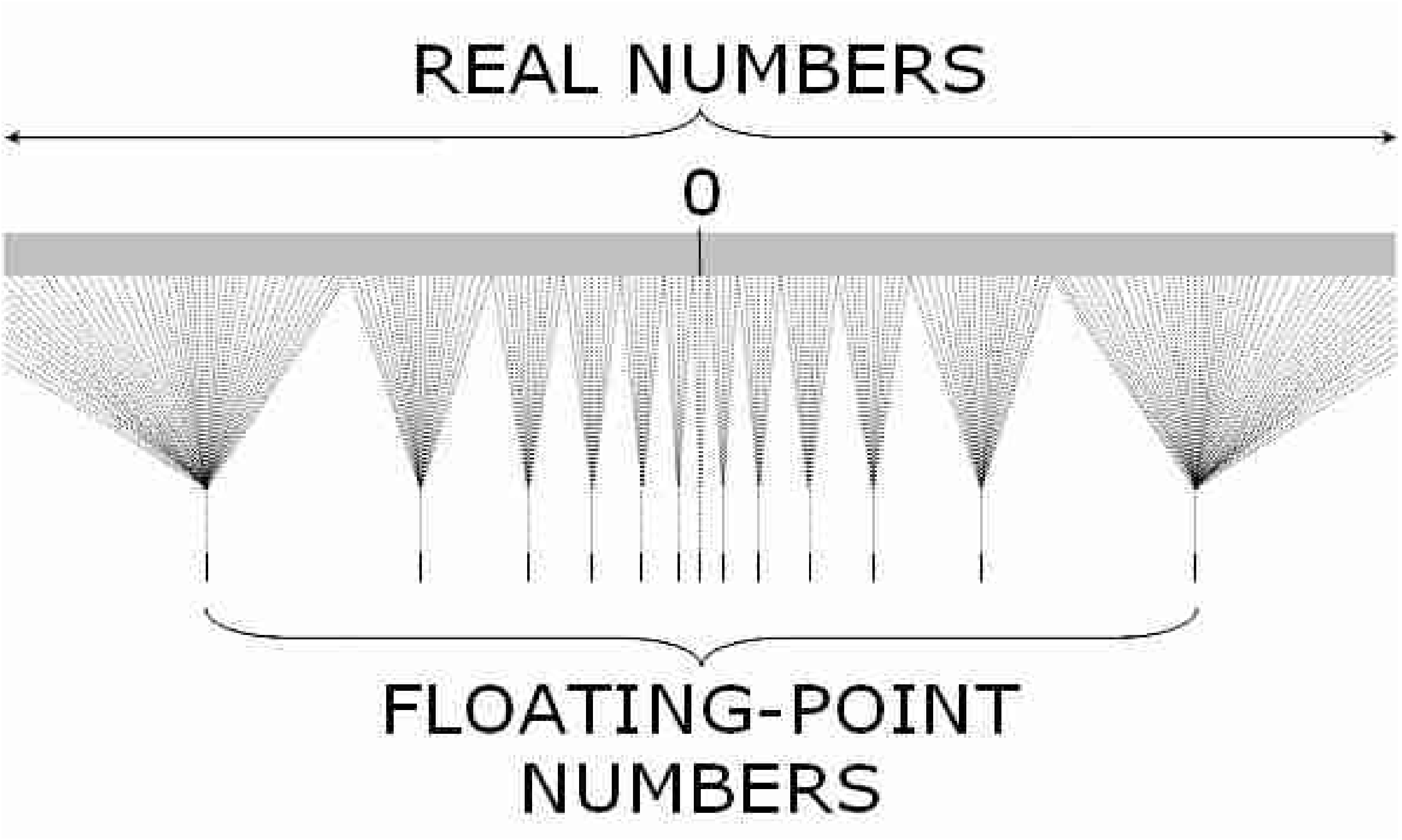
  Anything that can go wrong will go wrong

Outline

- **The root of all evil**
- Sequential Computations
- Parallel Computations
- Software Quality

Floating Point Numbers

- Machine numbers are an imperfect mapping of the infinite real numbers to a finite number of machine values!



Imperfect Mapping - Examples

- Double precision numbers (64 bits)
  - Sign bit: 1 bit
  - Exponent: 11 bits
  - Significand precision: 53 bits (52 explicitly stored)
- $2^{52} + 0.2 = 2^{52}$ (next number after 2^{52} is $2^{52} + 1$)
- $1 + 1 / 2^{54} = 1$ (next number after $1 + 1/2^{52}$)
- Between 2^n and 2^{n+1} there are always 2^{52} values that are evenly spaced !

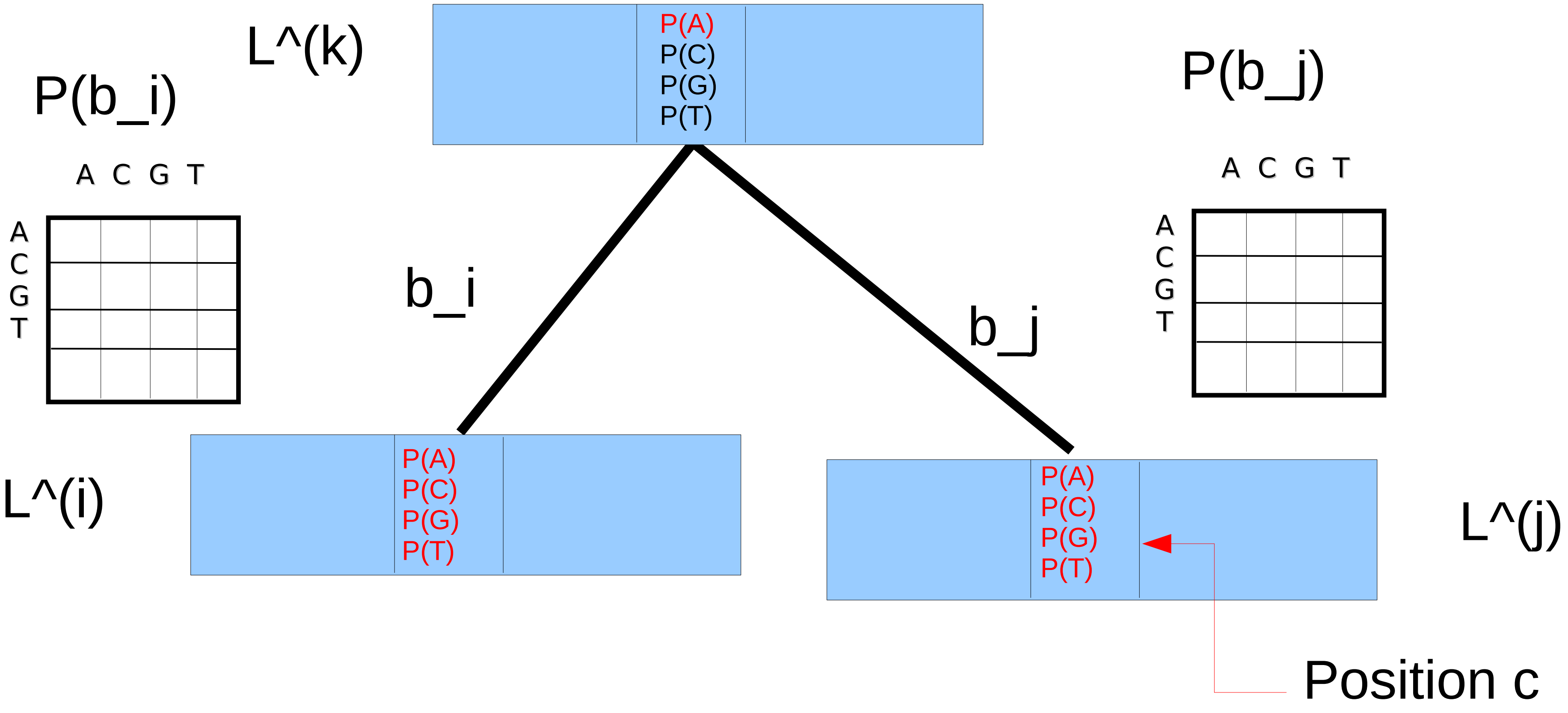
Statistics

- In most lectures of this course we deal with statistical computations

  → on the computer we need to use floating point values to represent probabilities

Felsenstein pruning

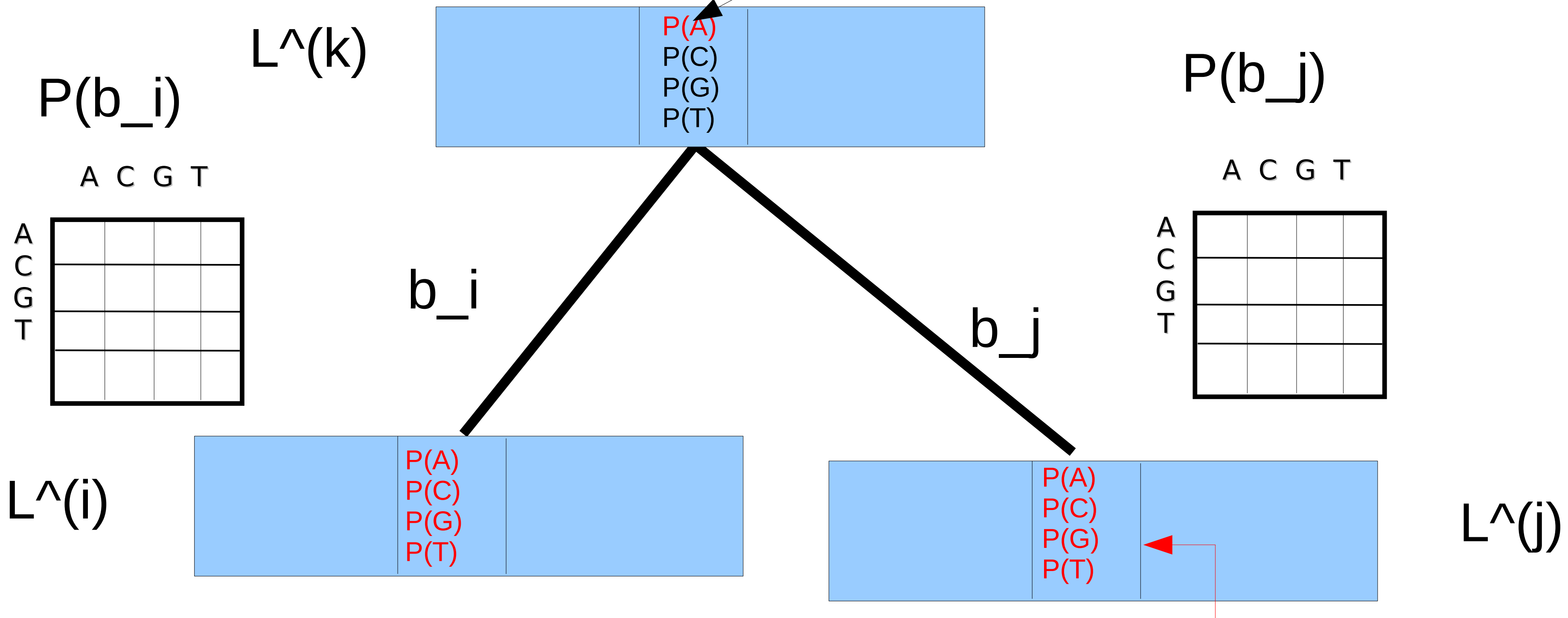
$$\vec{L}^{(k)}_A(c) = (\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}^{(i)}_S(c))(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}^{(j)}_S(c))$$



Felsenstein pruning

Values get smaller and smaller to as we approach the virtual root

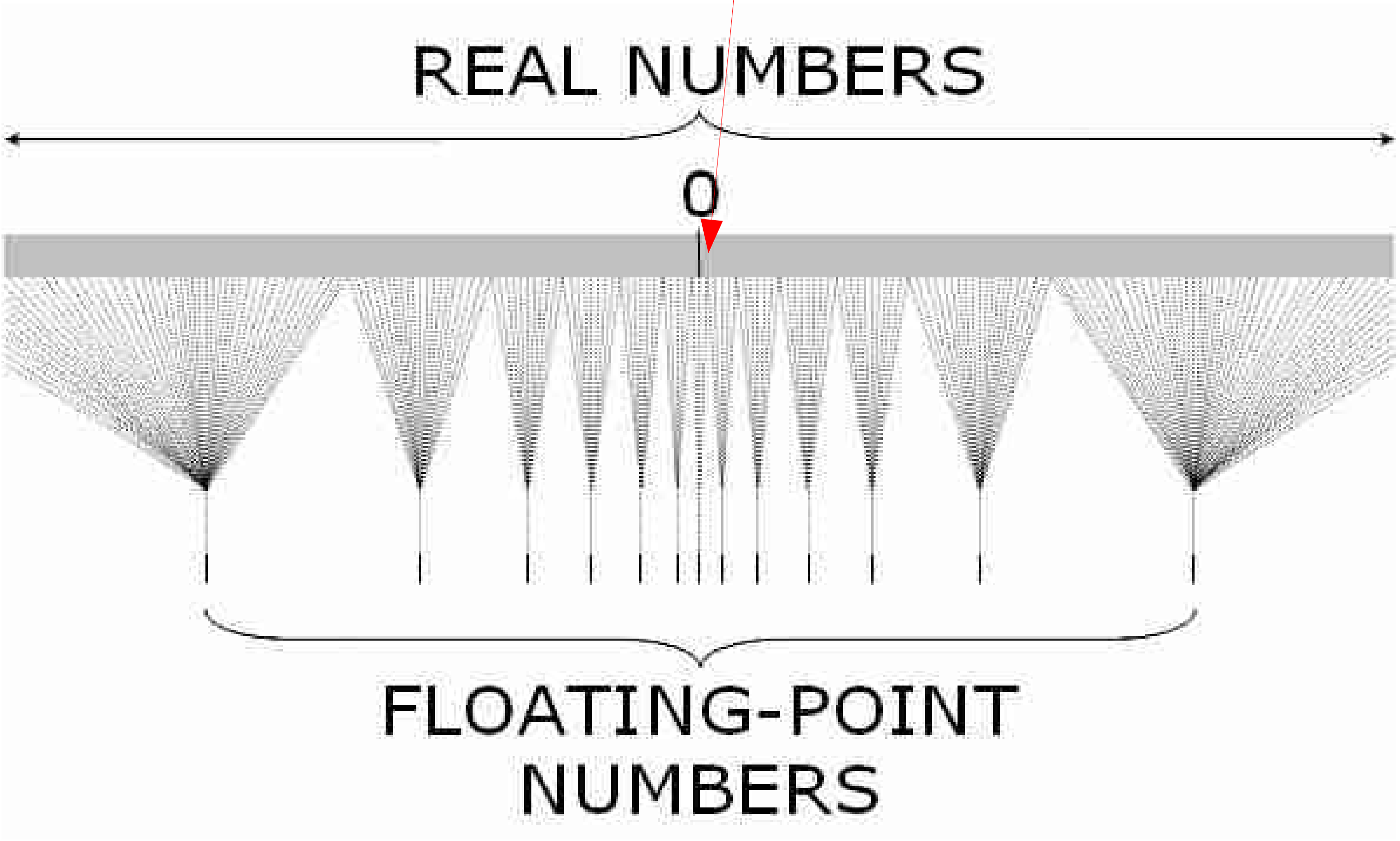
$$\vec{L}_A^{(k)}(c) = \Big(\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}_S^{(i)}(c)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}_S^{(j)}(c)\Big)$$



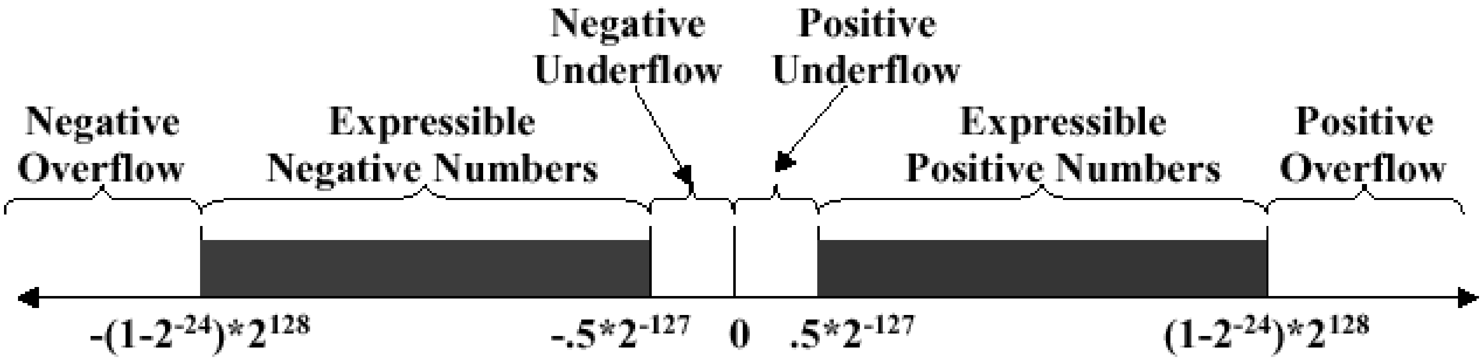
Position c

Numerical Underflow

Conditional likelihood values become so small that they can not be represented on a computer any more → **underflow !!!!**



Overflow & Underflow



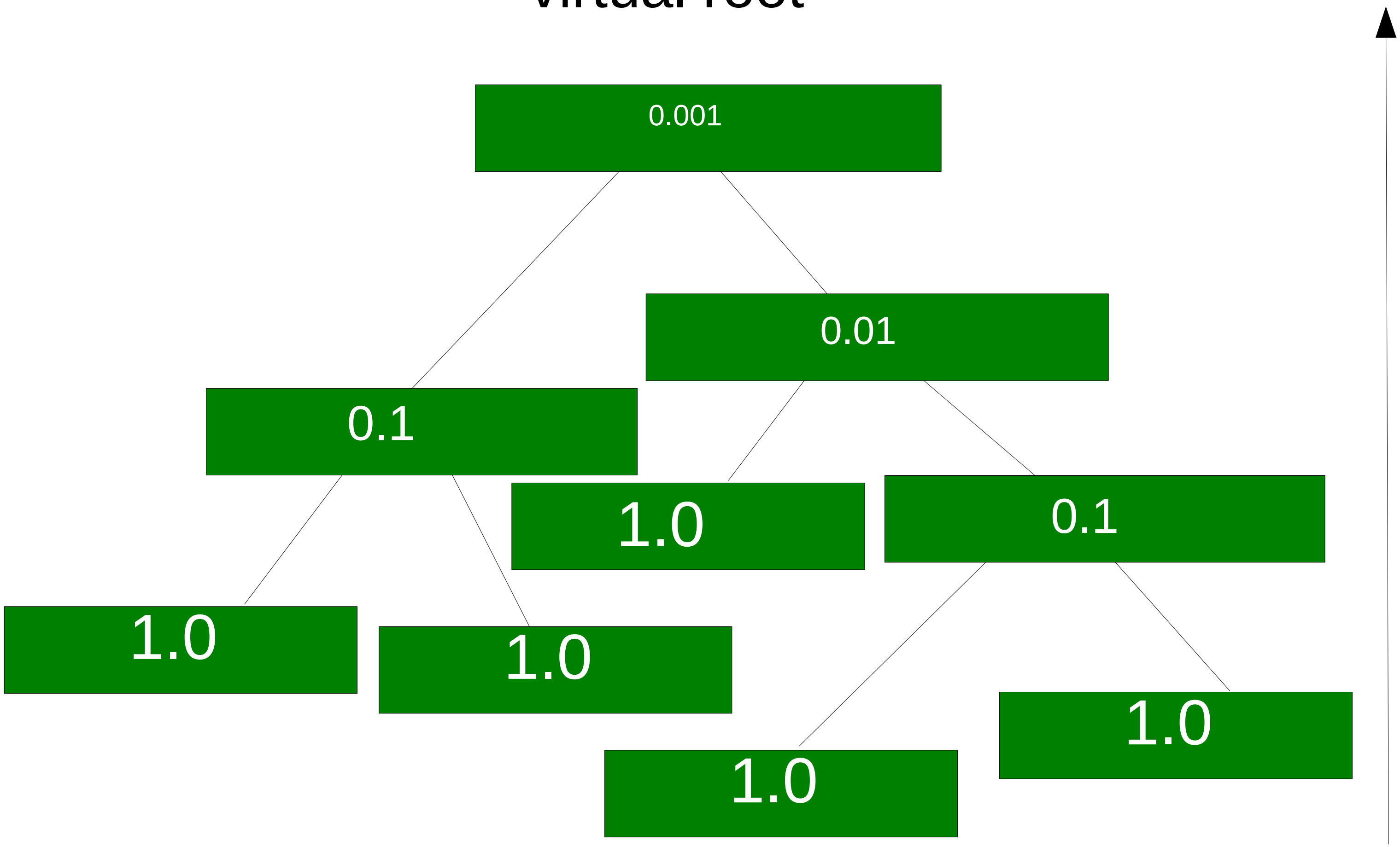
IEEE 754 standard for 32-bit floating point numbers

1 bit	sign
8 bits	exponent
23 bits	significand

Post-order Traversal

preventing underflow

virtual root



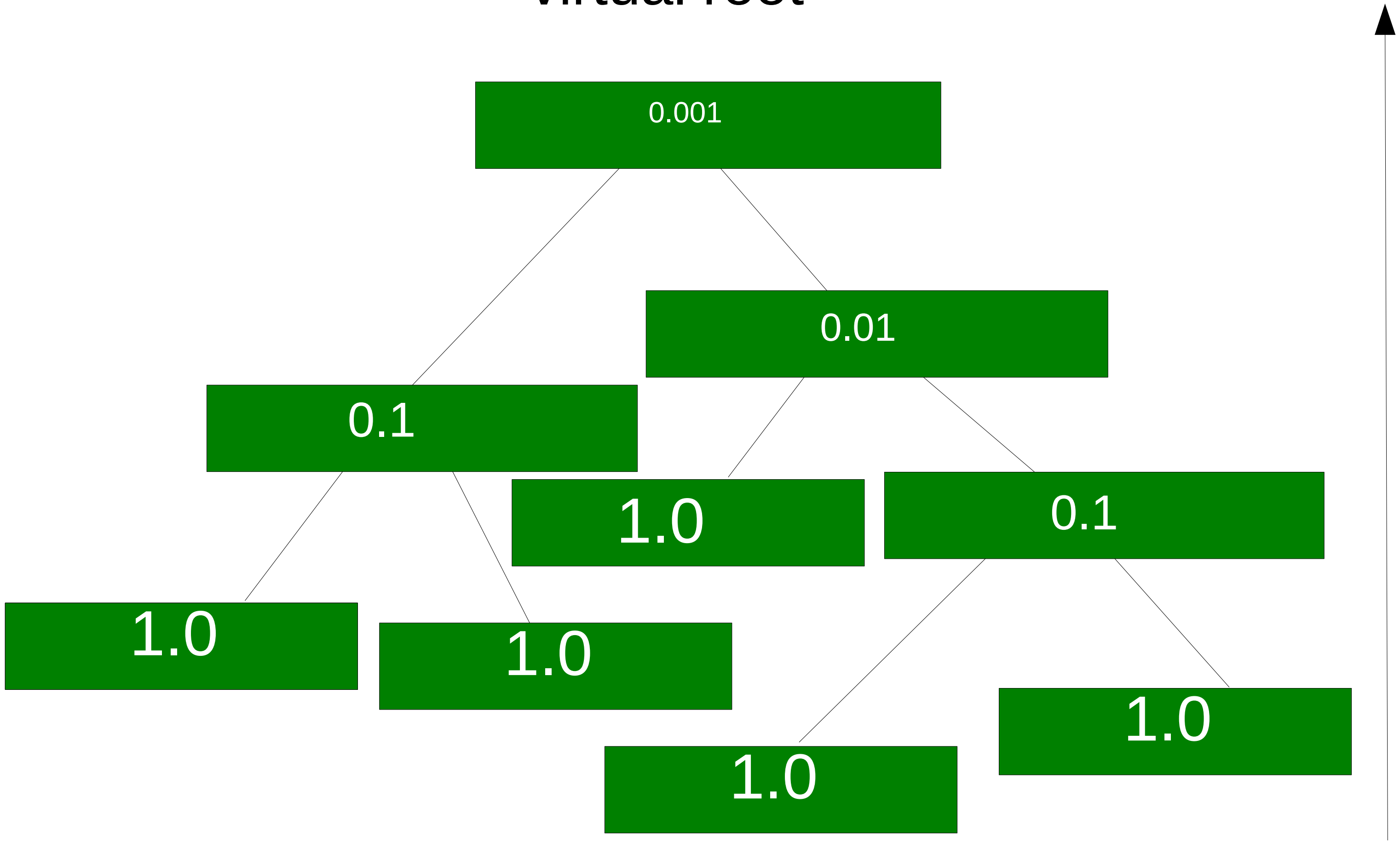
Values in conditional likelihood vectors get smaller and smaller as we move to the root

Post-order Traversal

preventing underflow

We need to apply numerical scaling techniques to avoid underflow!

virtual root



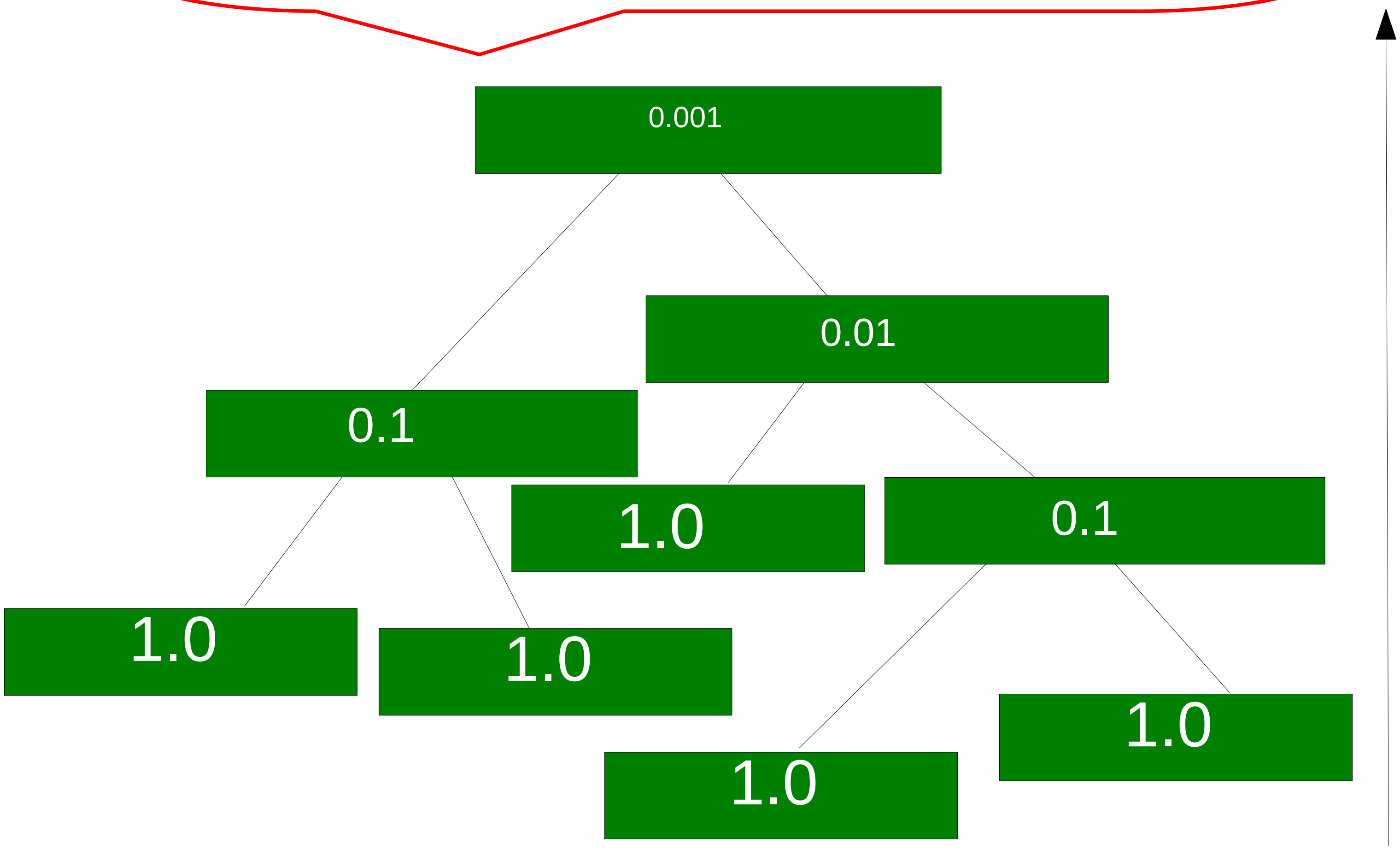
Values in conditional likelihood vectors get smaller and smaller as we move to the root → this needs to be handled!

Post-order Traversal

preventing underflow

Typical approach

1) Check if values are too small
2) If so multiply with some large number
3) Undo those scaling multiplications (somehow) in the end
4) for likelihood this undoing is easy



Values in conditional likelihood vectors get smaller and smaller as we move to the root → this needs to be handled!

What went wrong?

- For DNA models without rate heterogeneity this scaling approach worked fine

  → check if all *4* conditional likelihoods at a given CLV and site are smaller than a minimum & multiply with large number

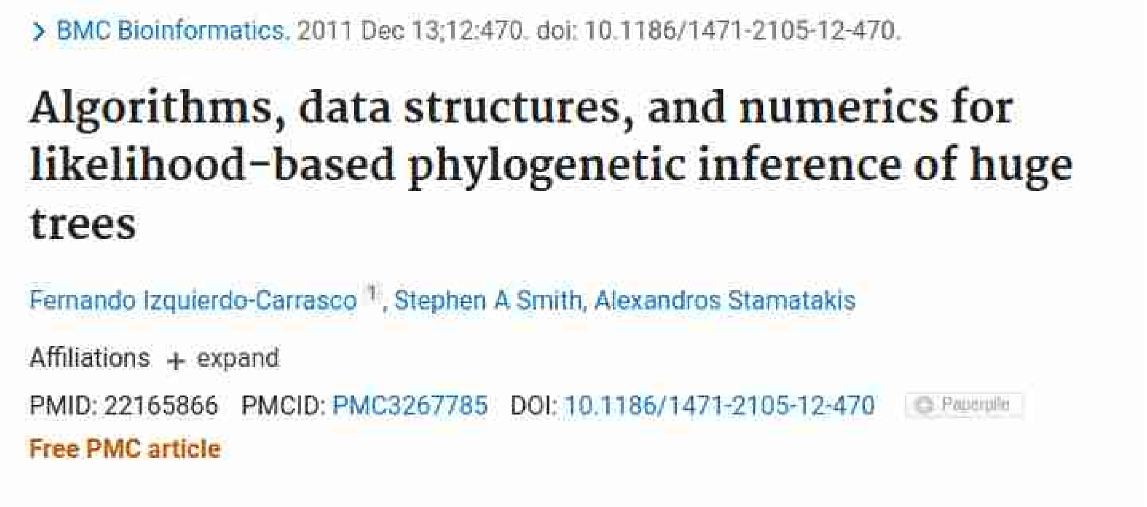
- For DNA models with rate heterogeneity this doesn't always work

  → jointly checking that all *16* conditional likelihoods for the *4* typical discrete rates are smaller than a minimum doesn't work

  → the spread of the values is too large because of the distinct rate categories

  → scale individually per rate category

  → higher computational cost



› BMC Bioinformatics. 2011 Dec 13;12:470. doi: 10.1186/1471-2105-12-470.

Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees

Fernando Izquierdo-Carrasco [1], Stephen A Smith, Alexandros Stamatakis

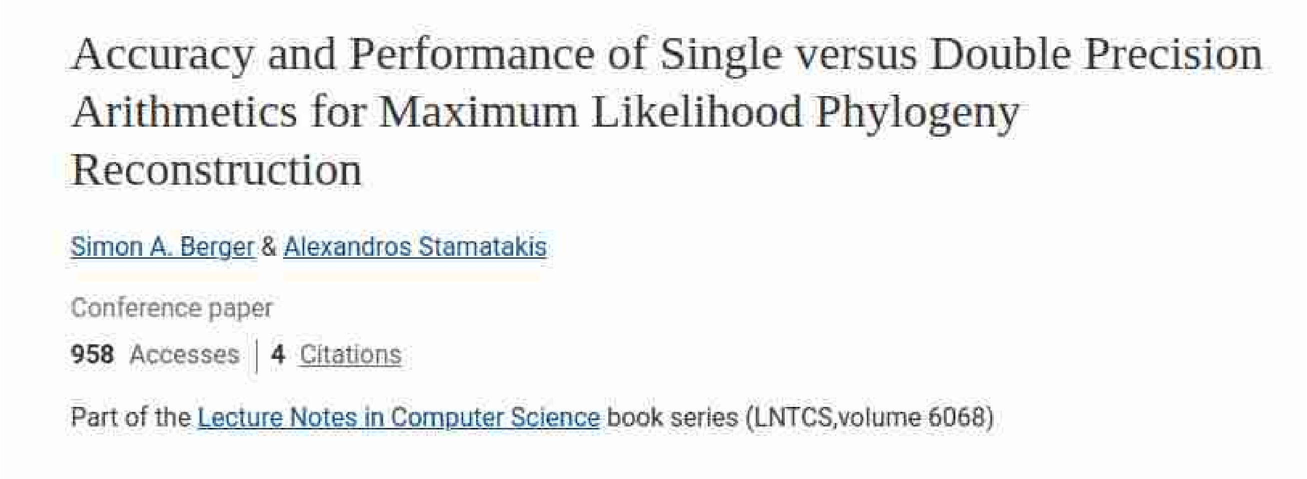
Affiliations + expand

PMID: 22165866 PMCID: PMC3267785 DOI: 10.1186/1471-2105-12-470

Free PMC article

What went wrong?

- We know that likelihood claculations are compute- and memory-intensive
- So why not use single-precision (32 bit) instead of double precision (64 bit) floating point values?
- Numerics for Maximum Likelihood break down
- 10-fold increase in scaling multiplications when using single precision



Accuracy and Performance of Single versus Double Precision Arithmetics for Maximum Likelihood Phylogeny Reconstruction

Simon A. Berger & Alexandros Stamatakis

Conference paper

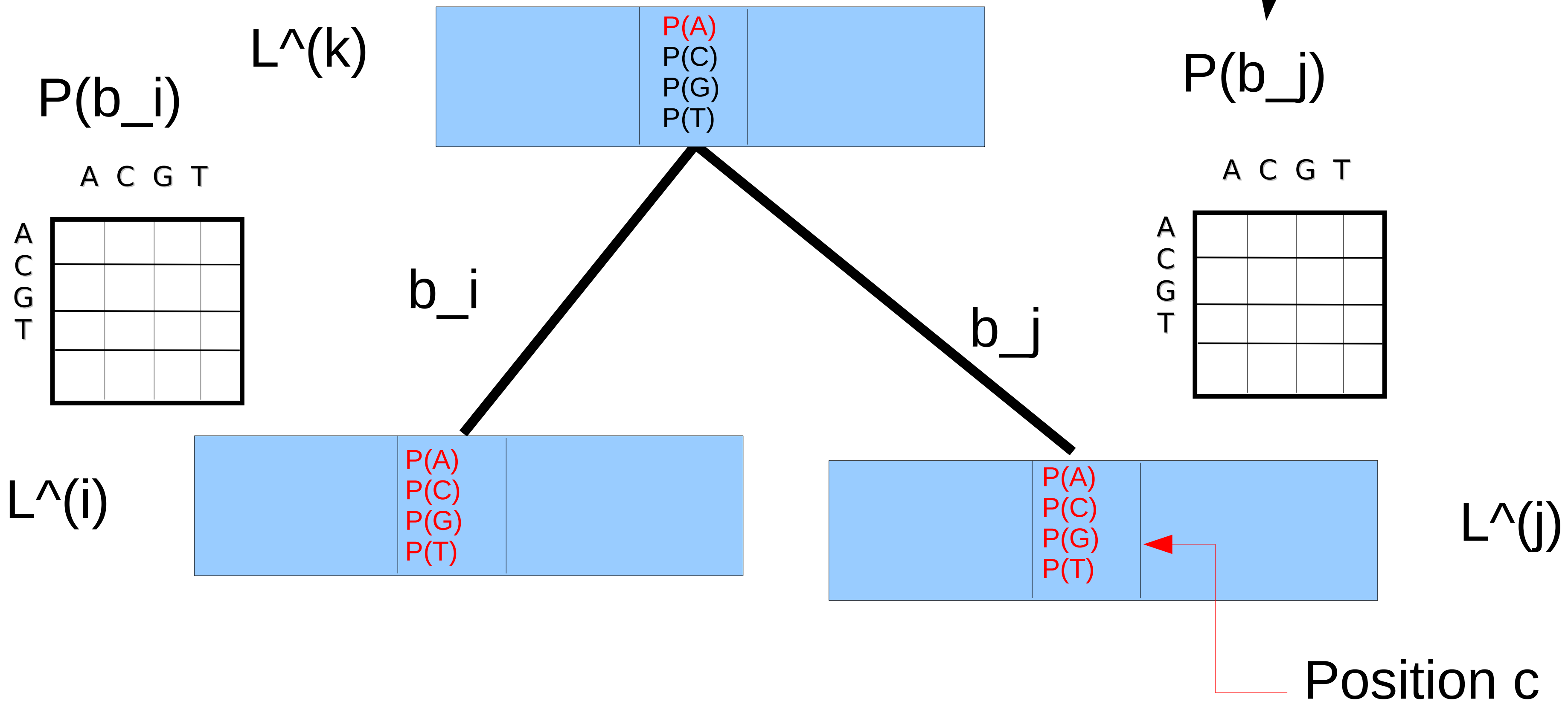
958 Accesses | 4 Citations

Part of the Lecture Notes in Computer Science book series (LNTCS,volume 6068)

Felsenstein pruning

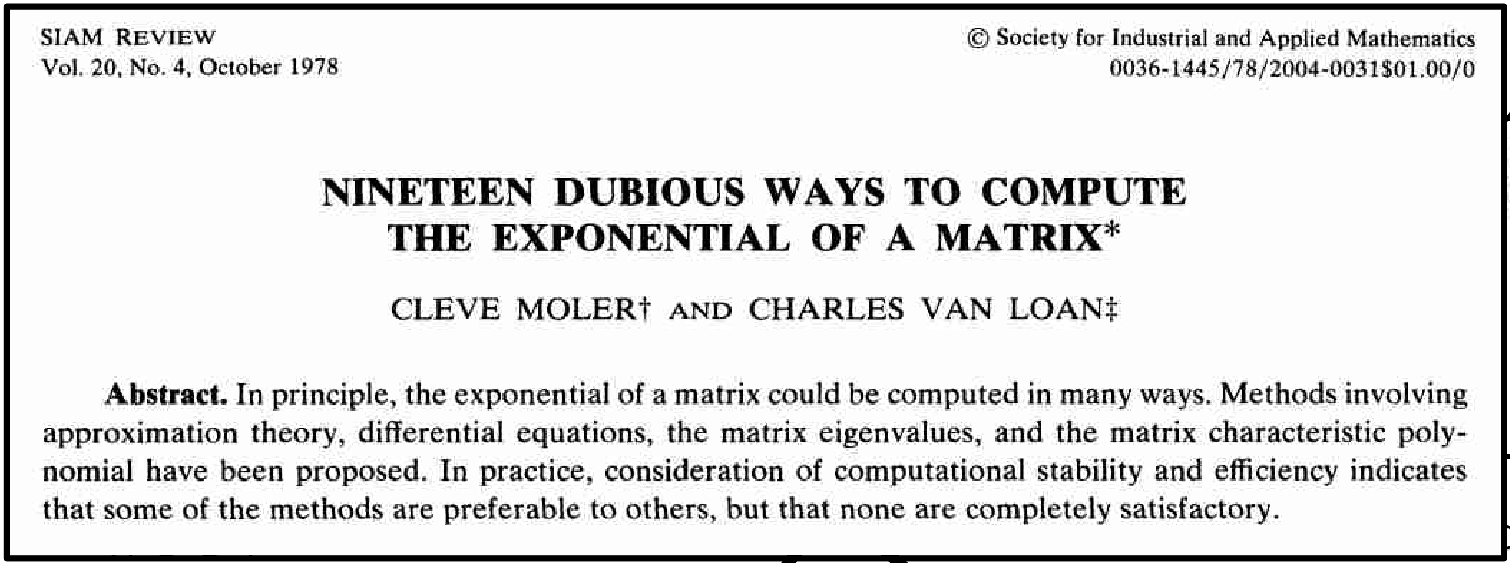
P(t) = e^{Qt} is numerically not easy

$$\vec{L}_A^{(k)}(c) = (\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}_S^{(i)}(c))(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}_S^{(j)}(c))$$



Felsenstein pruning

P(t) = e^{Qt} is numerically not easy



SIAM Review
Vol. 20, No. 4, October 1978

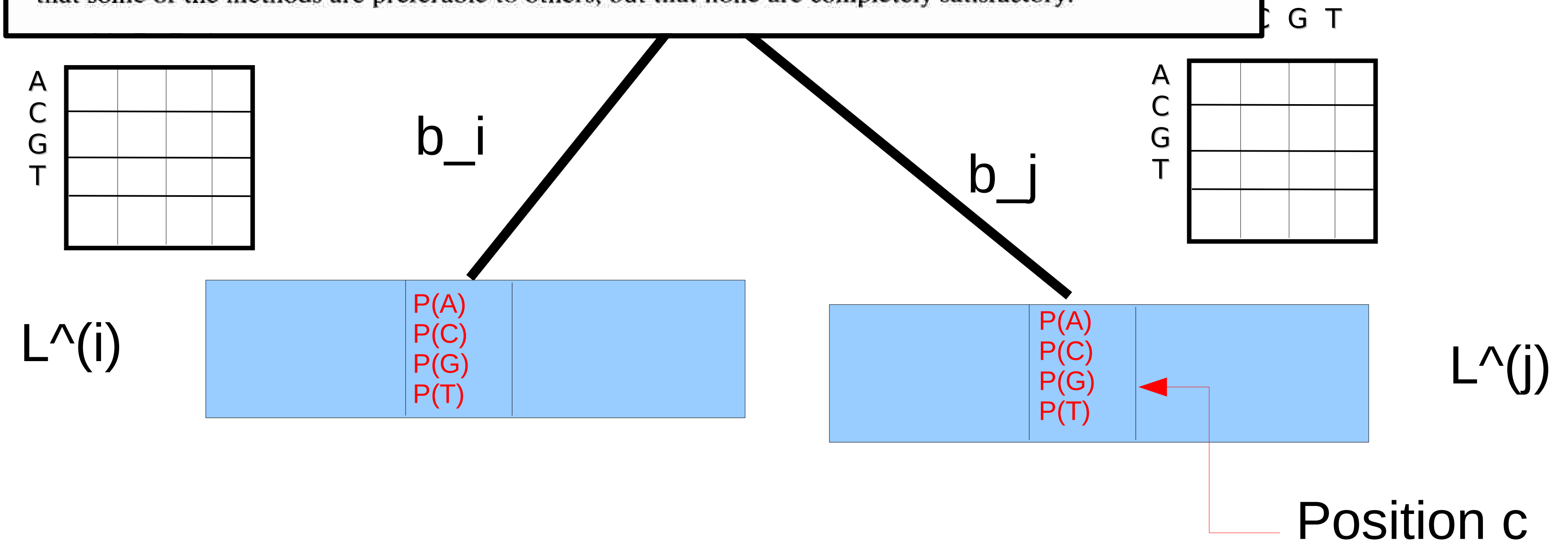
NINETEEN DUBIOUS WAYS TO COMPUTE THE EXPONENTIAL OF A MATRIX*

CLEVE MOLER† AND CHARLES VAN LOAN‡

Abstract. In principle, the exponential of a matrix could be computed in many ways. Methods involving approximation theory, differential equations, the matrix eigenvalues, and the matrix characteristic polynomial have been proposed. In practice, consideration of computational stability and efficiency indicates that some of the methods are preferable to others, but that none are completely satisfactory.

$)\vec{L}_S^{(j)}(c))$

j)



What went wrong?

- In RAxML we used the matrix exponential function from the book - *Numerical Recipees in C*
- Uses Eigenvector/Eigenvalue decomposition
- Especially the Intel icc compiler tended to be very aggressive when trying to optimize this function

  → numerical breakdown
- Solution

```
eigen.o : eigen.c $(GLOBAL_DEPS)
             $(CC) -c -o eigen.o eigen.c
```

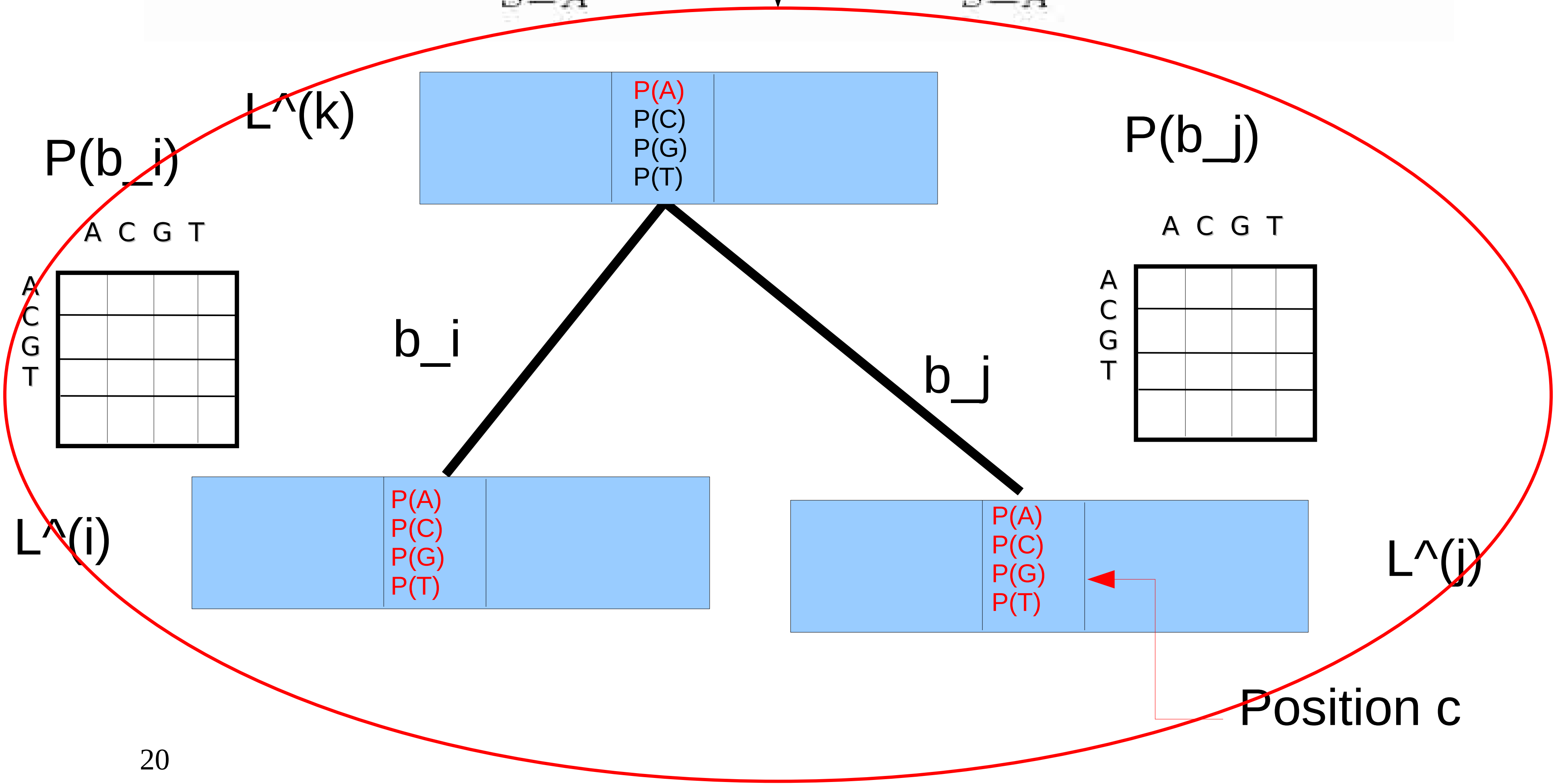
  Compile eigenvector decomposition function without optimization flags

Are you occasionally using PCA?

- Principal Component Analysis
- Also relies on Eigenvector/Eigenvalue decomposition → **beware !!!!**

Consider that you only want to compute this triplet of conditional likelihood vectors of fixed length *n*.
L^(i), L^(j), P(b_i), P(b_j) are given as input and you just compute *L^(k)* as output of a micro-benchmark.
What do you expect the run-times to be if you just provide different input vectors *L^(i)’, L^(j)’* but again of length *n*?

$$\vec{L}^{(k)}_A(c) = \Big(\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}^{(i)}_S(c)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}^{(j)}_S(c)\Big)$$

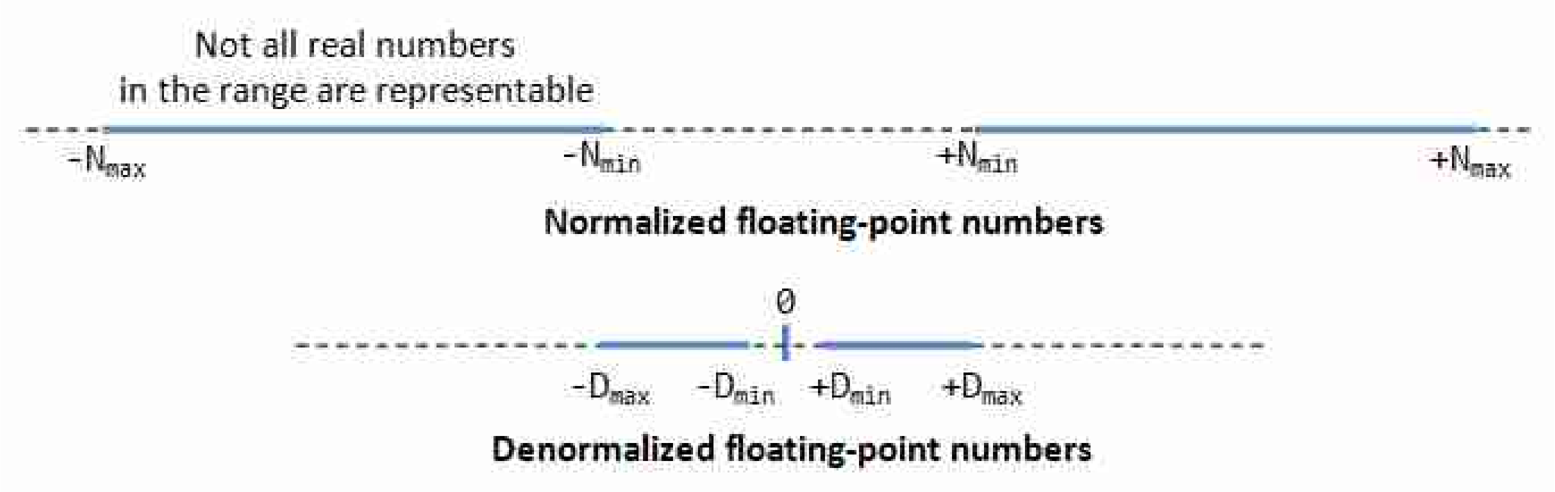


What went wrong?

- When developing phylogenetic placement methods, we observed some inexplicable run time deviations of about 50% for exactly this operation
- It didn’t make any sense since we executed *n* times the exact same arithmetic operations, just on different input data

→ until we learned about de-normalized floating point values

Denormalized Floating Point Numbers


Not all real numbers
in the range are representable
-Nmax
-Nmin
+Nmin
+Nmax
Normalized floating-point numbers
0
-Dmax
-Dmin
+Dmin
+Dmax
Denormalized floating-point numbers

Intended to allow for gradual underflow to zero

When de-normalized values are encountered, the processing cost inside the CPU for multiplications and additions is increased.

→ the runtimes are input data dependent !
→ Problem with **reproducibility of run time performance benchmarks**

Denormalized Numbers

- De-normalized floating point numbers and their impact on run-times and performance benchmark
  - J. Björndalen, O. Anshus: “Trusting floating point benchmarks-are your benchmarks really data-independent?” Applied Parallel Computing. State of the art in Scientific Computing 2010; pp 178-188, Springer.
  - Alexandre F. Tenca, Kyung-Nam Han, David Tran: “Performance Impact of Using Denormalized Numbers in Basic Floating-point Operations” IEEE, Forty-First Asilomar Conference on Signals, Systems and Computers, 2007.
- The concrete example with Conditional Likelihood Vector computations that yielded highly diverging run times due to de-normalized floating point numbers can be found here https://github.com/stamatak/denormalizedFloatingPointNumbers

The story so far

- Flaoting point number are an imperfect mapping of the real numbers to machine numbers
- All sorts of numerical instabilities can arise
- There can be in issue when trying to reproduce performance results
- → Distinct processor types (hardware architectures) may be handling denormalized floating point numbers differently!

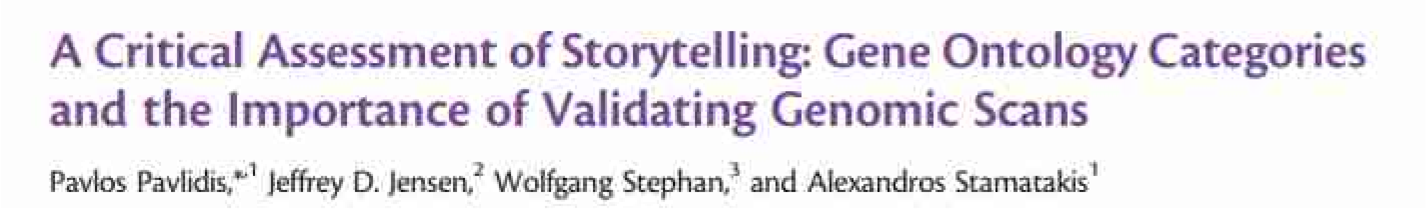
Does weird stuff only happen for floating point?

- It is more likely to happen

  → with integer arithmetic there exists an exact mapping of integers to machine numbers

  → however overflow can still occur !!!
- But what if the same integer random number seed yields a different series of random numbers ???
- We need random numbers a lot in our tools
- Specifying a random number seed should normally guarantee that the same sequence of random numbers is generated → reproducibility of results!!!

What went wrong?

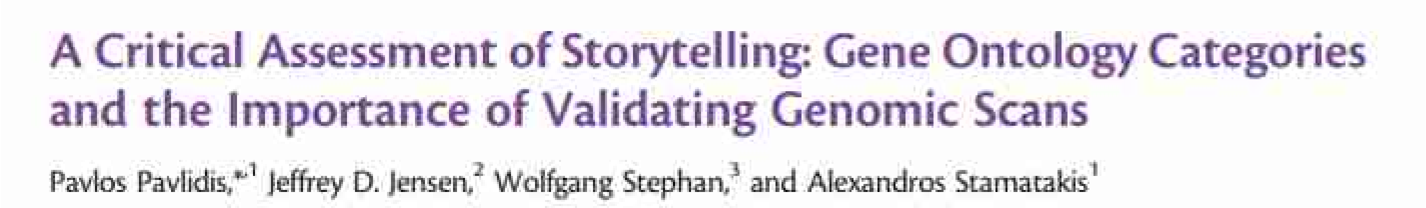


A Critical Assessment of Storytelling: Gene Ontology Categories and the Importance of Validating Genomic Scans

Pavlos Pavlidis,[*,1] Jeffrey D. Jensen,[2] Wolfgang Stephan,[3] and Alexandros Stamatakis[1]

- We were not able to reproduce our own results on a different machine!!!
- Any ideas?

What went wrong?



A Critical Assessment of Storytelling: Gene Ontology Categories and the Importance of Validating Genomic Scans

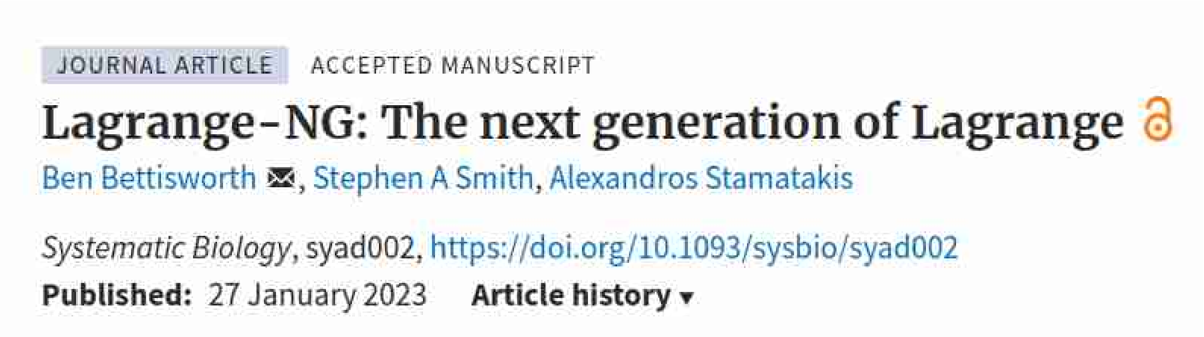
Pavlos Pavlidis,[*,1] Jeffrey D. Jensen,[2] Wolfgang Stephan,[3] and Alexandros Stamatakis[1]

- We were not able to reproduce our own results on a different machine!!!
- Any ideas?
- *The constant changes in computer architectures, compilers, and scientific libraries further complicate the reproducibility of experiments. For example, in the current analysis, MaCS (v.0.4c) produced different results when using identical random number seeds but different versions of the boost library (www.boost.org, v1.33 and v1.40) because of code changes in the random number generator implementation (supplementary section X, Supplementary Material online). We observed this behavior by pure chance ...*

Take Home Message

- Strict version control !!!!
- Not only control the version of the code you used but also of the external libraries it relies upon
- Ideally, don’t rely on external libraries when developing own code !!!

Compiler Optimization
What went wrong?



JOURNAL ARTICLE ACCEPTED MANUSCRIPT

Lagrange-NG: The next generation of Lagrange

Ben Bettisworth ✉, Stephen A Smith, Alexandros Stamatakis

Systematic Biology, syad002, https://doi.org/10.1093/sysbio/syad002

Published: 27 January 2023 **Article history** ▾

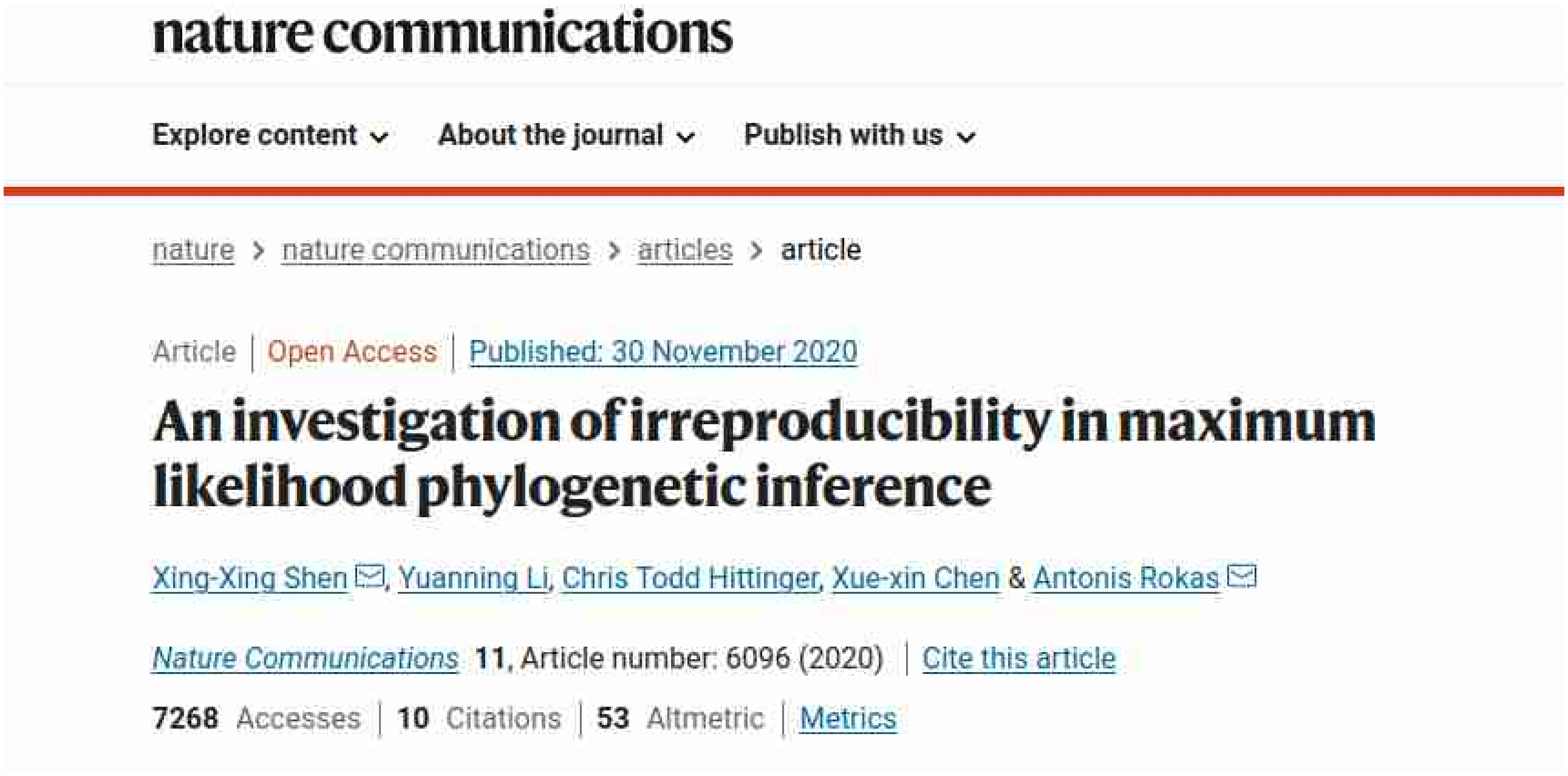
- We recently re-designed/re-wrote the popular `Lagrange` biogeography tool
- Initially, we were very excited as we easily got 10-fold speedups
- It turned out that:

  "we identified and corrected a configuration error in the process of building `Lagrange`*, where important compiler optimization options were not properly utilized. Fixing this configuration error alone increased the computational efficiency of the original* `Lagrange` *by up to 10x. While this error is easy to overlook, yet trivial to fix, we assume that many past Lagrange analyses were conducted using the unoptimized code"*
- `Lagrange` was being distributed in unoptimized form (without the `-O2` flag) for many years!

Floating Point
The Root of All Evil

- Computational science mostly relies on floating-point intensive codes
- How do we verify these codes?
  - Numerical instabilities
  - Unstable run-time performance benchmarks
  - Distinct round of error propagation
- We stand on shaky grounds
- Scientists using those codes assume numerical results are exact

Reproducibility – no surprises



nature communications

Explore content ⌄ About the journal ⌄ Publish with us ⌄

nature > nature communications > articles > article

Article | Open Access | Published: 30 November 2020

An investigation of irreproducibility in maximum likelihood phylogenetic inference

Xing-Xing Shen ✉, Yuanning Li, Chris Todd Hittinger, Xue-xin Chen & Antonis Rokas ✉

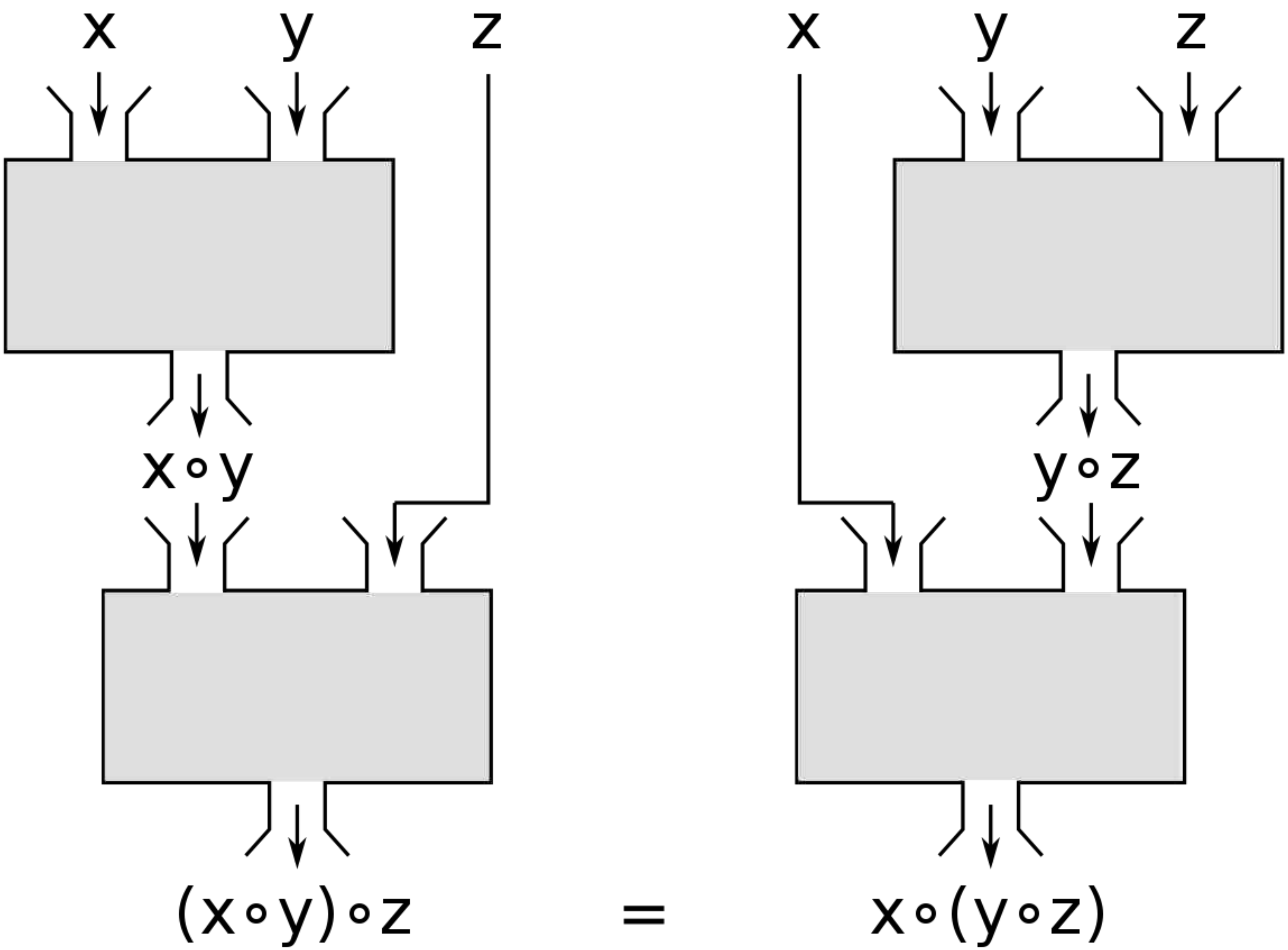
Nature Communications **11**, Article number: 6096 (2020) | Cite this article

7268 Accesses | **10** Citations | **53** Altmetric | Metrics

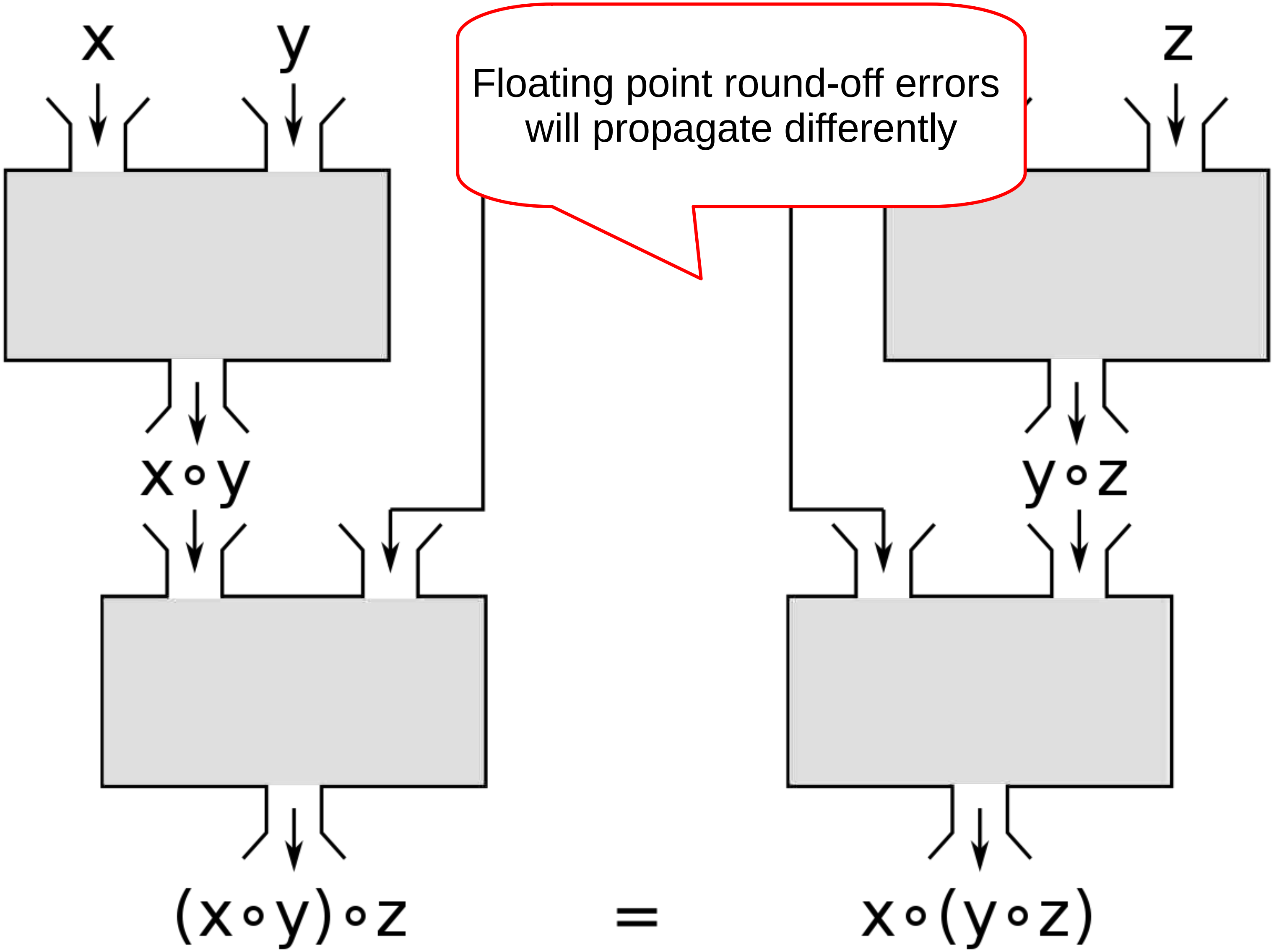
Outline

- The root of all evil
- **“Sequential” Computations**
- Parallel Computations
- Software Quality

Associativity



Associativity



Floating point round-off errors will propagate differently

Reproducibilty

- Under floating point

  (a + b) + c ≠ a + (b + c)

  → Order of operations will affect the result

  → round off errors due to imperfect representation of real numbers
  will propagate differently

- Manual code optimization or automatic code optimization with compilers (`gcc -O2` flag, for instance) always assumes that

  (a + b) + c = a + (b + c)

  → Same code, same input, same options, at different optimization levels can yield different results

  → Same code, same input, same options, run on a distinct CPU architecture can yield different result

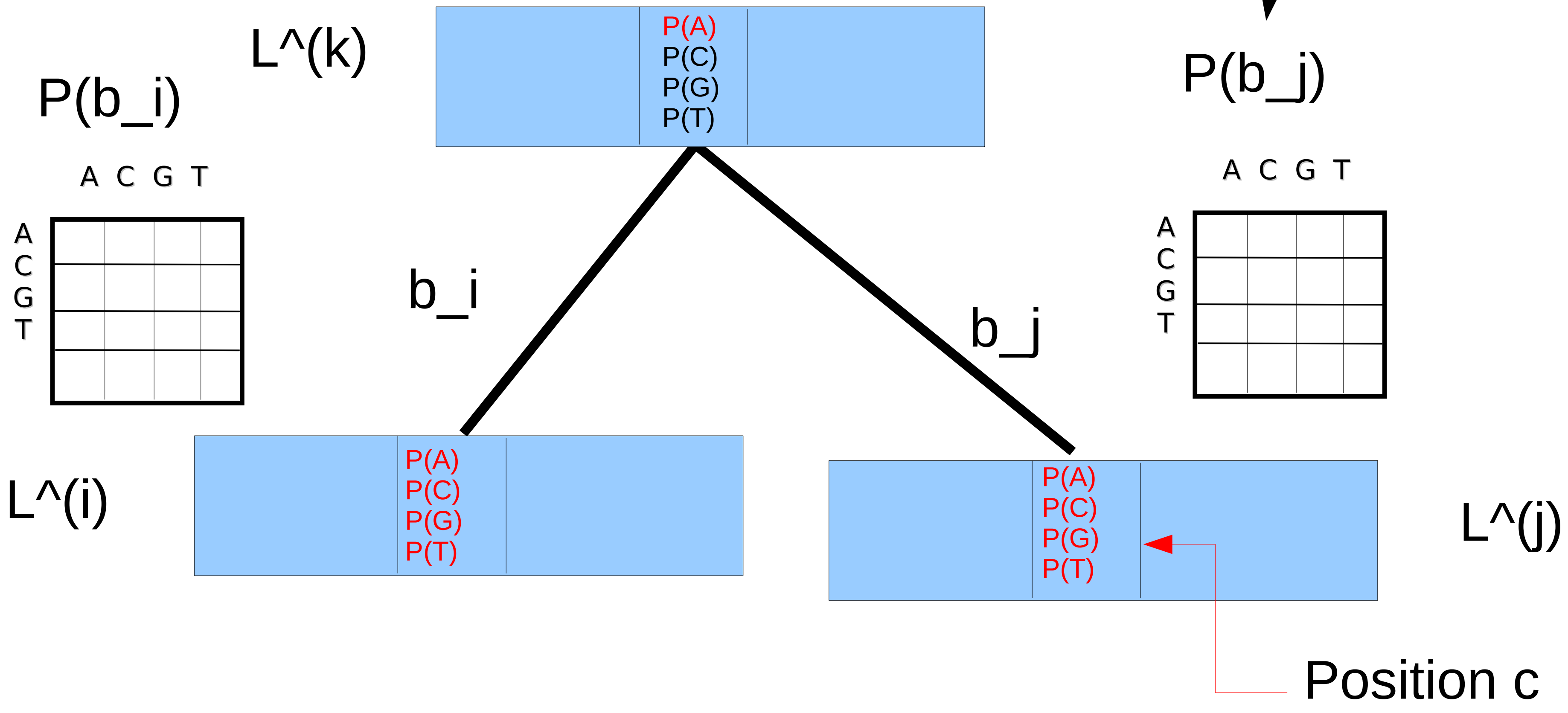
  → on GPUs this is even more likely to happen

  → for instance, we couldn’t get Lagrange-NG to run in a numerically stable way on a GPU

Felsenstein pruning

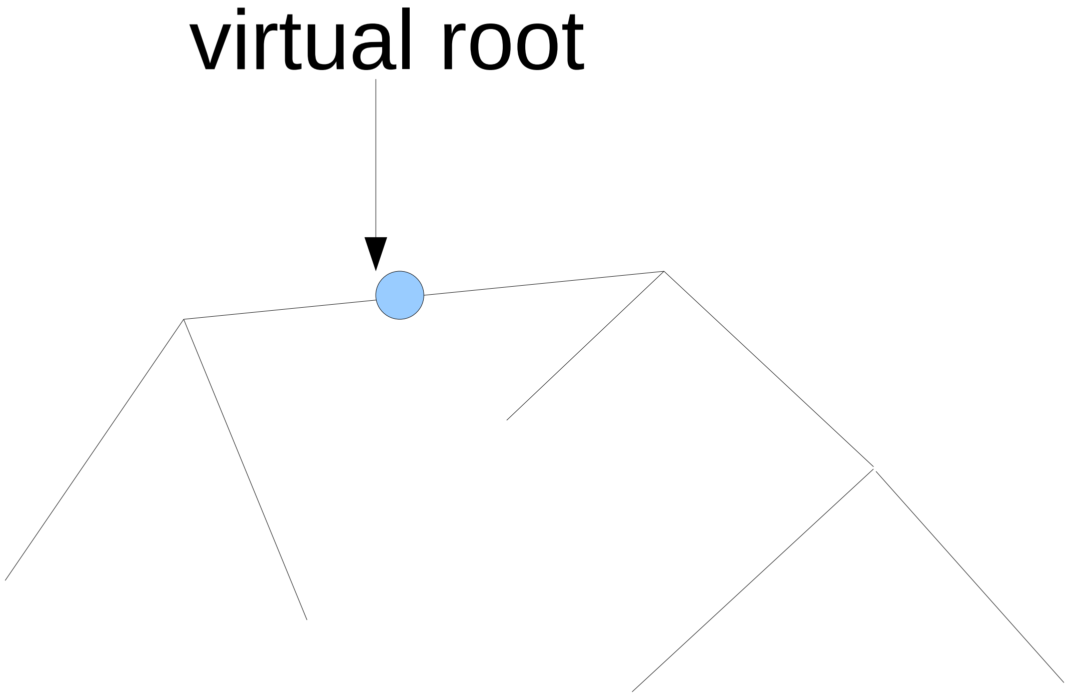
There are numerous ways to re-order these associative computations!

$$\vec{L}_A^{(k)}(c) = \Big(\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}_S^{(i)}(c)\Big)\Big(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}_S^{(j)}(c)\Big)$$

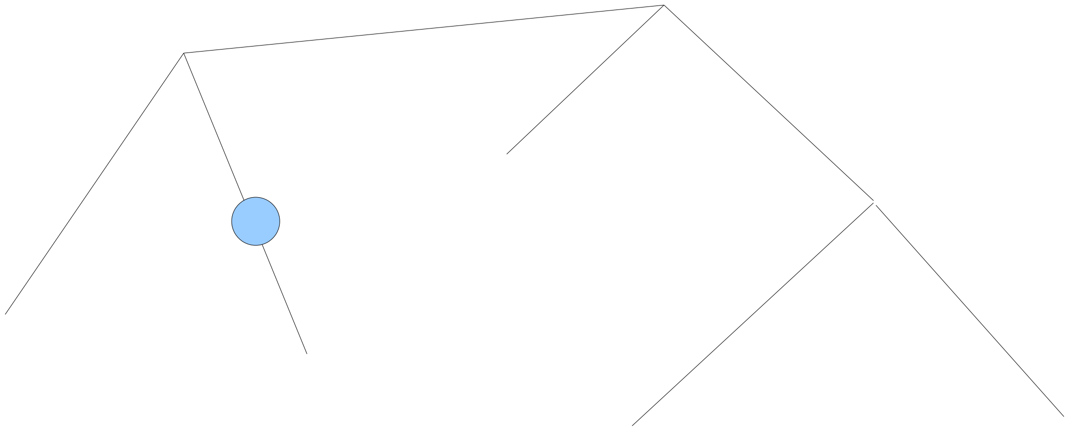


An Example:
Post-order Traversal

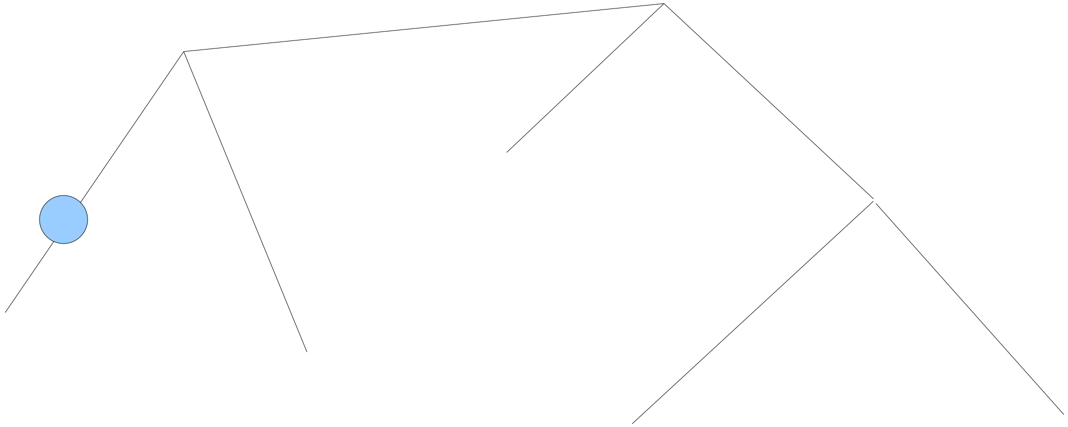
Different virtual root placements
also change the order of operations



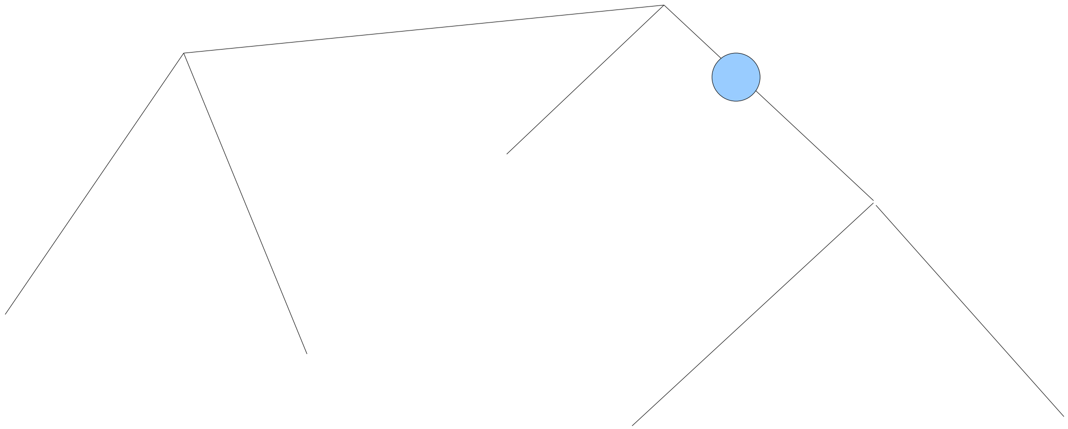
Post-order Traversal



Post-order Traversal

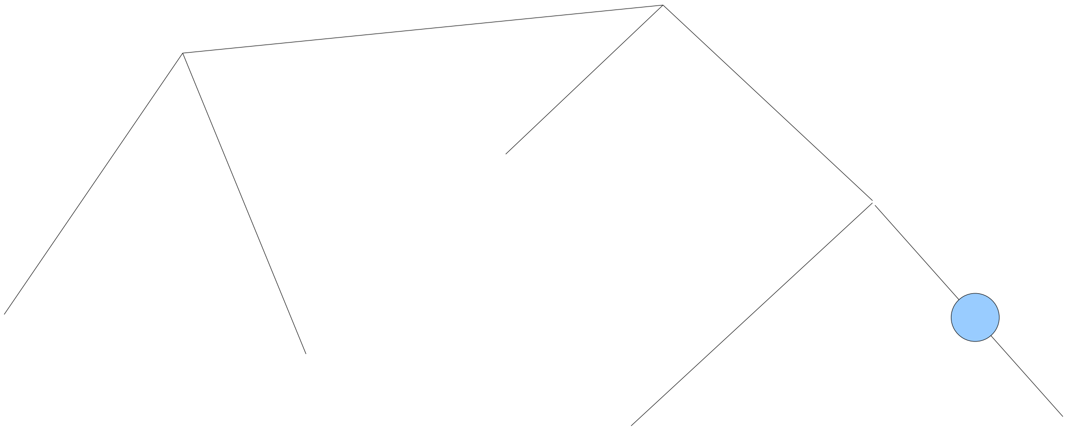


Post-order Traversal

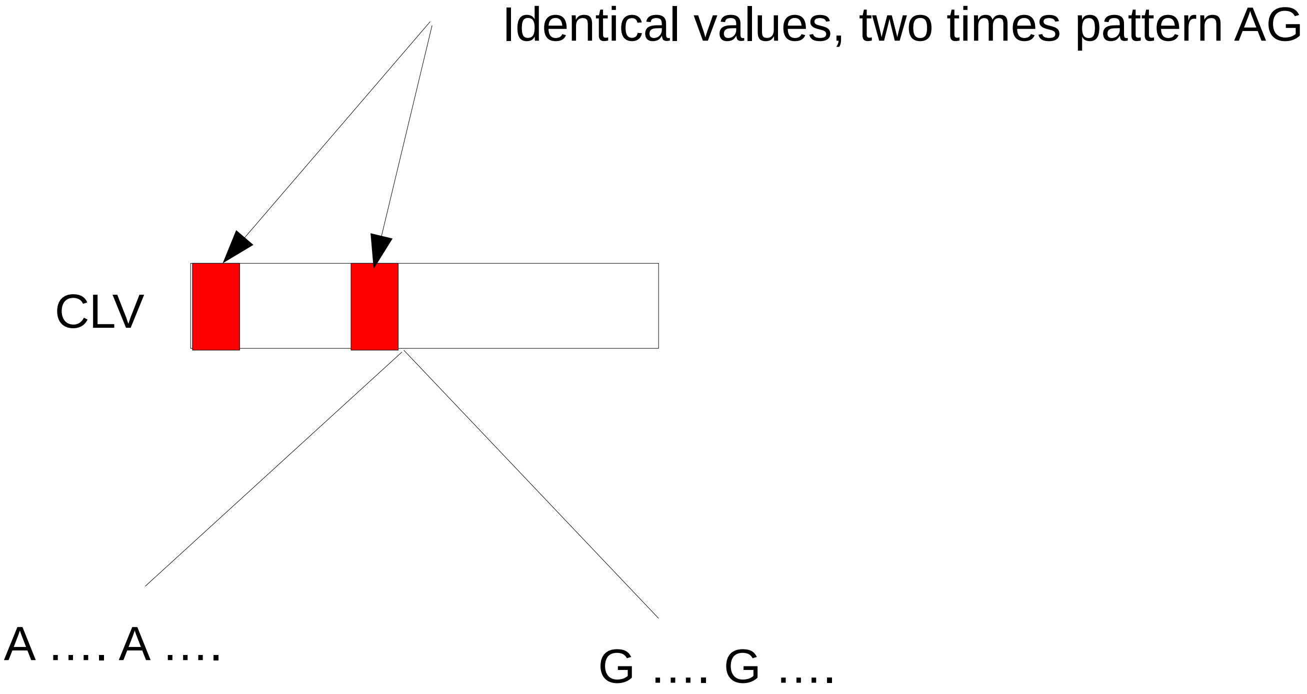


Post-order Traversal

In which order do we actually optimize the branch lengths by the way?



More Re-Ordering: Repeating Patterns



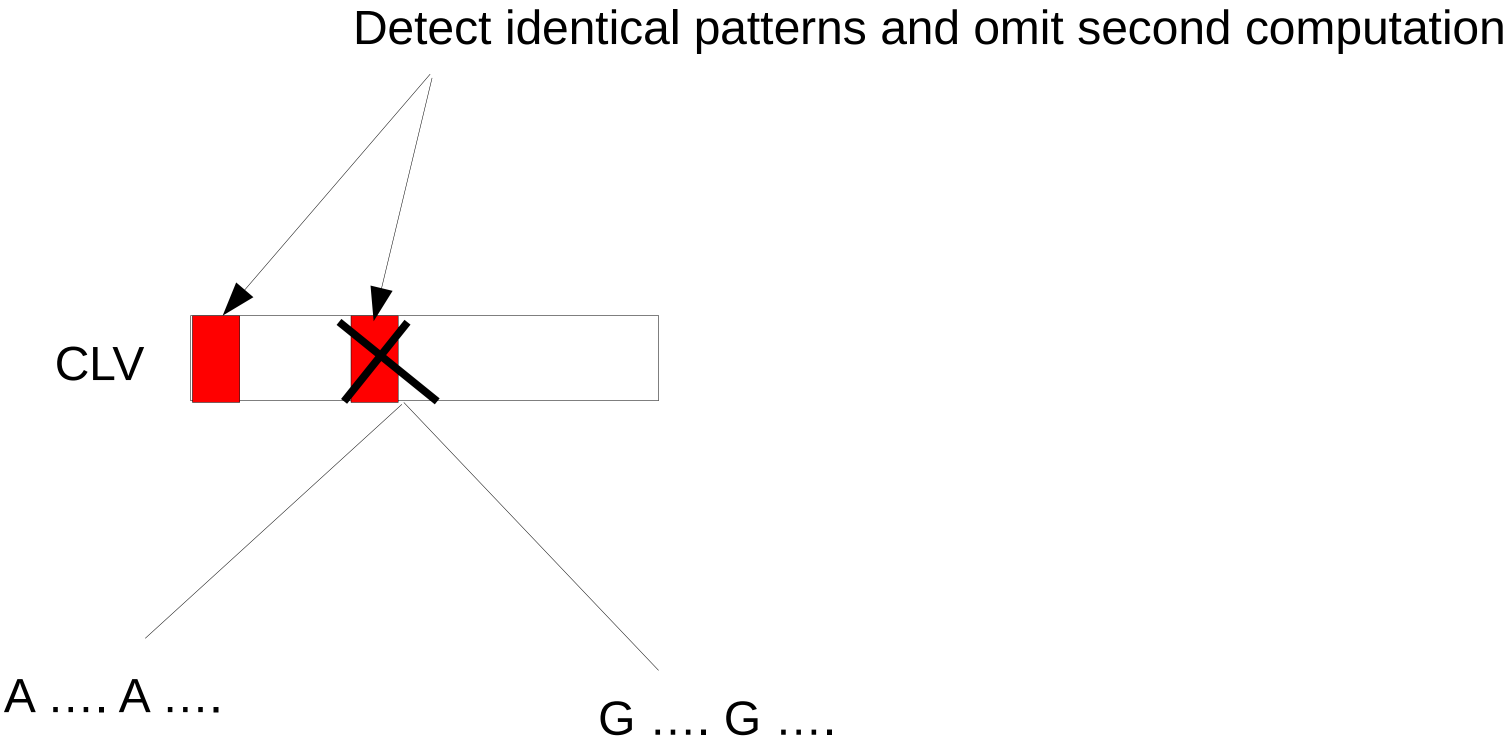
Identical values, two times pattern AG

CLV

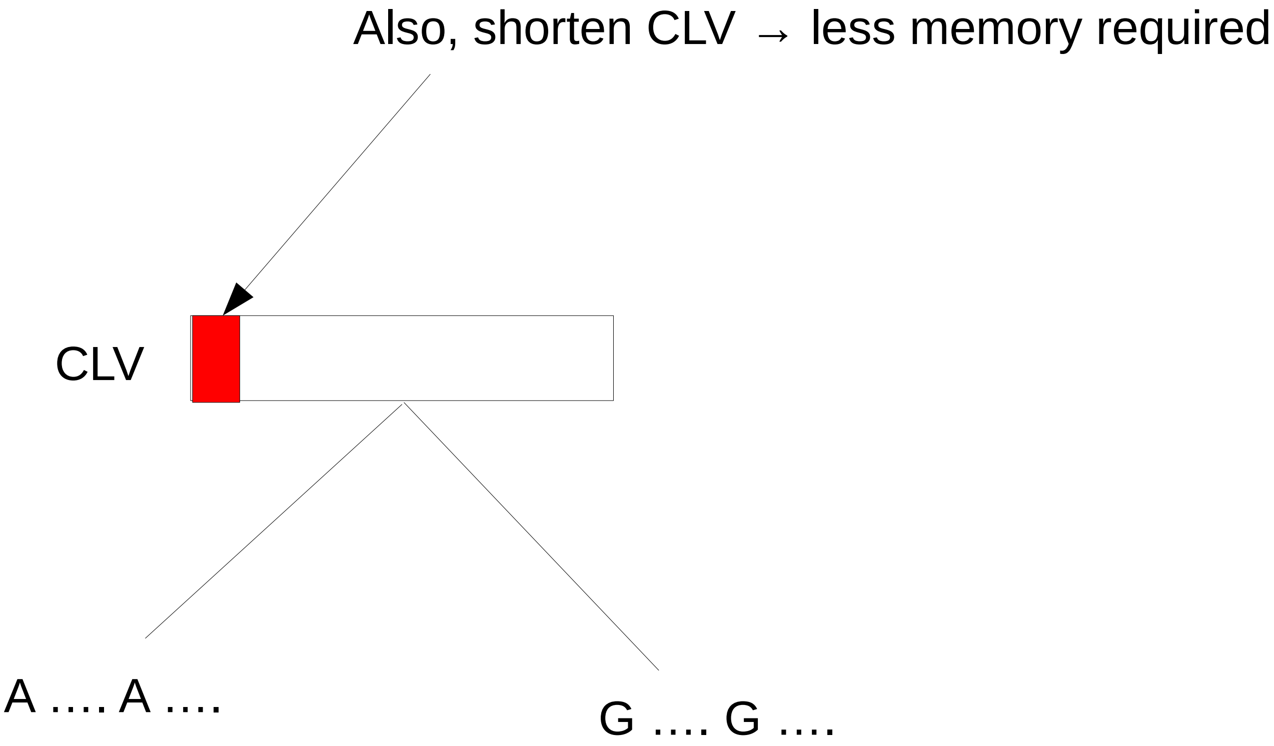
A .... A ....

G .... G ....

Repeating Patterns



Repeating Patterns



Also, shorten CLV → less memory required

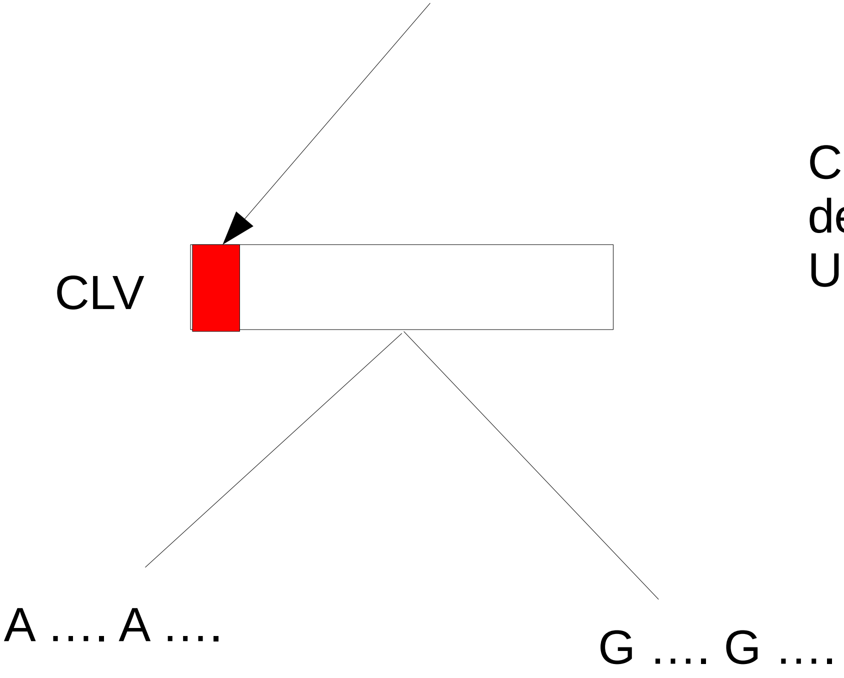
CLV

A .... A ....

G .... G ....

Repeating Patterns

Also, shorten CLV → less memory required



CLV

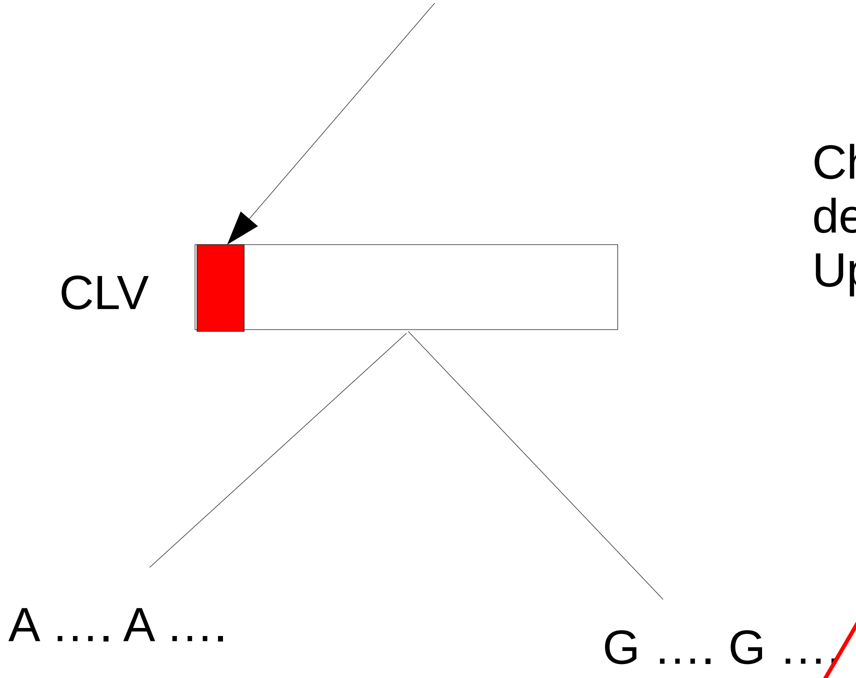
A .... A ....

G .... G ....

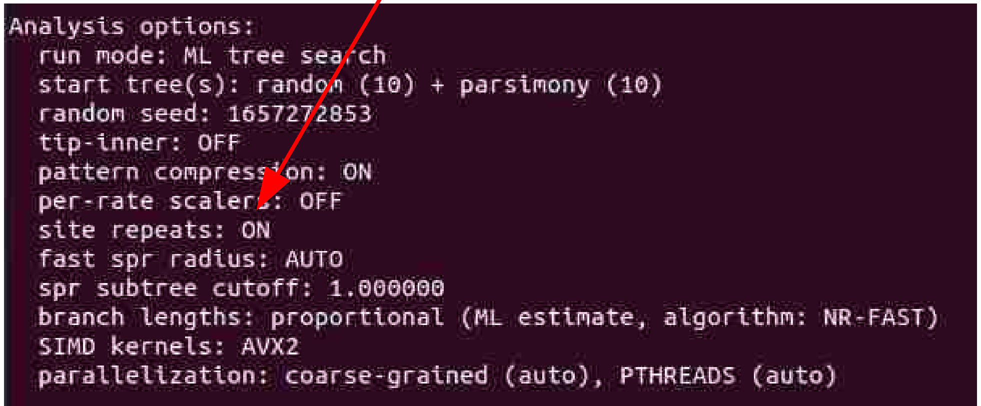
Challenge: Efficient data structure to detect & store repeats
Up to 10-fold run-time improvements

Repeating Patterns

Also, shorten CLV → less memory required



Challenge: Efficient data structure to detect & store repeats
Up to 10-fold run-time improvements



```
Analysis options:
  run mode: ML tree search
  start tree(s): random (10) + parsimony (10)
  random seed: 1657272853
  tip-inner: OFF
  pattern compression: ON
  per-rate scalers: OFF
  site repeats: ON
  fast spr radius: AUTO
  spr subtree cutoff: 1.000000
  branch lengths: proportional (ML estimate, algorithm: NR-FAST)
  SIMD kernels: AVX2
  parallelization: coarse-grained (auto), PTHREADS (auto)
```

Reproducibilty

- Under floating point

  $(a + b) + c \neq a + (b + c)$

  → Order will affect the result – distinct round off error propagation
- Sequential execution:
  - Tree inference might yield different trees if you use different compiler
  - Tree inference might yield different trees if you use `SSE3` (128 bits) or `AVX` (256 bits)
- We have observed this on real data !

Vector Instructions

parallelism within a *single* CPU

Vector Instructions

parallelism within a *single* CPU

- Vector Architectures: *SSE3, AVX, AVX-512*
  - Execute the same operation simultaneously on more than one value/datum

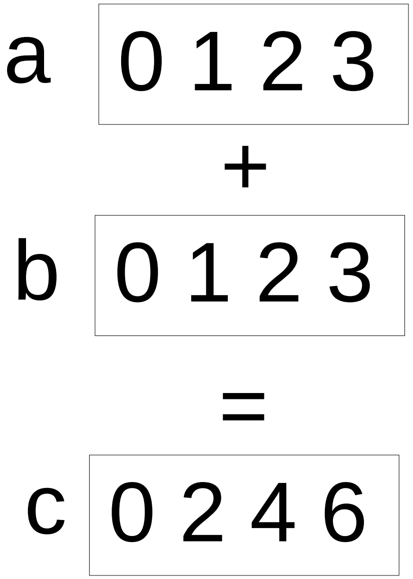
Vector Instructions

parallelism within a *single* CPU

- Vector Architectures: *SSE3, AVX, AVX-512*
  - Execute the same operation simultaneously on more than one value/datum
- GPUs are also just vector processors!

Vector Instructions

- `RAxML-NG` SSE3 & AVX versions
- A clock tick: execute one instruction
- 2.2 GHz: 2.2 * 10^9 instructions per second



a 0 1 2 3

+

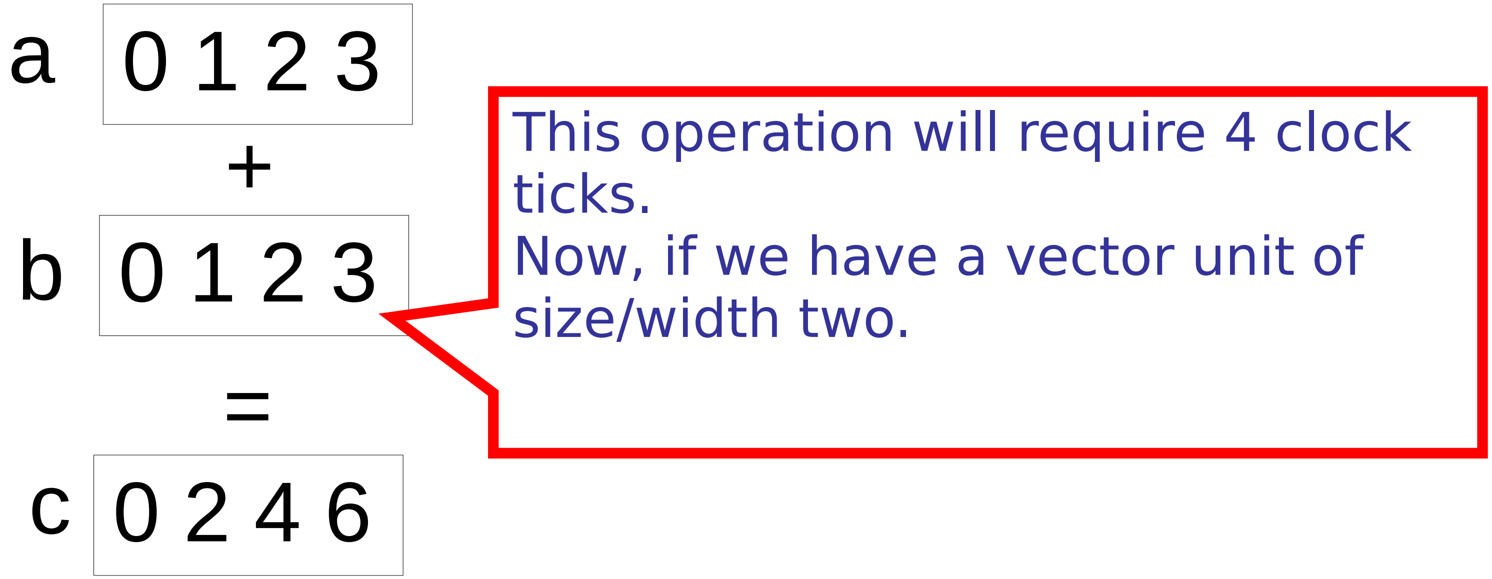
b 0 1 2 3

=

c 0 2 4 6

Vector Instructions

- `RAxML-NG` SSE3 & AVX versions
- A clock tick: execute one instruction
- 2.2 GHz: 2.2 * 10^9 instructions per second



a 0 1 2 3

+

b 0 1 2 3

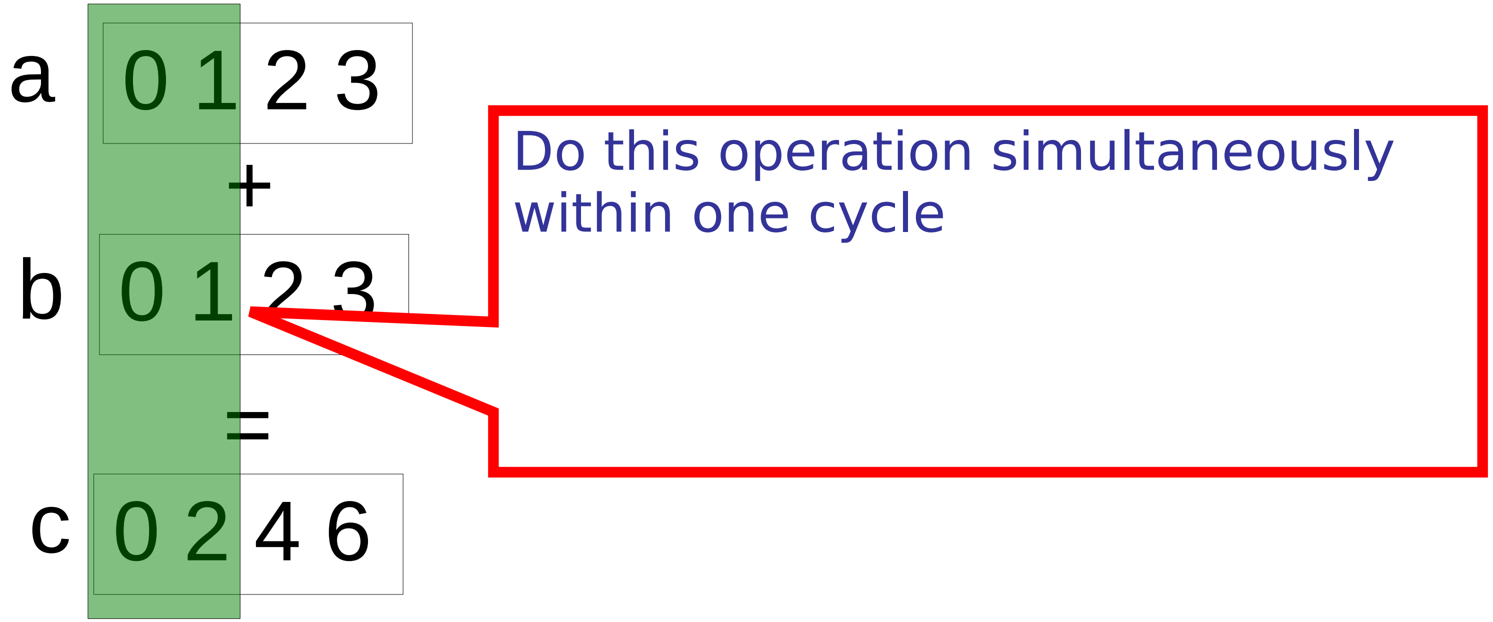
=

c 0 2 4 6

This operation will require 4 clock ticks.
Now, if we have a vector unit of size/width two.

Vector Instructions

- RAxML-NG SSE3 & AVX versions
- A clock tick: execute one instruction
- 2.2 GHz: 2.2 * 10^9 instructions per second



a 0 1 2 3

+

b 0 1 2 3

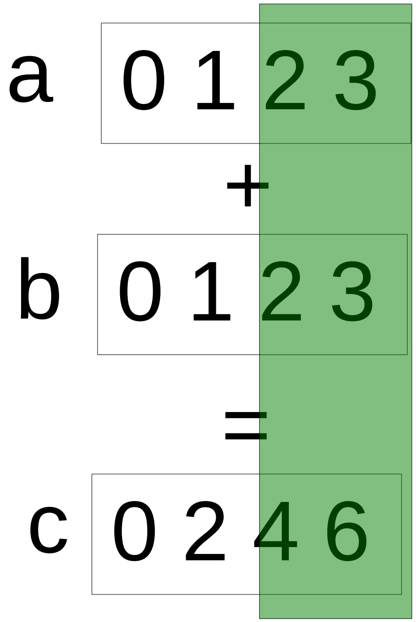
=

c 0 2 4 6

Do this operation simultaneously within one cycle

Vector Instructions

- `RAxML−NG` SSE3 & AVX versions
- A clock tick: execute one instruction
- 2.2 GHz: 2.2 * 10^9 instructions per second



a 0 1 2 3

+

b 0 1 2 3

=

c 0 2 4 6

... and this operation simultaneously within one cycle: only two clock cycles (ticks) required

Time permitting: Live demo

- We can also use vector instructions for parsimony calculations
- Check https://github.com/stamatak/Parsimonator-1.0.2

Vector Instructions

- Standard architectures (x86)
  - vector widths of 128 or 256 bits
  - As of 2017: 512 bit instructions on Intel CPUs
- GPUs: at least one order of magnitude larger vectors
- Vector instructions are synchronized automatically by the processor clock → no synchronization overhead :-)
- Always use **vectorized** versions of programs !

Horizontal Add

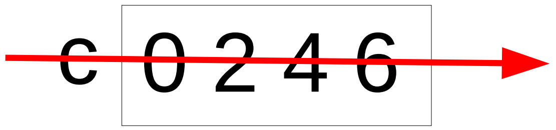
- Sometimes we need to sum over the values in a vector horizontally – we call this a horizontal add

  → different round off error propagation depending on vector width

Horizontal Add

- Sometimes we need to sum over the values in a vector horizontally – we call this a horizontal add

→ different round off error propagation depending on vector width

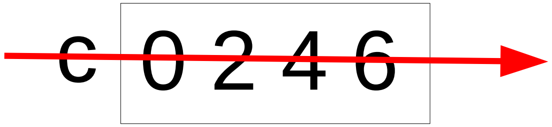


Sum over this values in the vector
→ could be per-site log likelihoods

Horizontal Add

- Sometimes we need to sum over the values in a vector horizontally – we call this a horizontal add

→ different round off error propagation depending on vector width



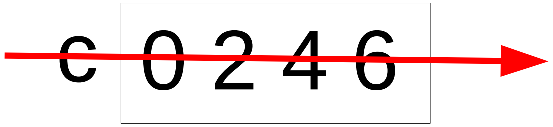
c 0 2 4 6

= (0 + 2) + (4 + 6)

Horizontal Add

- Sometimes we need to sum over the values in a vector horizontally – we call this a horizontal add

→ different round off error propagation depending on vector width

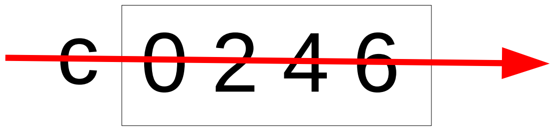


= (0 + 4) + (2 + 6)

Horizontal Add

- Sometimes we need to sum over the values in a vector horizontally – we call this a horizontal add

  → different round off error propagation depending on vector width



= 0 + (4 + (2 + 6))

Reproducibilty

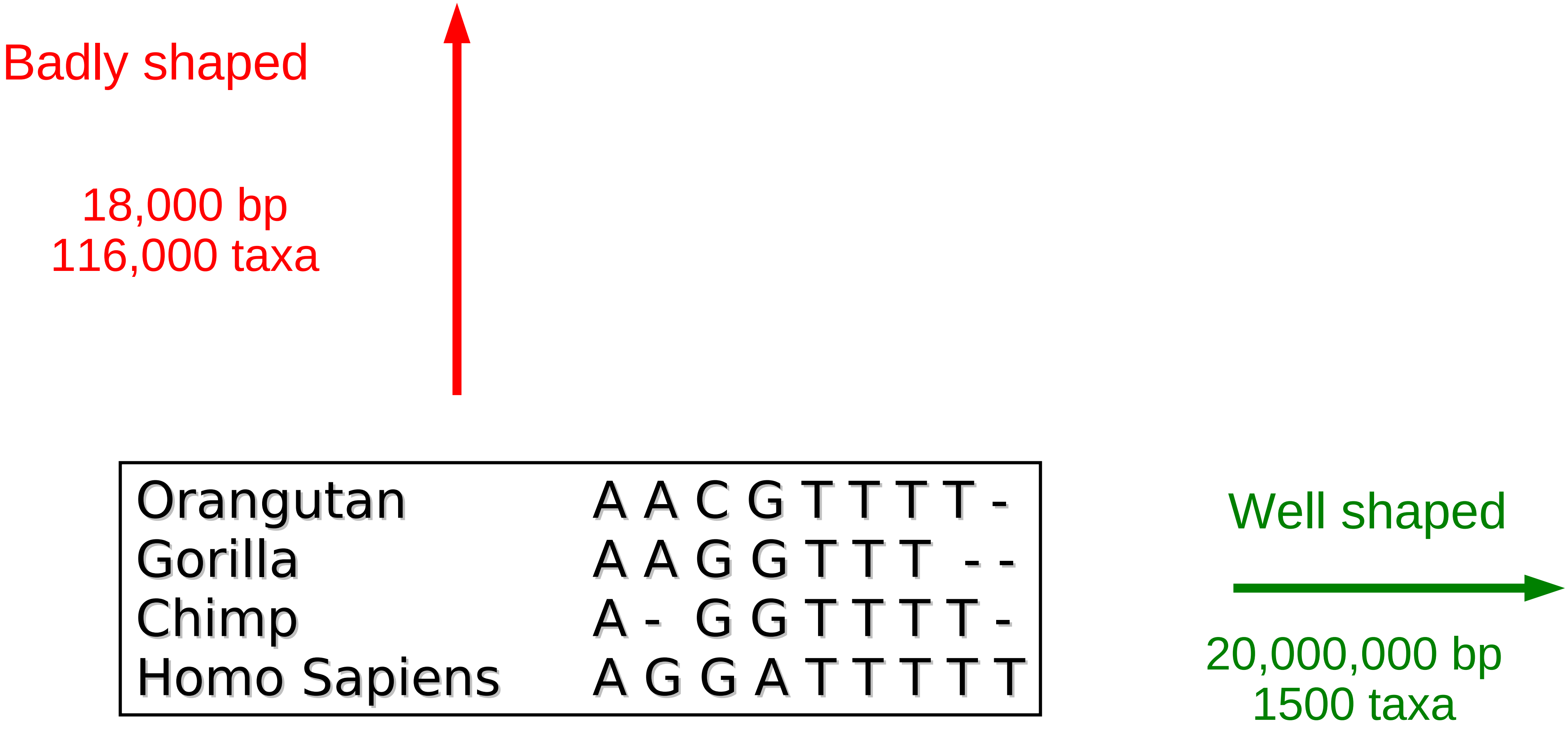
- Under floating point

  $(a + b) + c \neq a + (b + c)$

  → Order will affect the result – distinct round off error propagation
- Sequential execution:
  - Tree inference might yield different trees if you use different compiler
  - Tree inference might yield different trees if you use `SSE3` (128 bits) or `AVX` (256 bits)
- We have observed this on real data !
- If the dataset is difficult this is more likely to happen!

  → difficulty prediction with `Pythia`

Dataset Shapes



Badly shaped

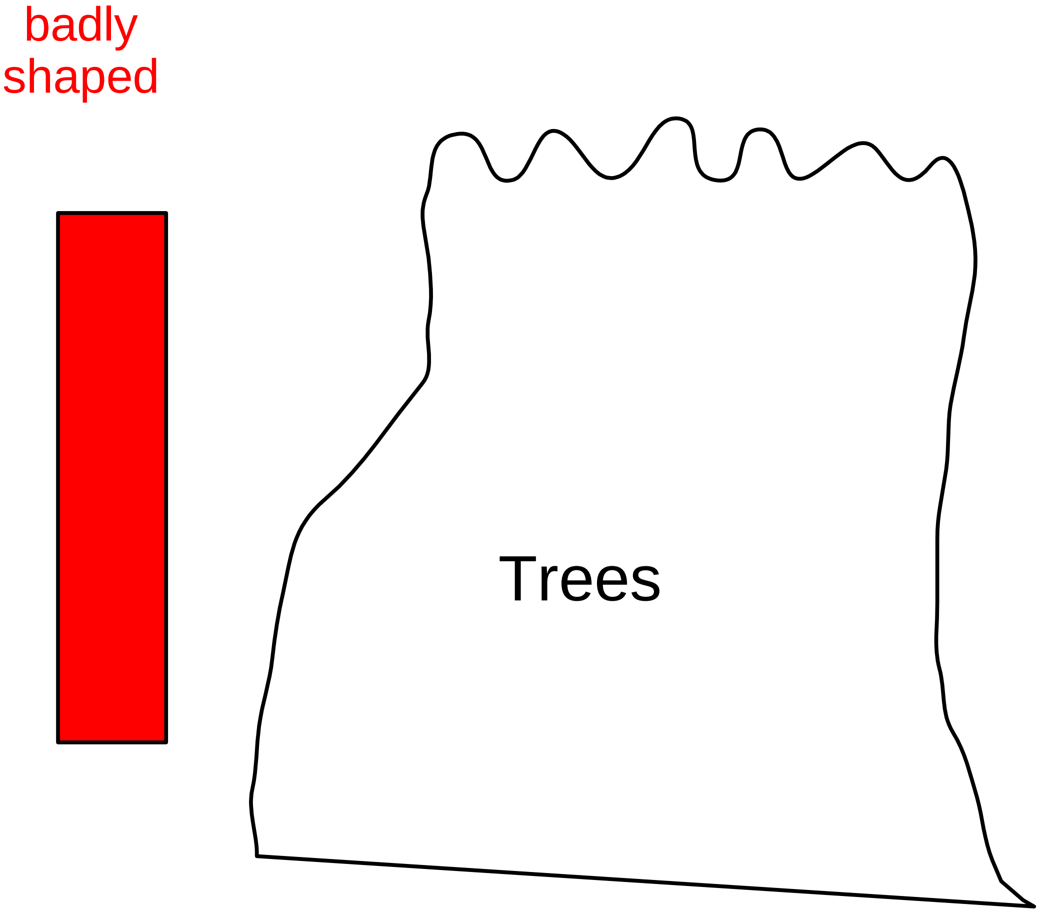
18,000 bp
116,000 taxa

Orangutan	A A C G T T T T -
Gorilla	A A G G T T T - -
Chimp	A - G G T T T T -
Homo Sapiens	A G G A T T T T T

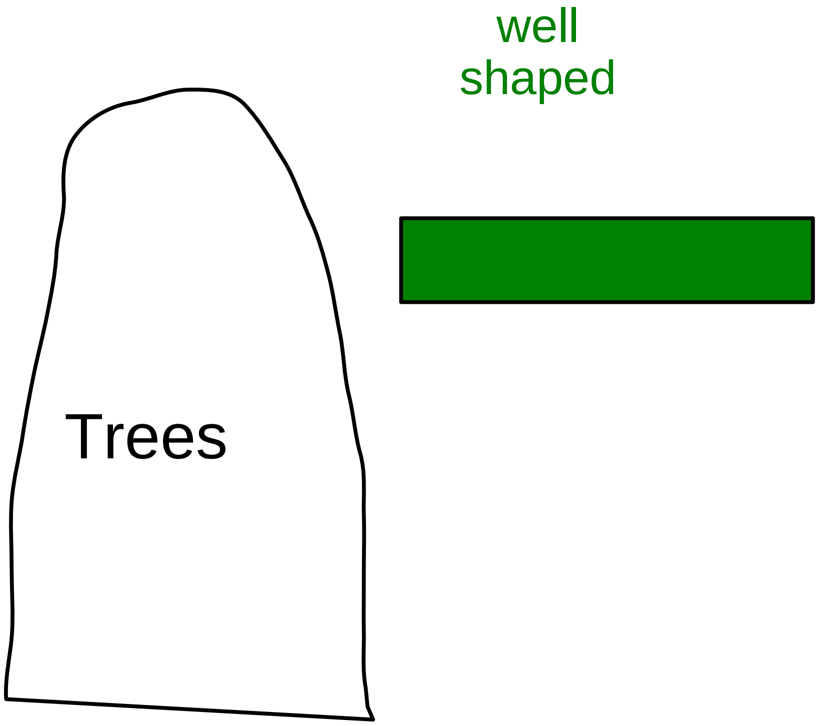
Well shaped

20,000,000 bp
1500 taxa

Easy & Difficult Likelihood Surfaces

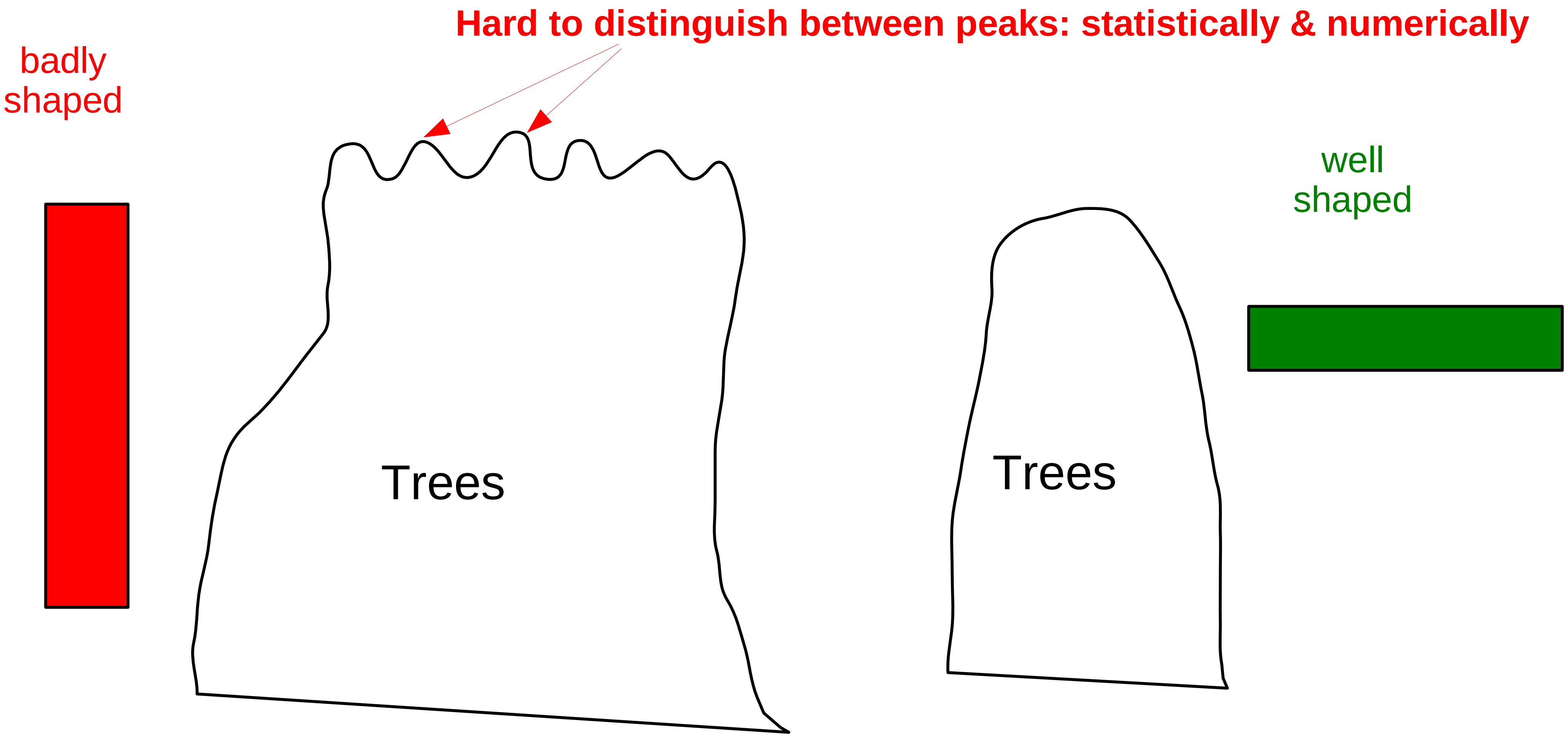


Rough likelihood surface:
many taxa, few bp



Smooth likelihood surface:
Few taxa, many bp

Easy & Difficult Likelihood Surfaces



Hard to distinguish between peaks: statistically & numerically

badly
shaped

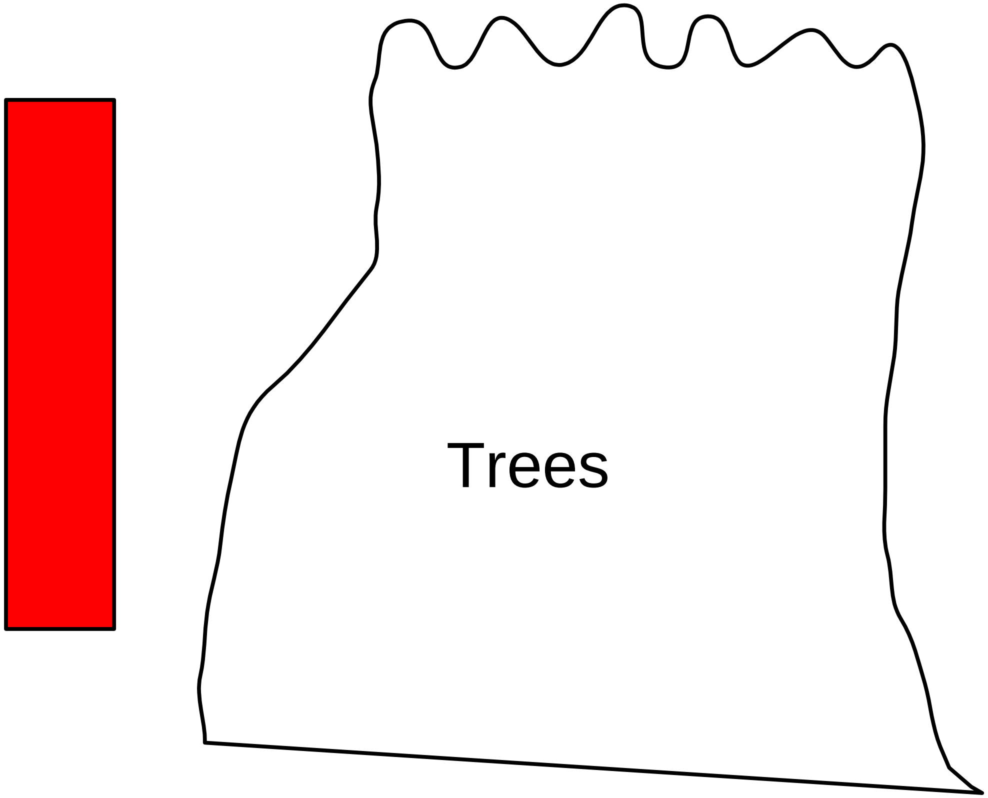
well
shaped

Rough likelihood surface:
many taxa, few bp

Smooth likelihood surface:
Few taxa, many bp

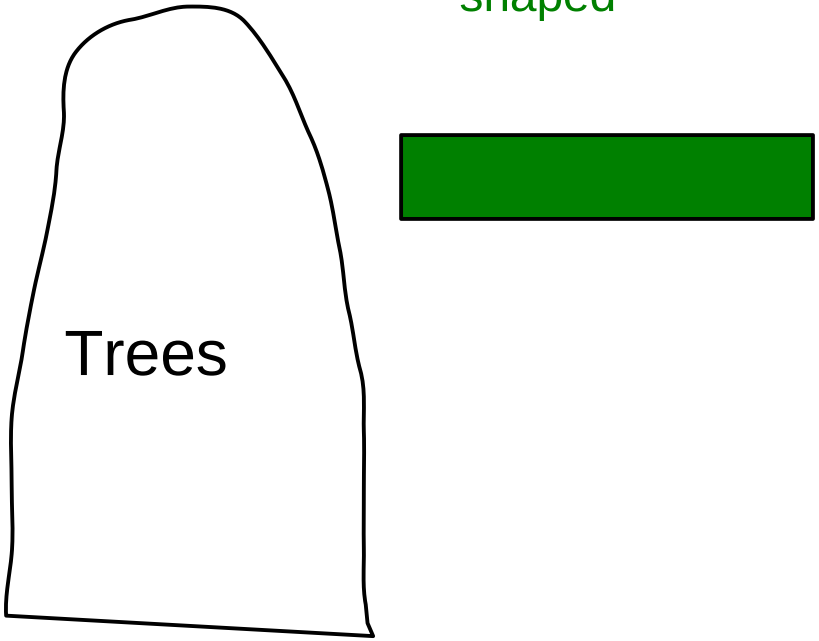
Easy & Difficult Likelihood Surfaces

badly shaped



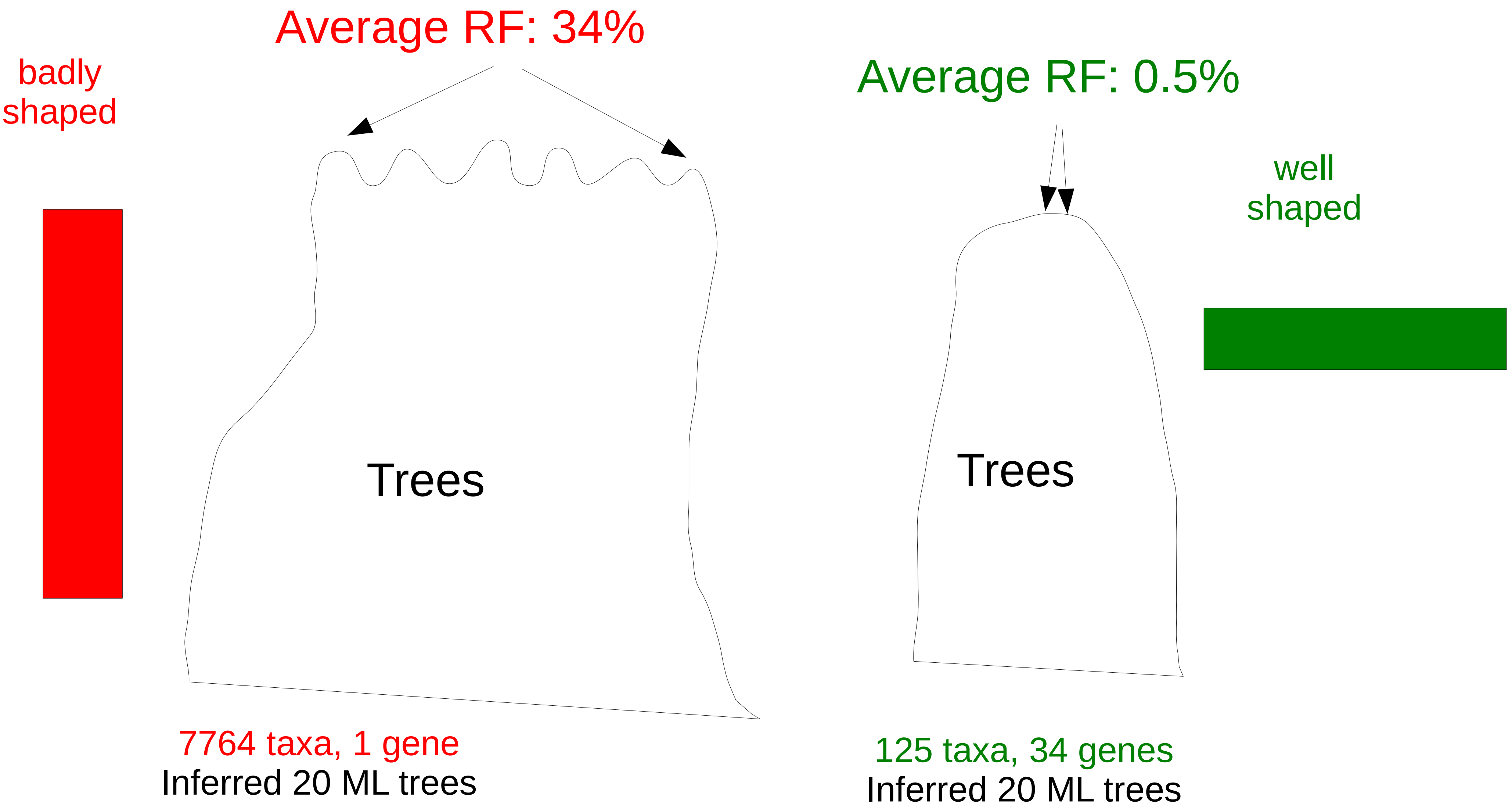
7764 taxa, 1 gene
Inferred 20 ML trees

well shaped



125 taxa, 34 genes
Inferred 20 ML trees

Easy & Difficult Likelihood Surfaces



Average RF: 34%

badly shaped

Trees

7764 taxa, 1 gene
Inferred 20 ML trees

Average RF: 0.5%

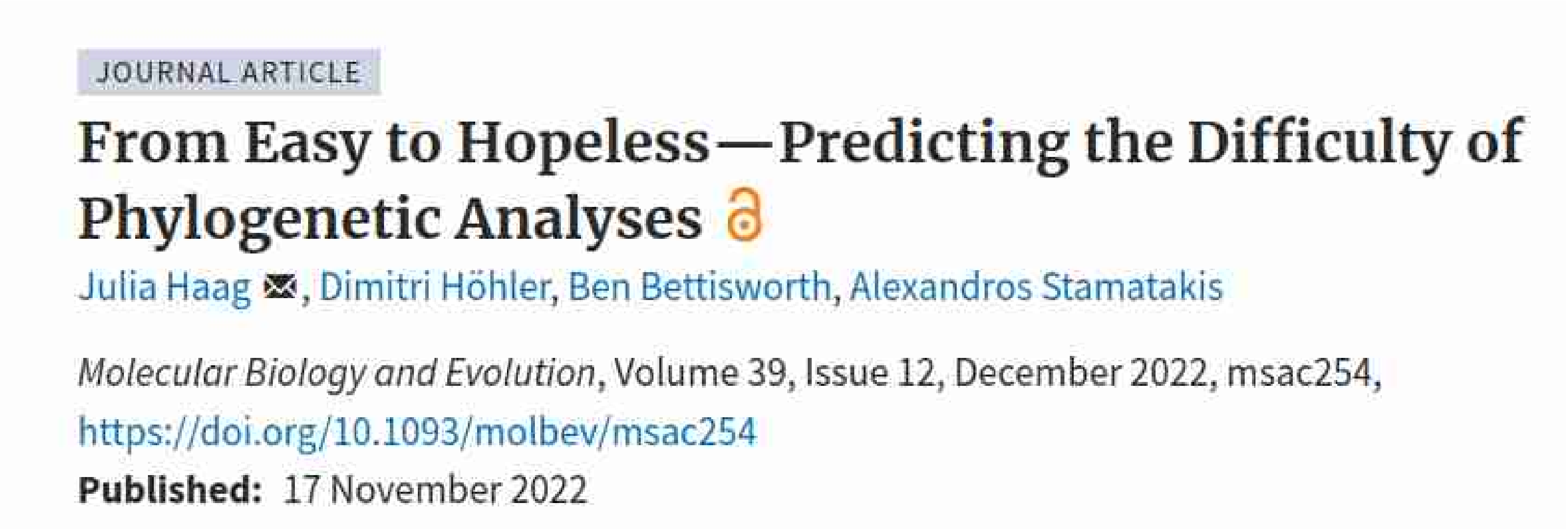
well shaped

Trees

125 taxa, 34 genes
Inferred 20 ML trees

Now we can quantify this

- In past years these slides about easy and hard datasets were very hand-wavy
- Since 2022 we can quantify & predict difficulty



JOURNAL ARTICLE

From Easy to Hopeless—Predicting the Difficulty of Phylogenetic Analyses

Julia Haag, Dimitri Höhler, Ben Bettisworth, Alexandros Stamatakis

Molecular Biology and Evolution, Volume 39, Issue 12, December 2022, msac254, https://doi.org/10.1093/molbev/msac254

Published: 17 November 2022

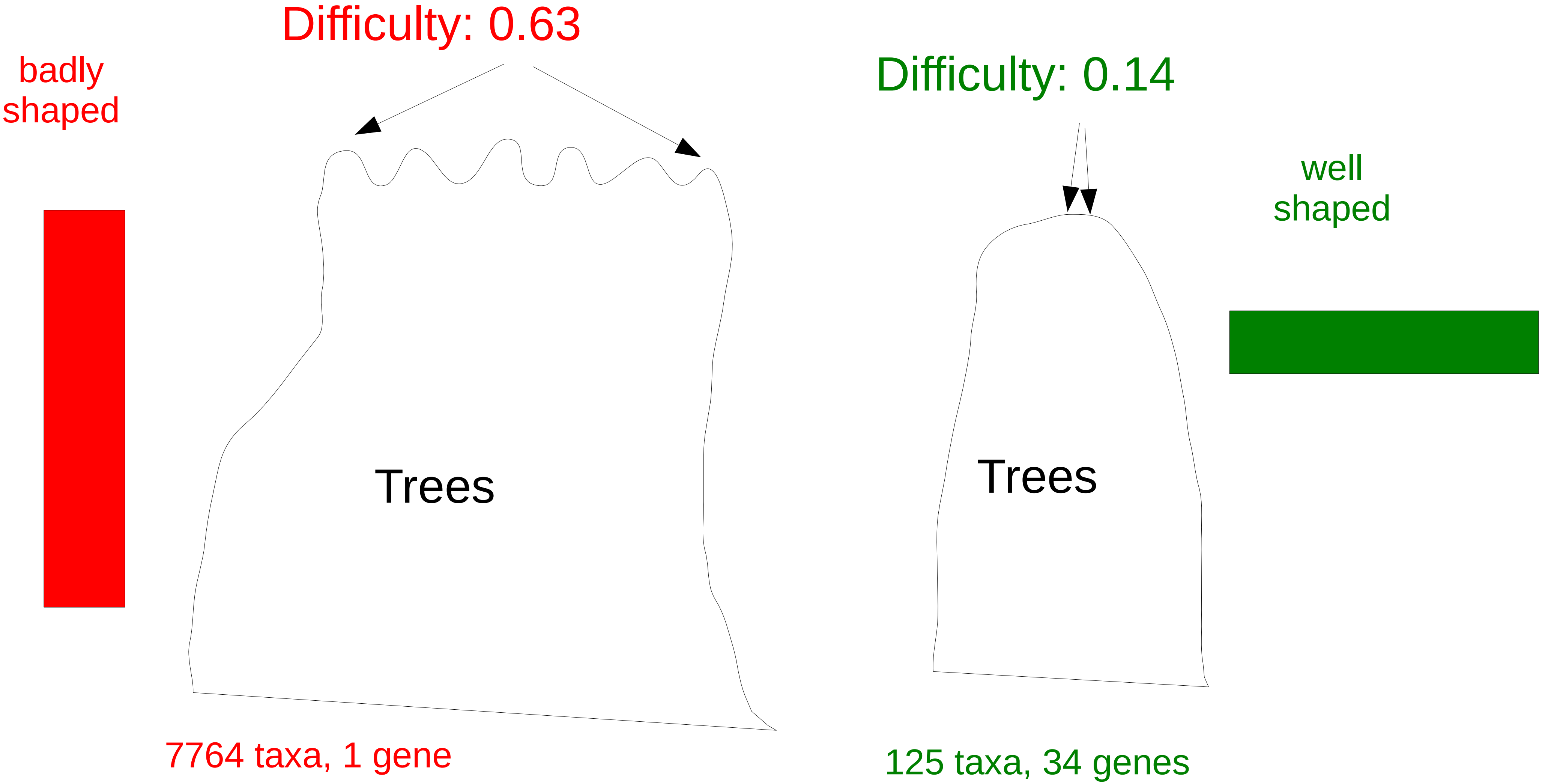
Predicting Dataset Difficulty

- `Pythia` tool to predict difficulty of phylogenetic analysis
- **Input:** MSA
- **Output:** a difficulty value ranging between 0.0 (easy) to 1.0 (hopeless)
- Invocations for our example datasets:

```
pythia --msa 125.phy --raxmlng ~/bin/raxml-ng
pythia --msa 7764.phy --raxmlng ~/bin/raxml-ng
```

- There seems to be a good correlation between the difficulty score and the average bootstrap support values
- Also, “apparent convergence” speed of MCMC analyses can potentially be predicted
- A small SARS-CoV-2 dataset we analyzed 2 years ago has a difficulty score of *0.84*

Easy & Difficult Likelihood Surfaces



Difficulty: 0.63

badly
shaped

Trees

7764 taxa, 1 gene

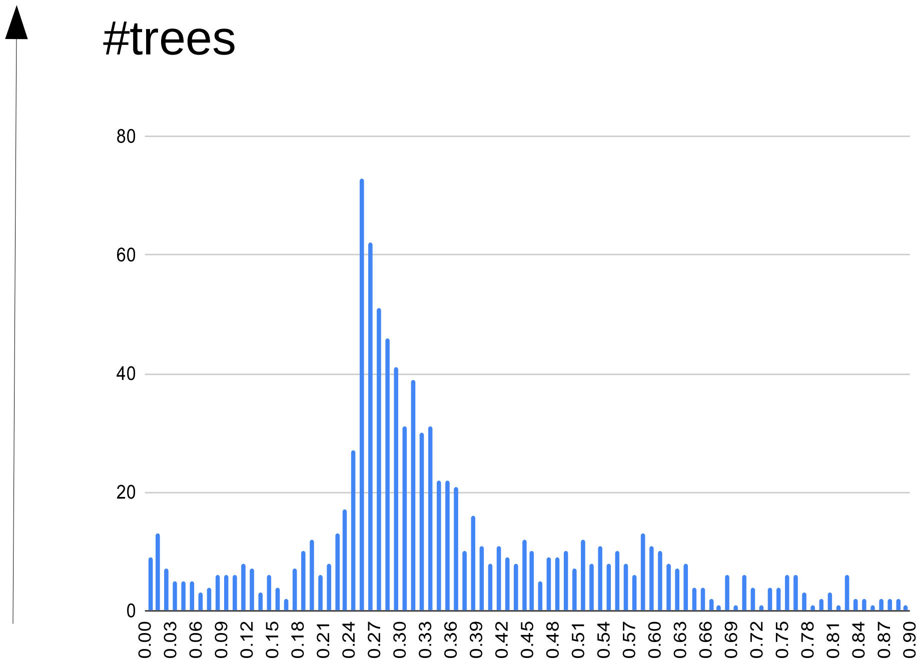
Difficulty: 0.14

well
shaped

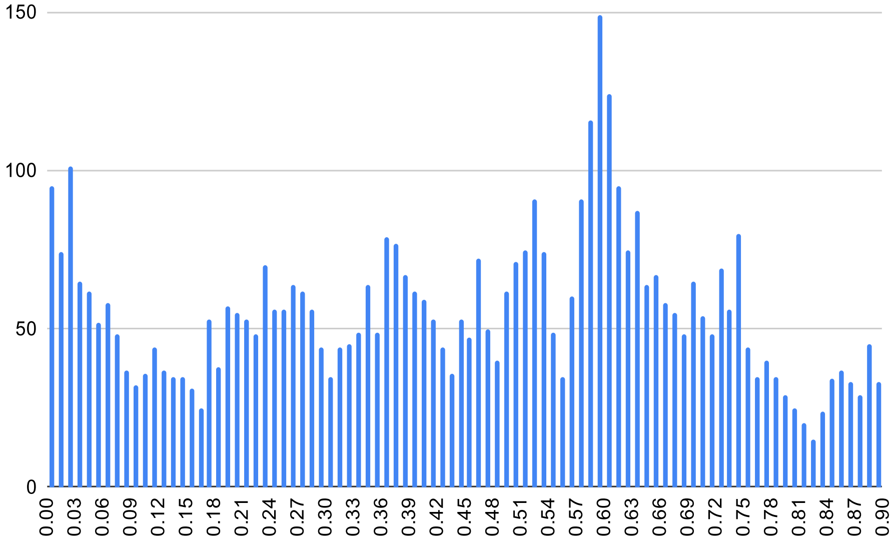
Trees

125 taxa, 34 genes

Difficulty Distributions



RAxML-Grove Database

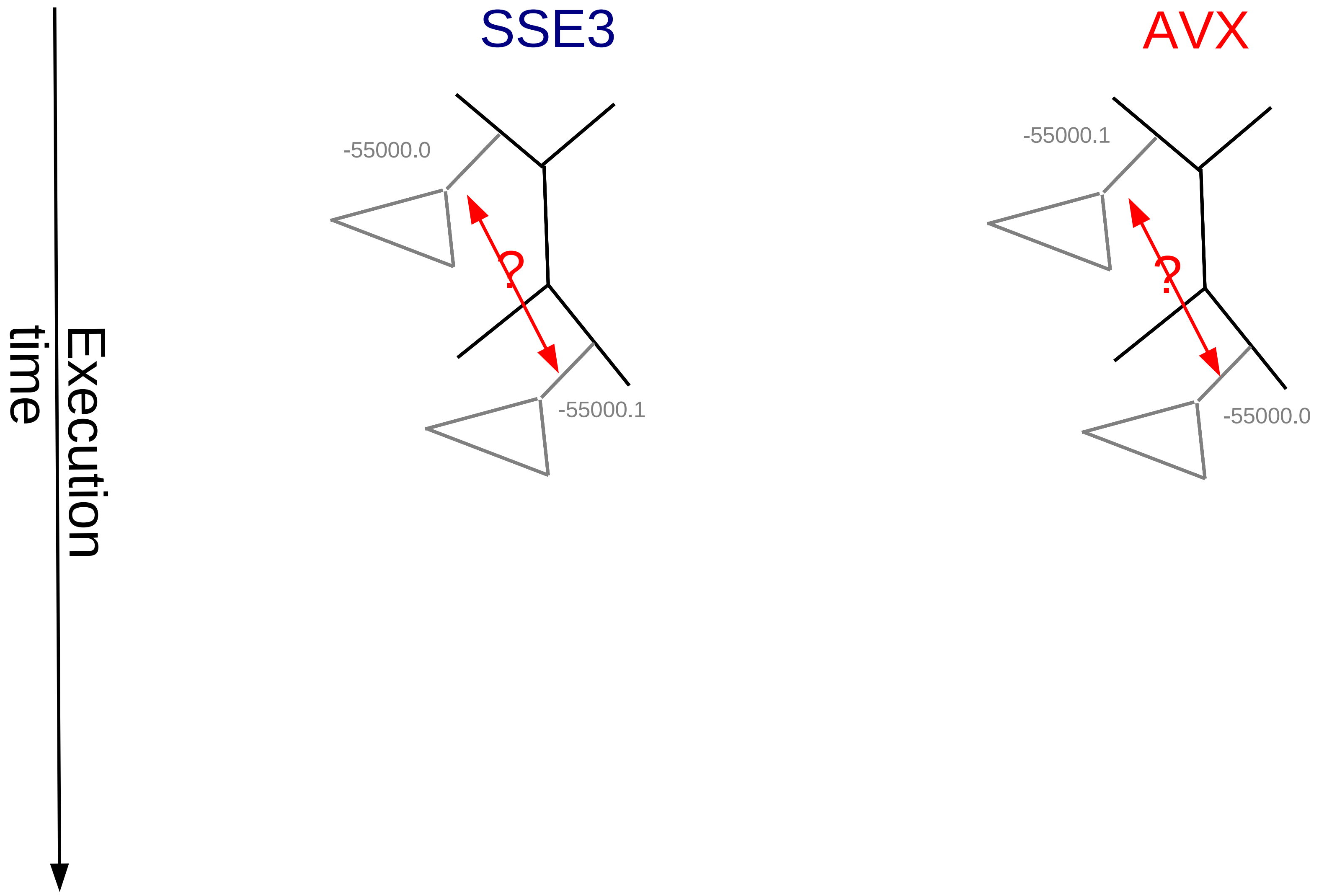


TreeBase Database

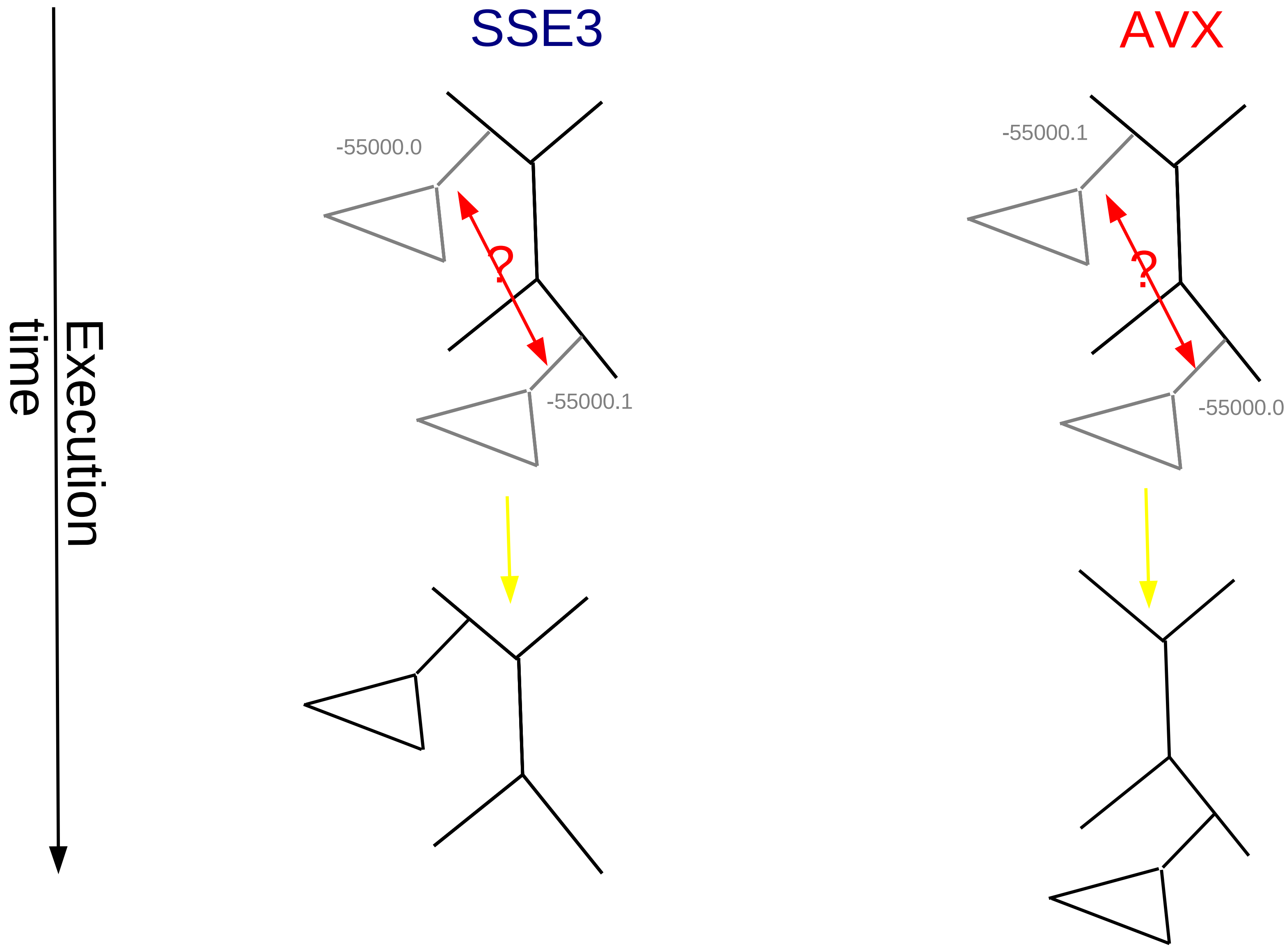
Easy ↔ Hopeless

Easy ↔ Hopeless

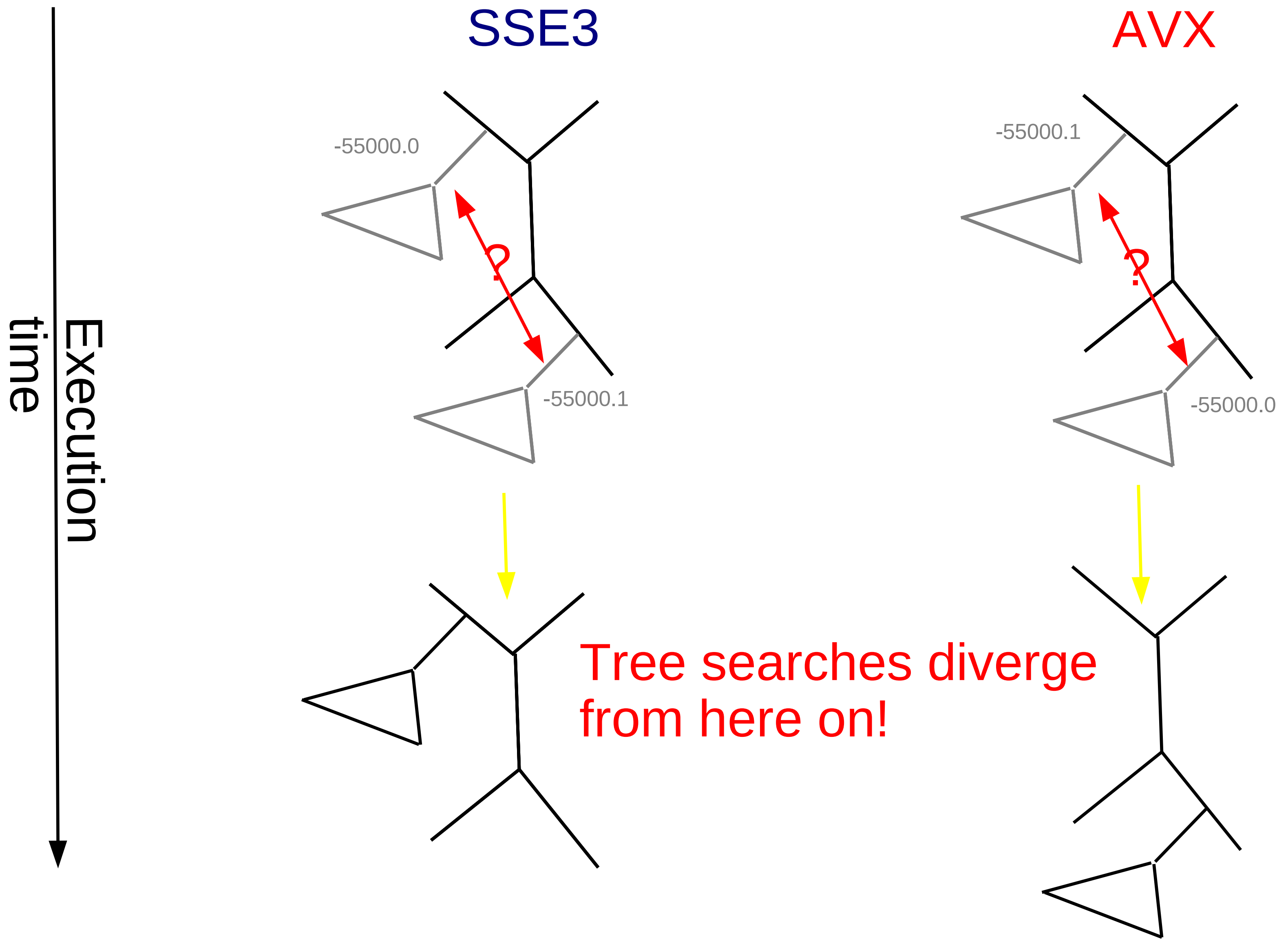
Why does difficulty matter for reproducibility?



Why does difficulty matter for reproducibility?



Why does difficulty matter for reproducibility?



SSE3

AVX

Tree searches diverge from here on!

Outline

- The root of all evil
- Sequential Computations
- **Parallel Computations**
- Software Quality

Reproducibilty

- Under floating point

  $(a + b) + c \neq a + (b + c)$

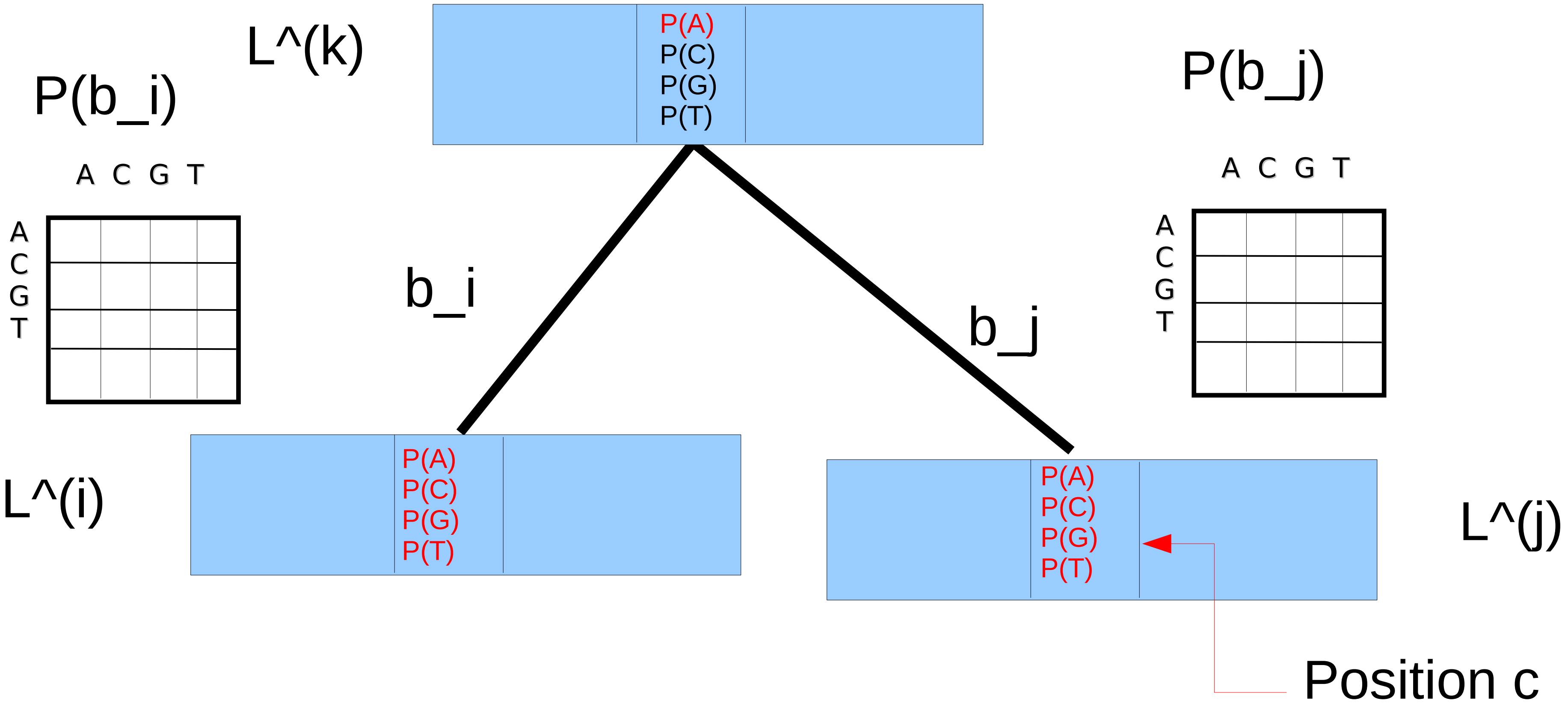
  → Order will affect the result – distinct round off error propagation

- Parallel execution: tree inference might yield different trees if you use 2 or 4 cores for parallel likelihood calculations

  We have observed this on real data !

Felsenstein pruning (again)

$$\vec{L}_A^{(k)}(c) = (\sum_{S=A}^{T} P_{AS}(b_i)\vec{L}_S^{(i)}(c))(\sum_{S=A}^{T} P_{AS}(b_j)\vec{L}_S^{(j)}(c))$$



L^(k)

P(A)
P(C)
P(G)
P(T)

P(b_i)

A C G T

A
C
G
T

P(b_j)

A C G T

A
C
G
T

b_i

b_j

L^(i)

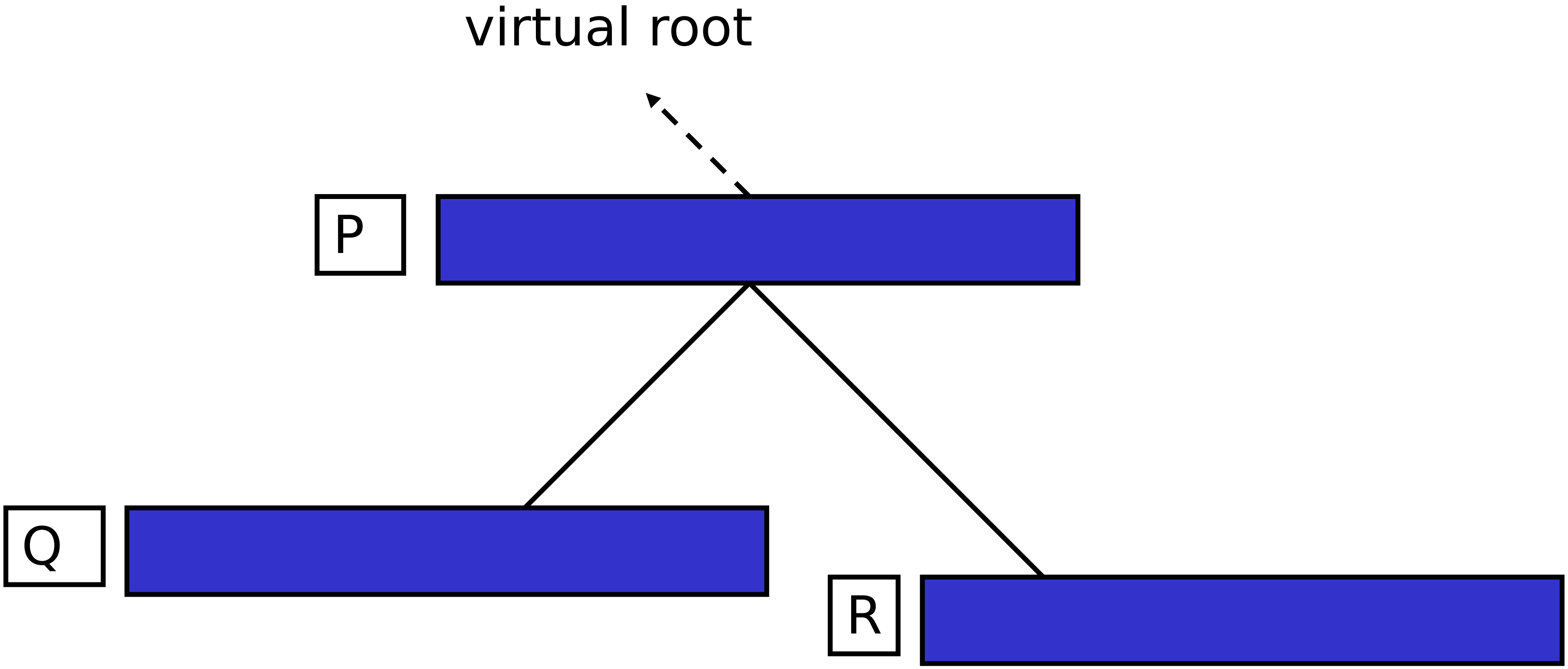
P(A)
P(C)
P(G)
P(T)

L^(j)

P(A)
P(C)
P(G)
P(T)

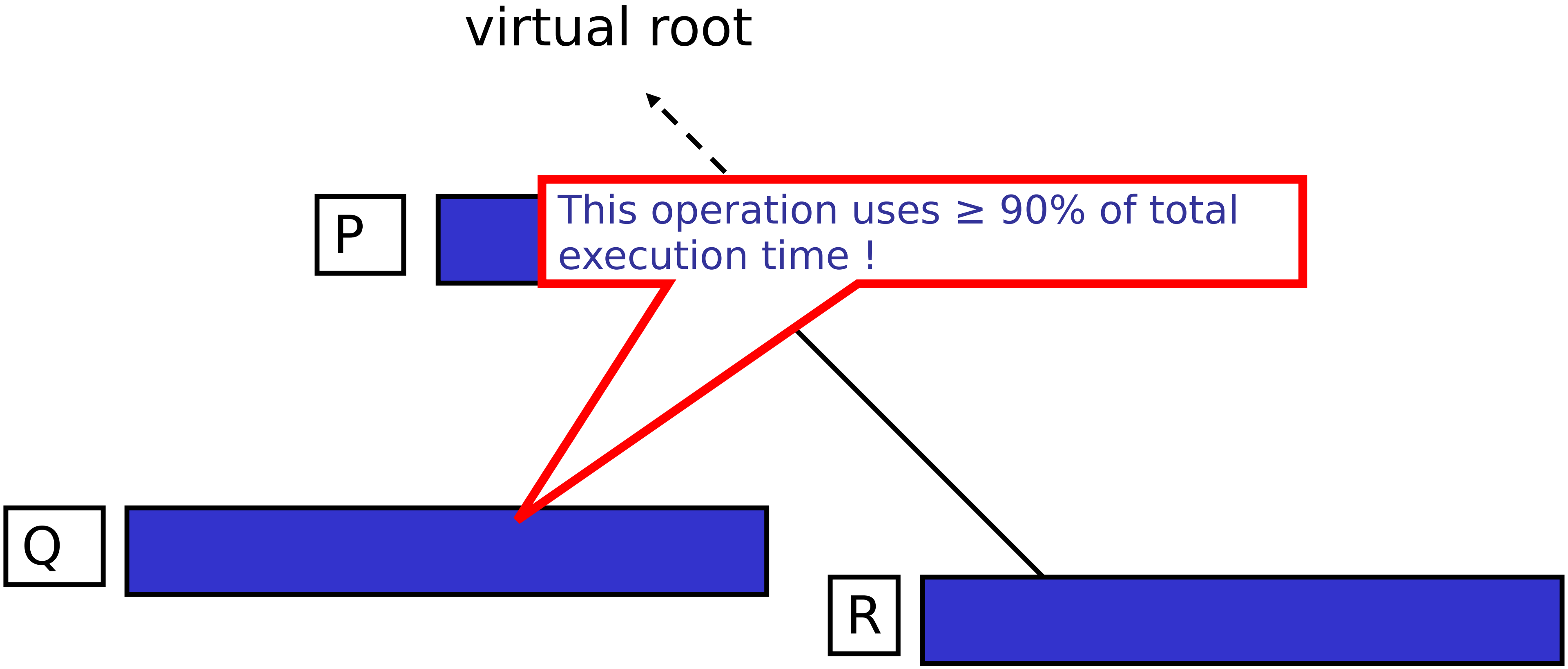
Position c

Loop Level Parallelism



P[i] = f(Q[i], R[i])

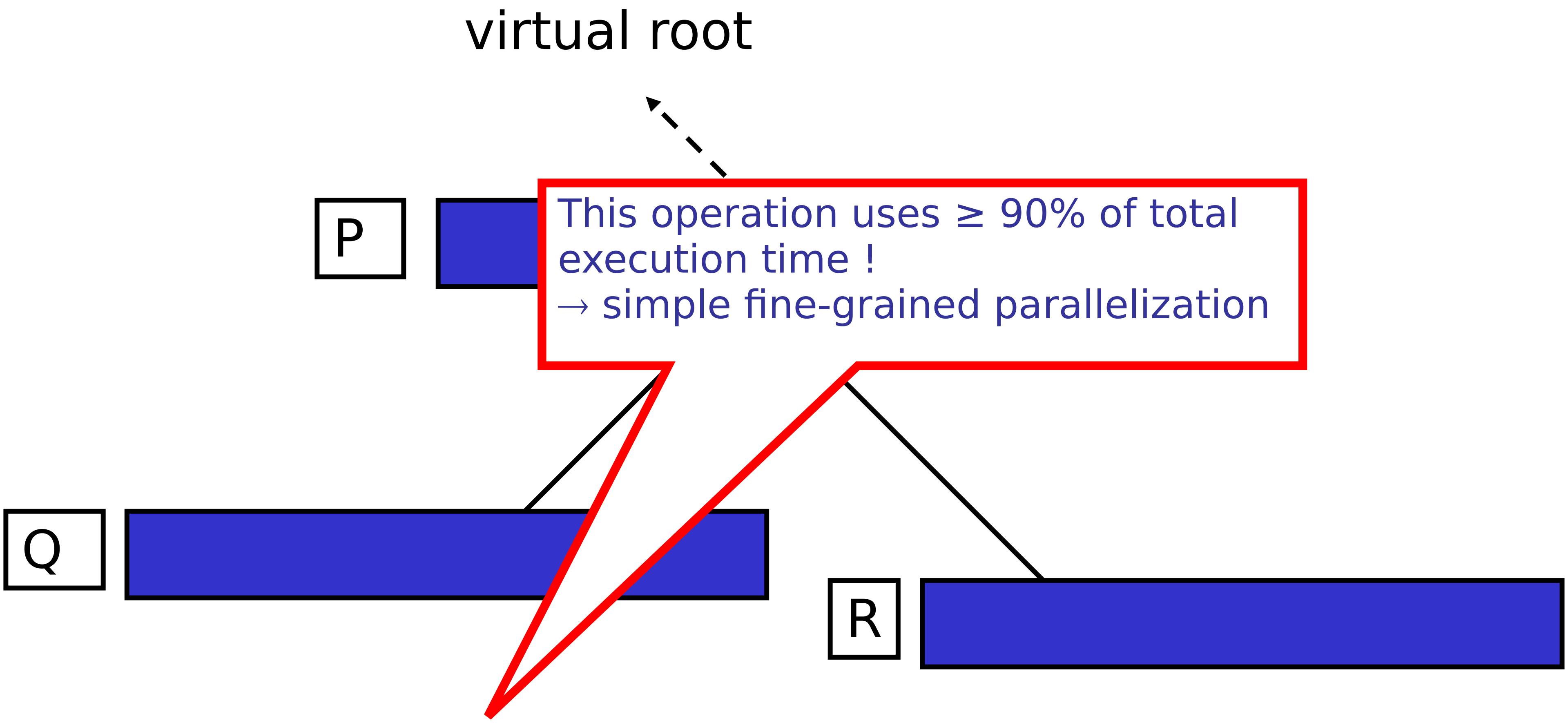
Loop Level Parallelism



This operation uses ≥ 90% of total execution time !

P[i] = f(Q[i], R[i])

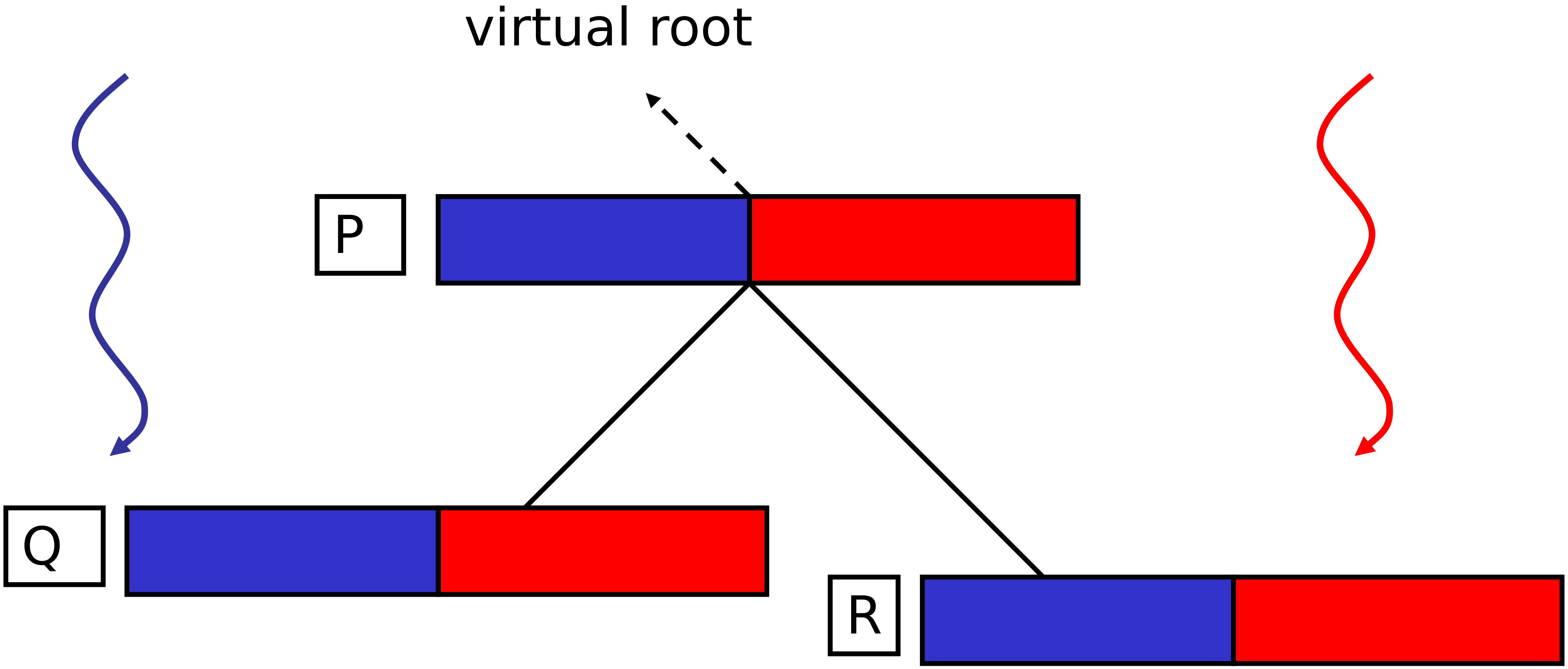
Loop Level Parallelism



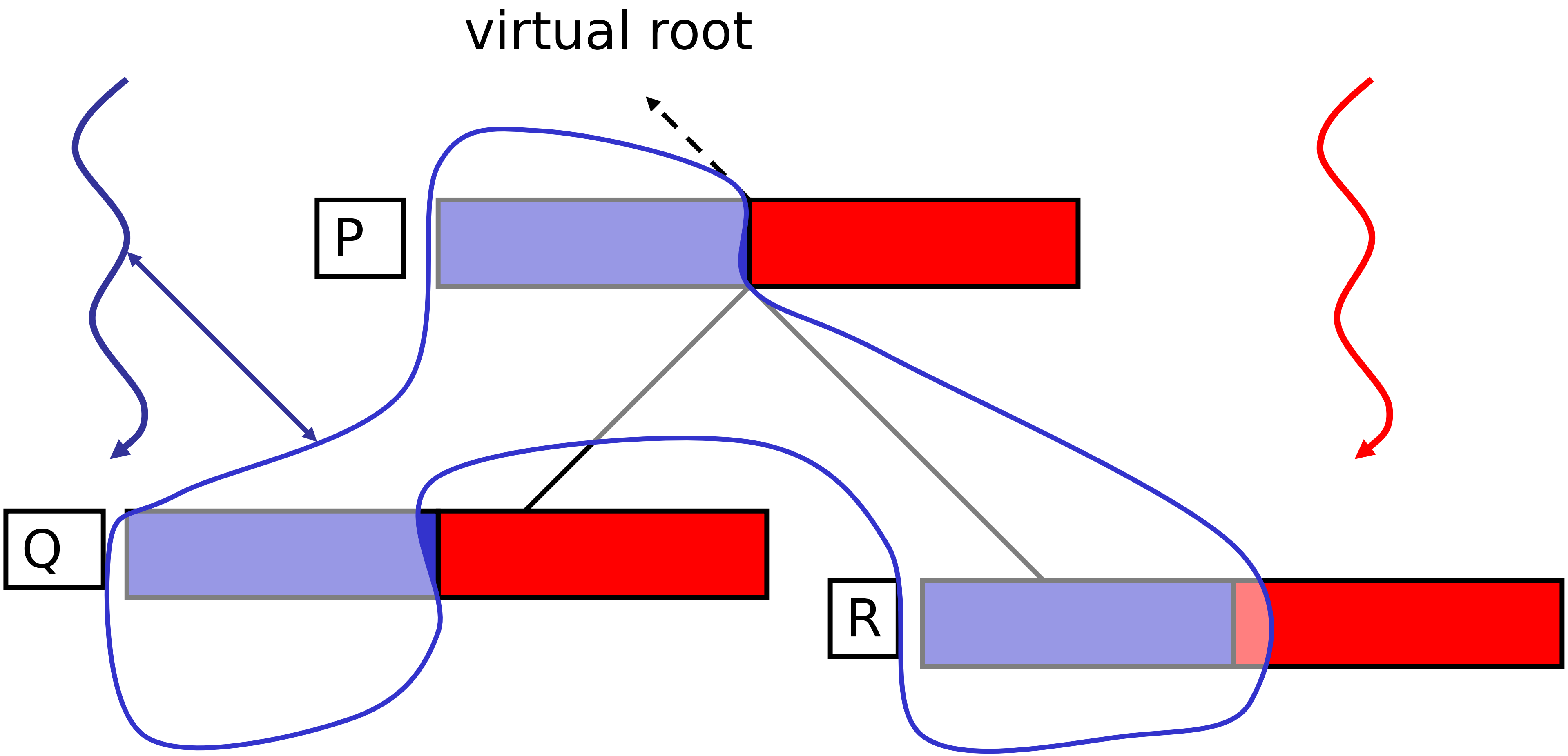
This operation uses ≥ 90% of total execution time !
→ simple fine-grained parallelization

P[i] = f(Q[i], R[i])

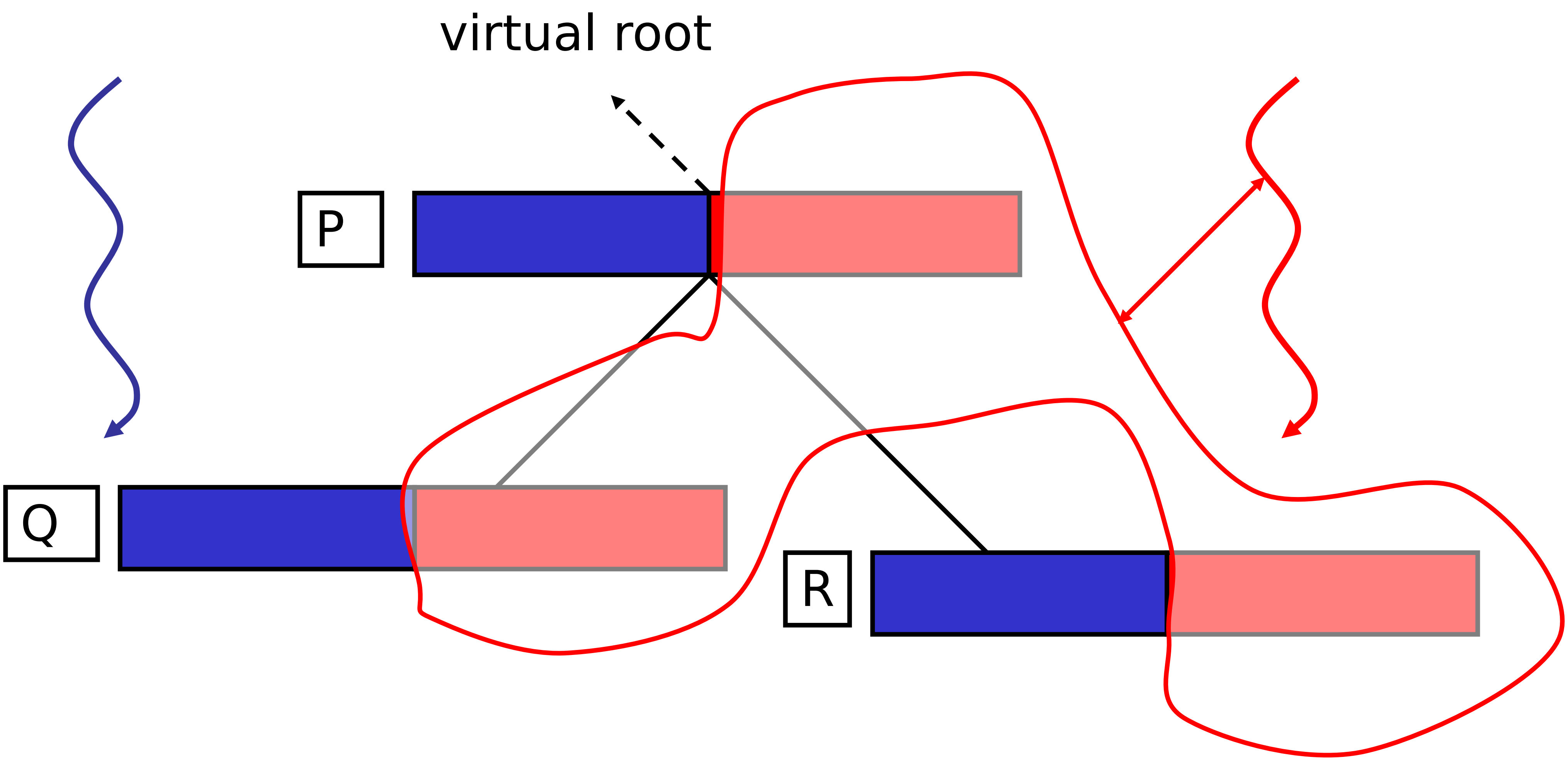
Loop Level Parallelism



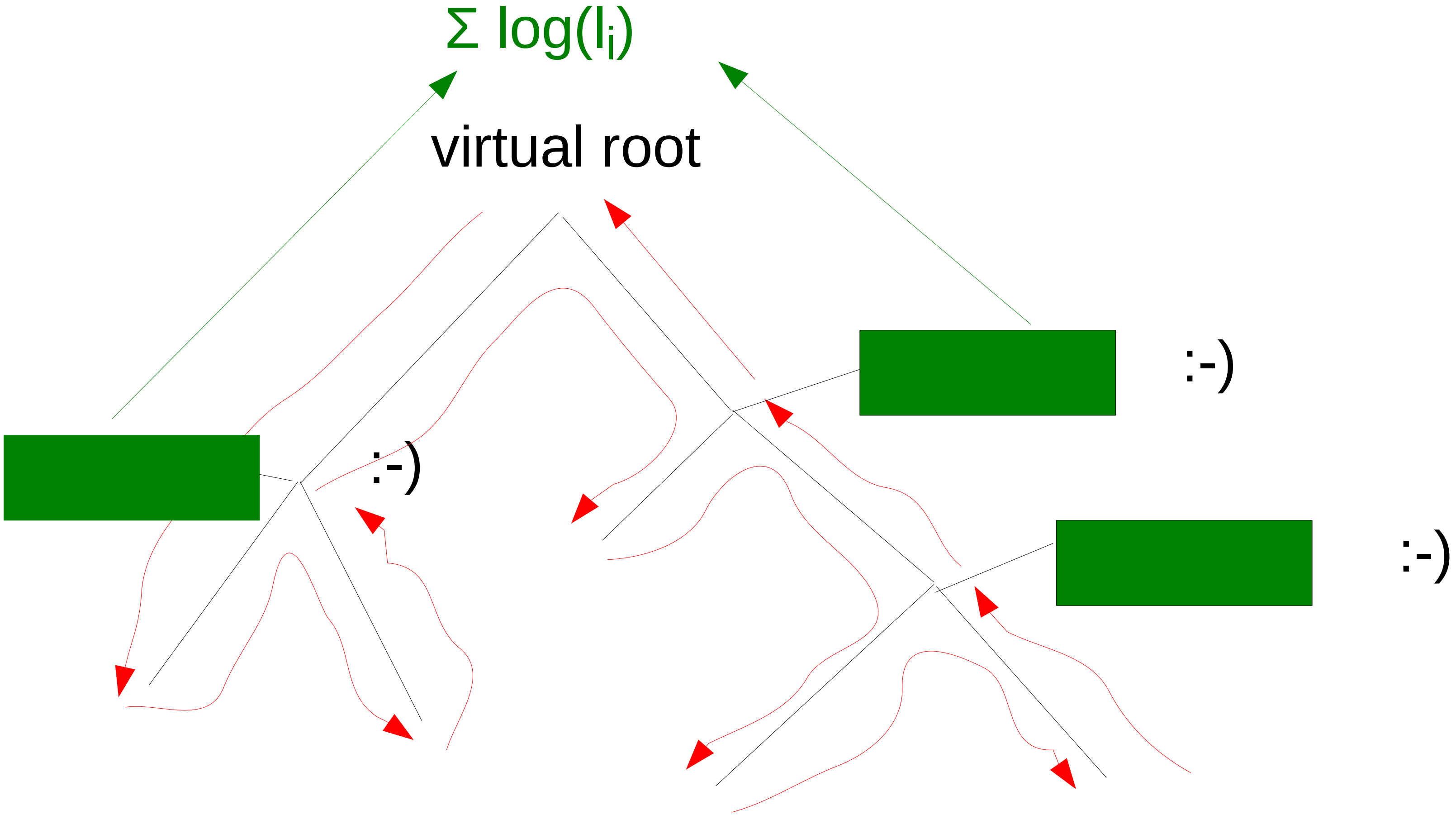
Loop Level Parallelism



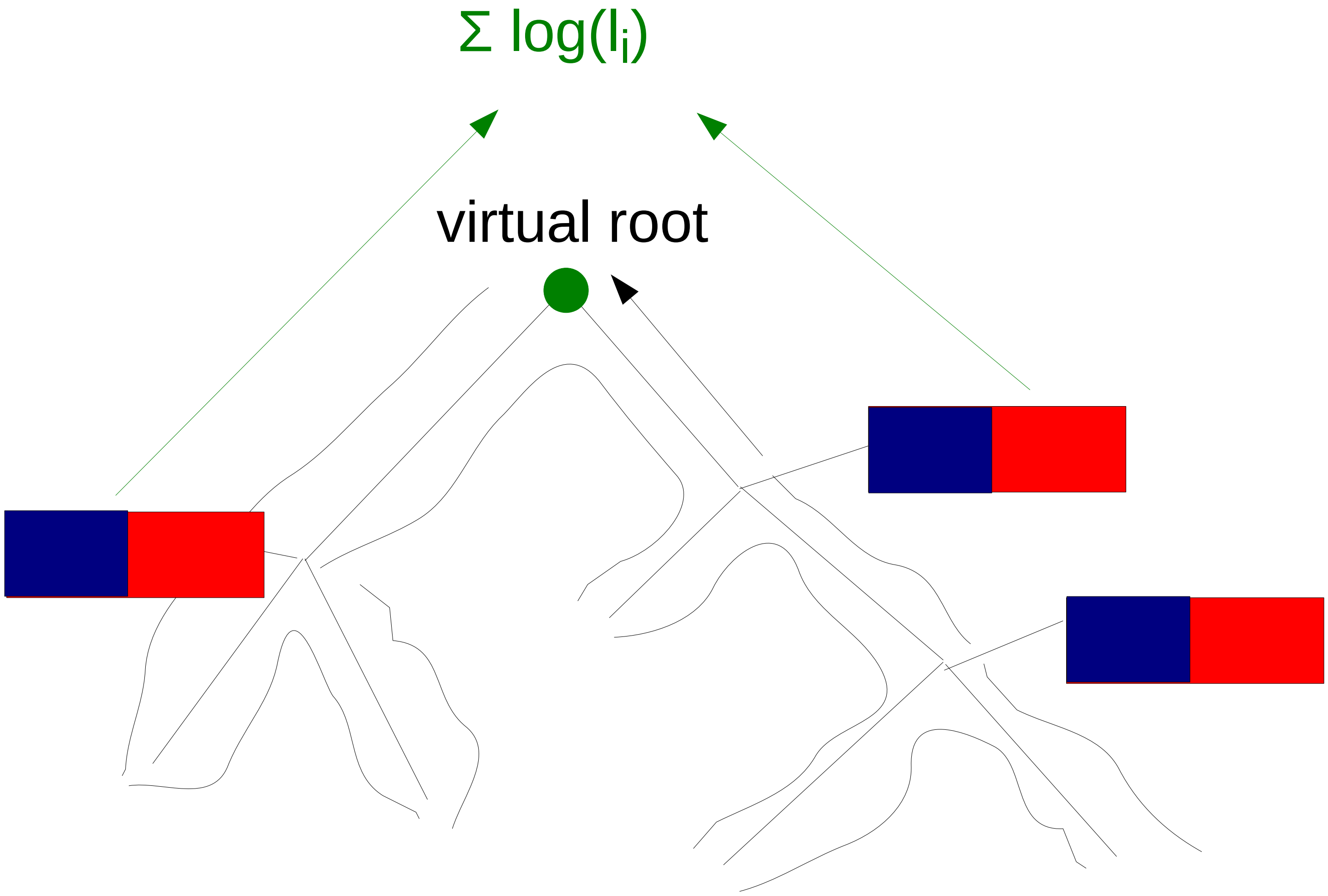
Loop Level Parallelism



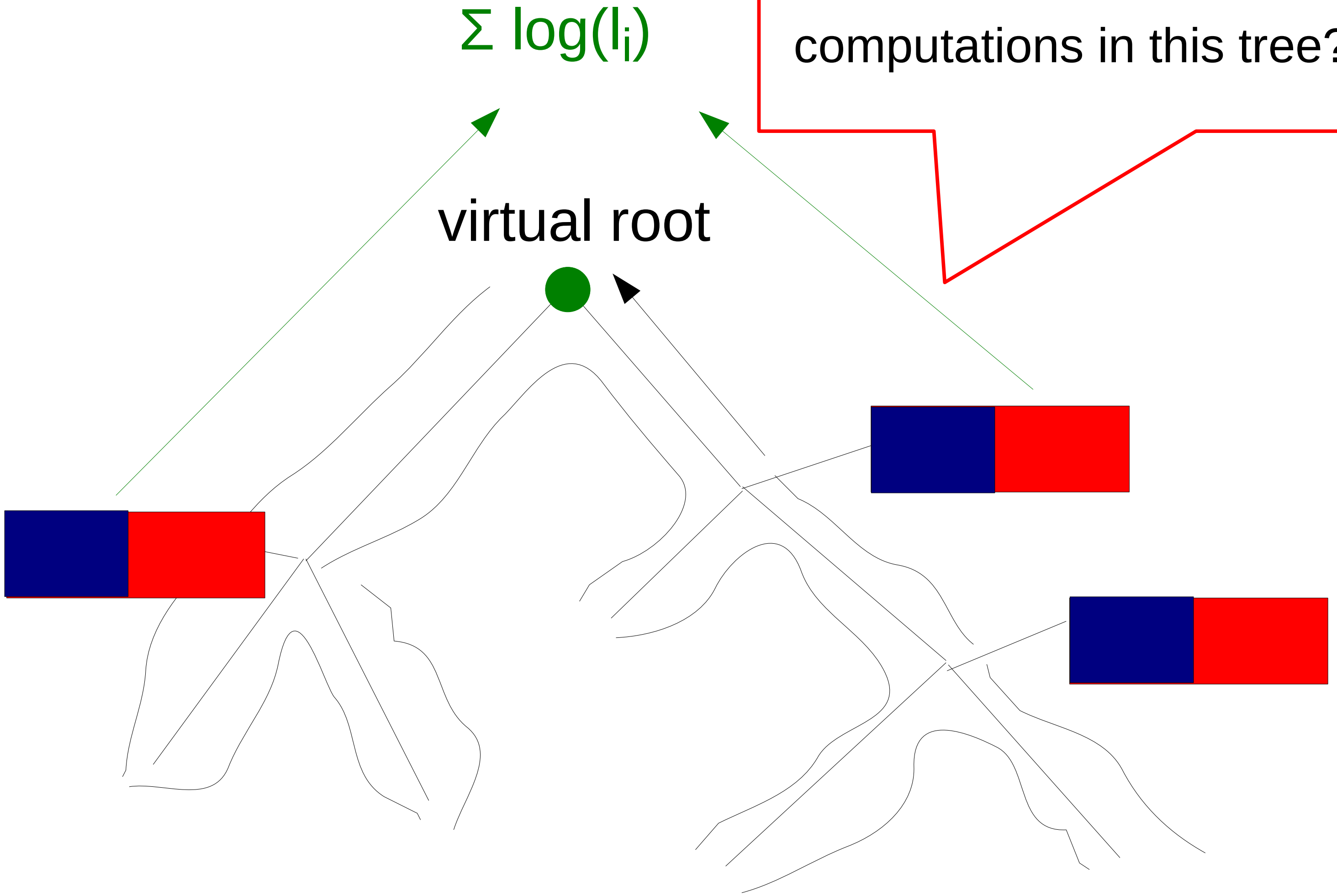
Post-order Traversal



Parallel Post-order Traversal

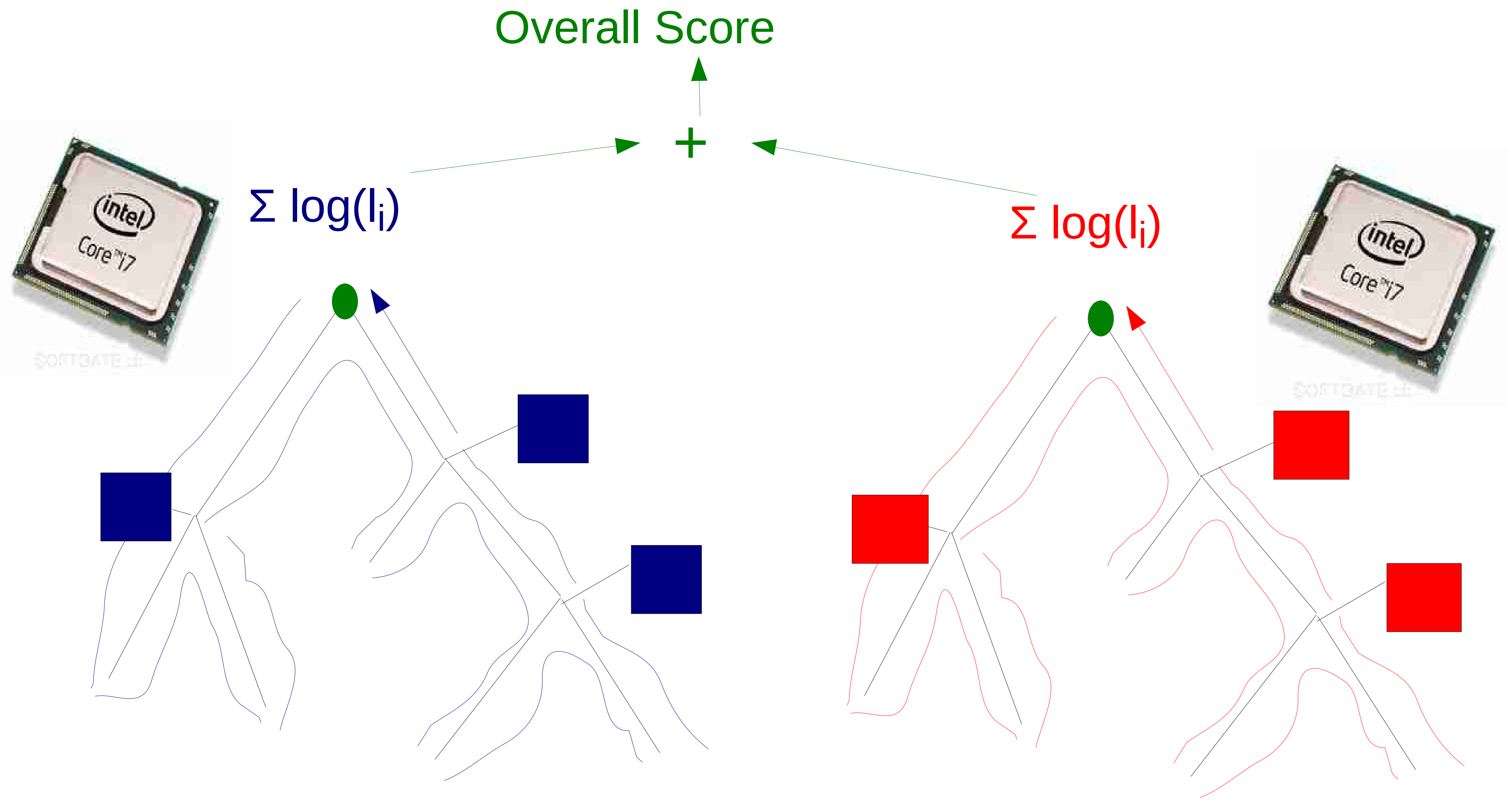


Parallel Post-order Traversal



How many times do we need to synchronize computations in this tree?

Parallel Post-order Traversal



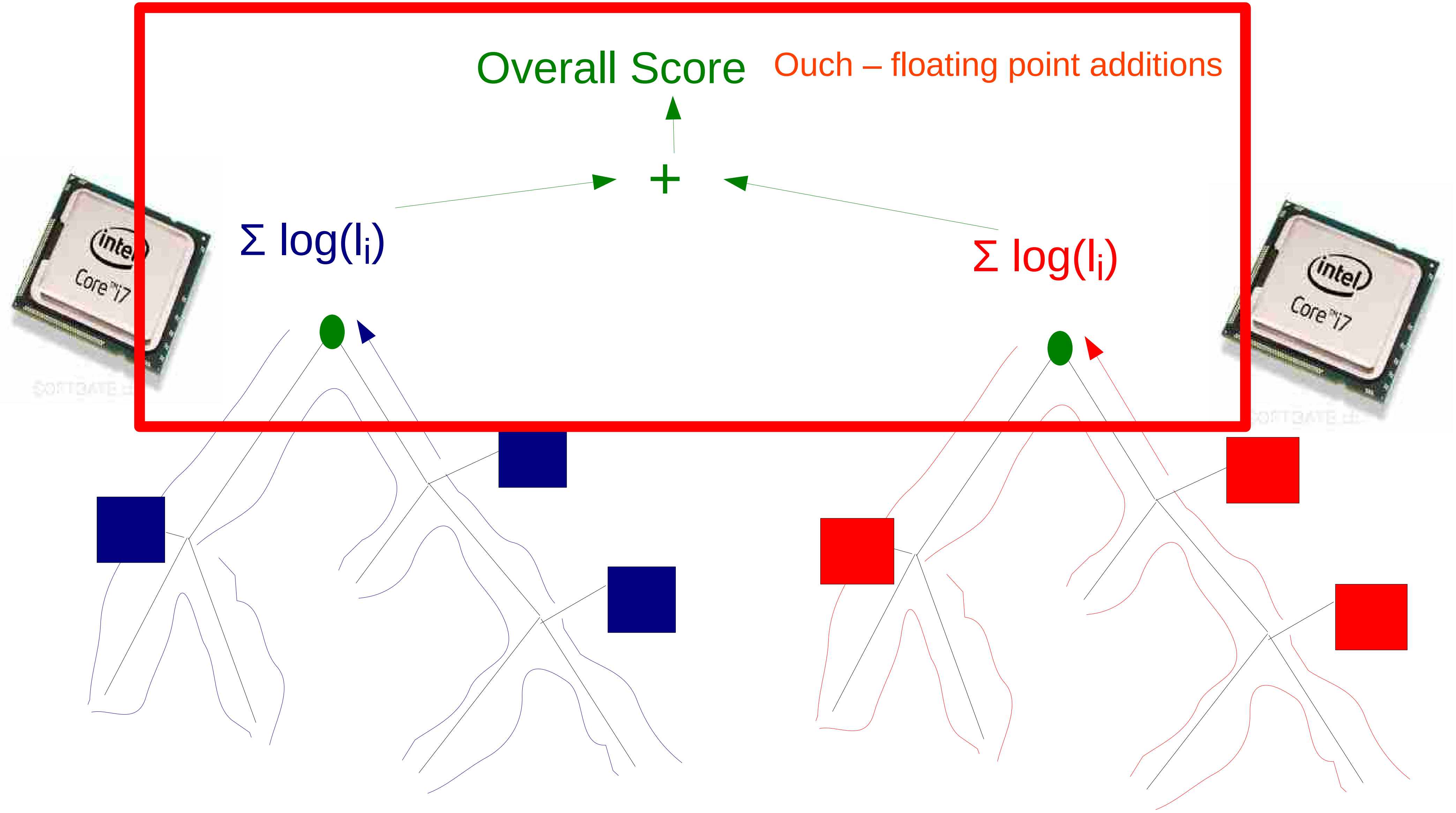
Overall Score

+

$\Sigma \log(l_i)$

$\Sigma \log(l_i)$

Parallel Post-order Traversal



Overall Score

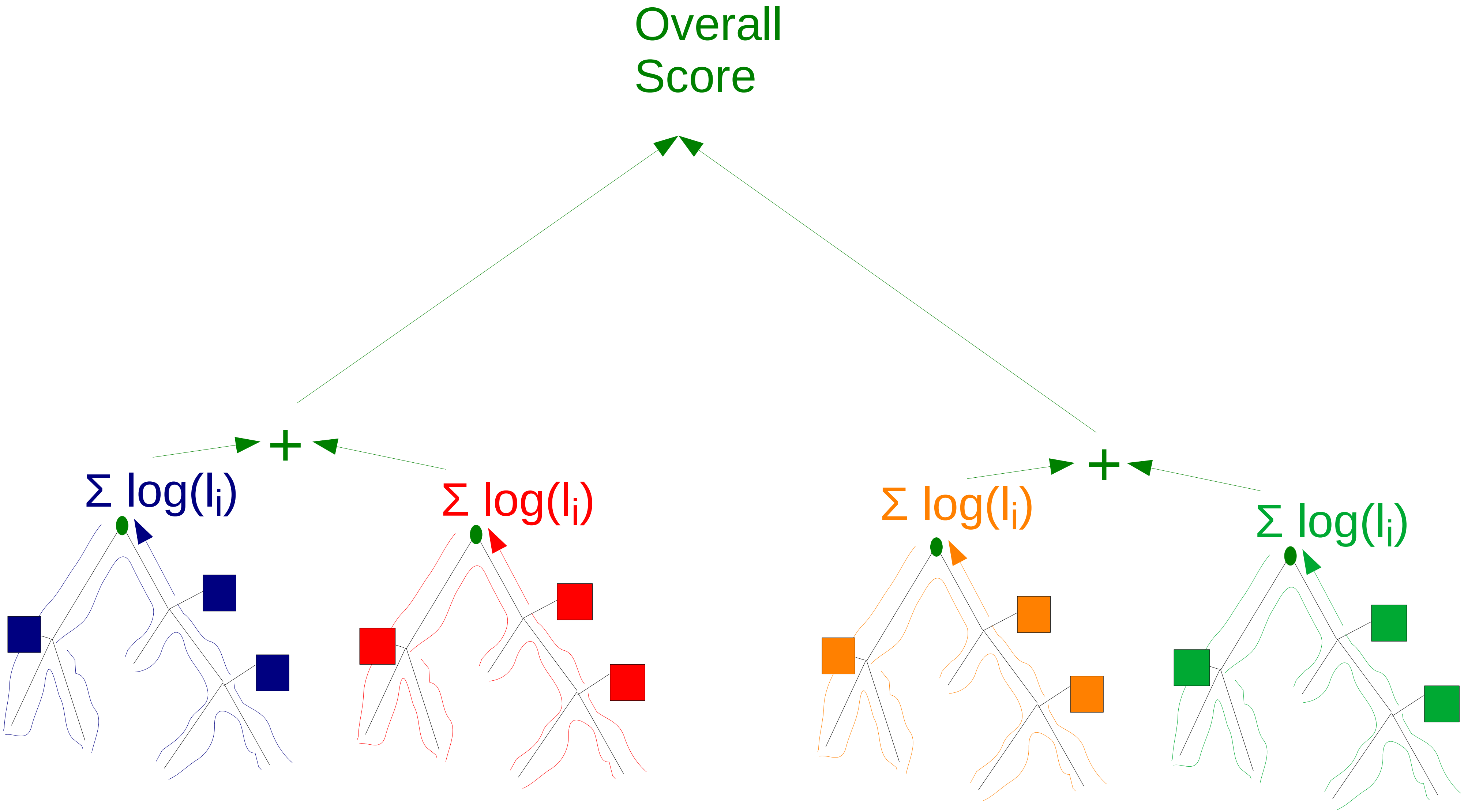
Ouch – floating point additions

+

$\Sigma \log(l_i)$

$\Sigma \log(l_i)$

Parallel Post-order Traversal



Overall Score

+

+

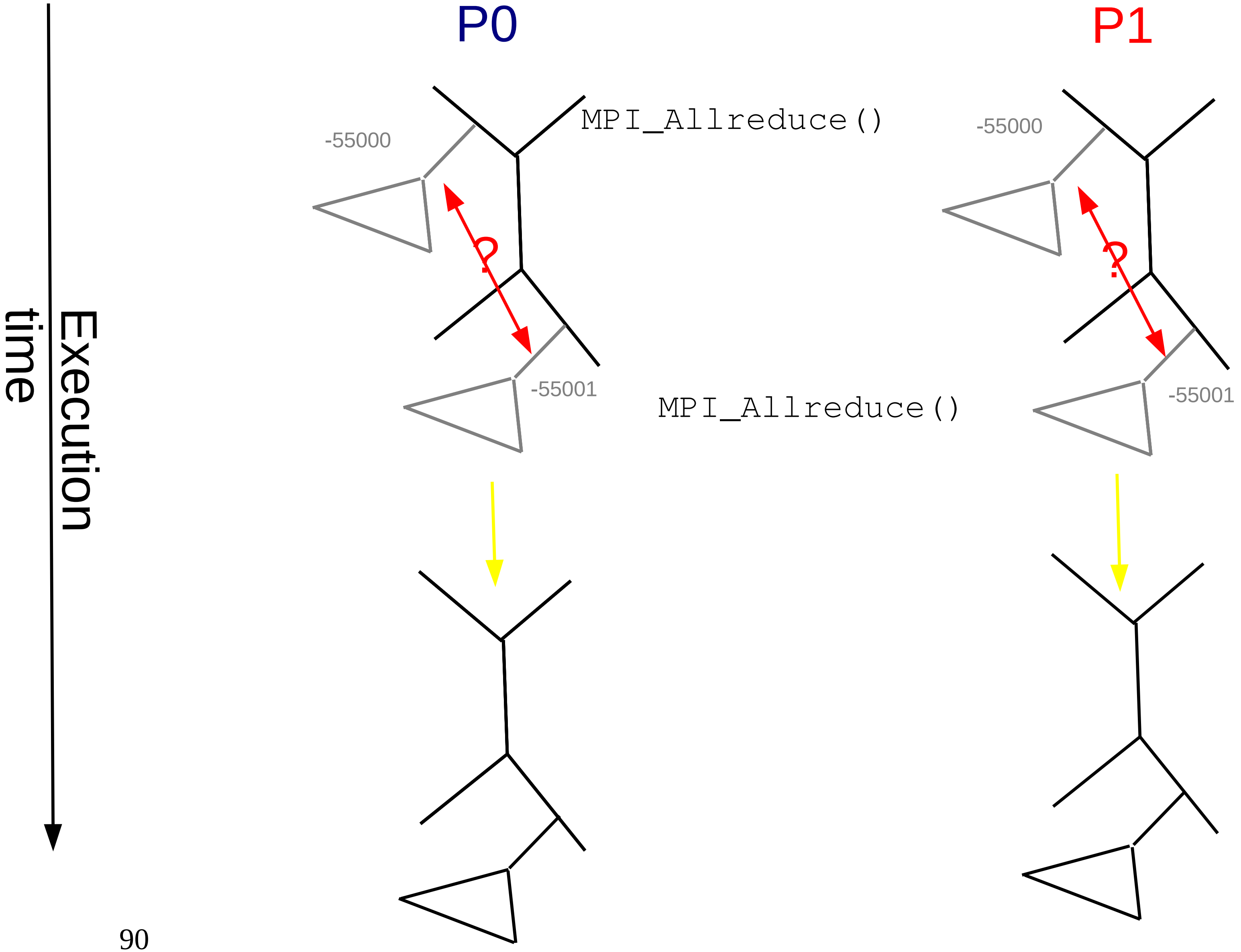
$\Sigma \log(l_i)$

$\Sigma \log(l_i)$

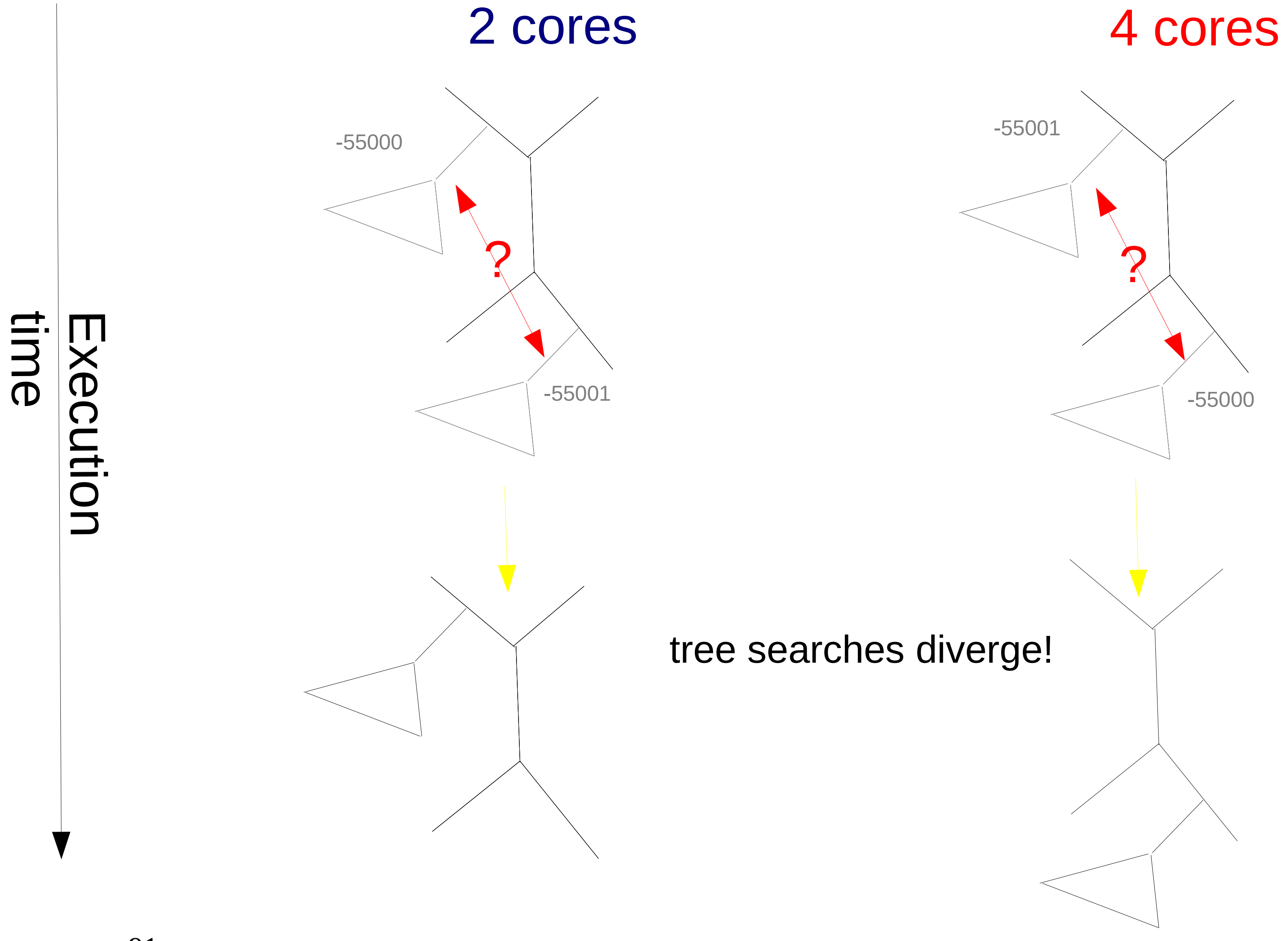
$\Sigma \log(l_i)$

$\Sigma \log(l_i)$

Current MPI parallelization of RAxML-NG



Why? → distinct round off error propagation



2 cores

4 cores

Execution time

tree searches diverge!

`MPI_Allreduce()`

- MSA with 1000 sites
- Two cores calculate LnL for 500 sites each
  - core 0: $\mathrm{LnL}_{[1\text{-}500]}$
  - core 1: $\mathrm{LnL}_{[501\text{-}1000]}$
- After executing an `MPI_Allreduce()`

  both cores have the overall

  $\mathrm{LnL} = \mathrm{LnL}_{[1\text{-}500]} + \mathrm{LnL}_{[501\text{-}1000]}$

  in memory

MPI_Allreduce()

- **Reproducibility:** Ideally we want to get bit-wise identical results regardless of the number of cores we use → **not the case**

`MPI_Allreduce()`

- **Reproducibility:** Ideally we want to get bit-wise identical results regardless of the number of cores we use
- For this we need a reproducible `MPI_Allreduce()`
- **Christoph Stelz** "Core-Count Independent Reproducible Reduce", Bachelor thesis, Institute of Theoretical Computer Science, Karlsruhe Institute of Technology, Germany, April 2022.
- Of course there is a performance trade-off a reproducible `MPI_Allreduce()` has higher computational cost

→ still needs to be assessed in `RAxML-NG`

Cost

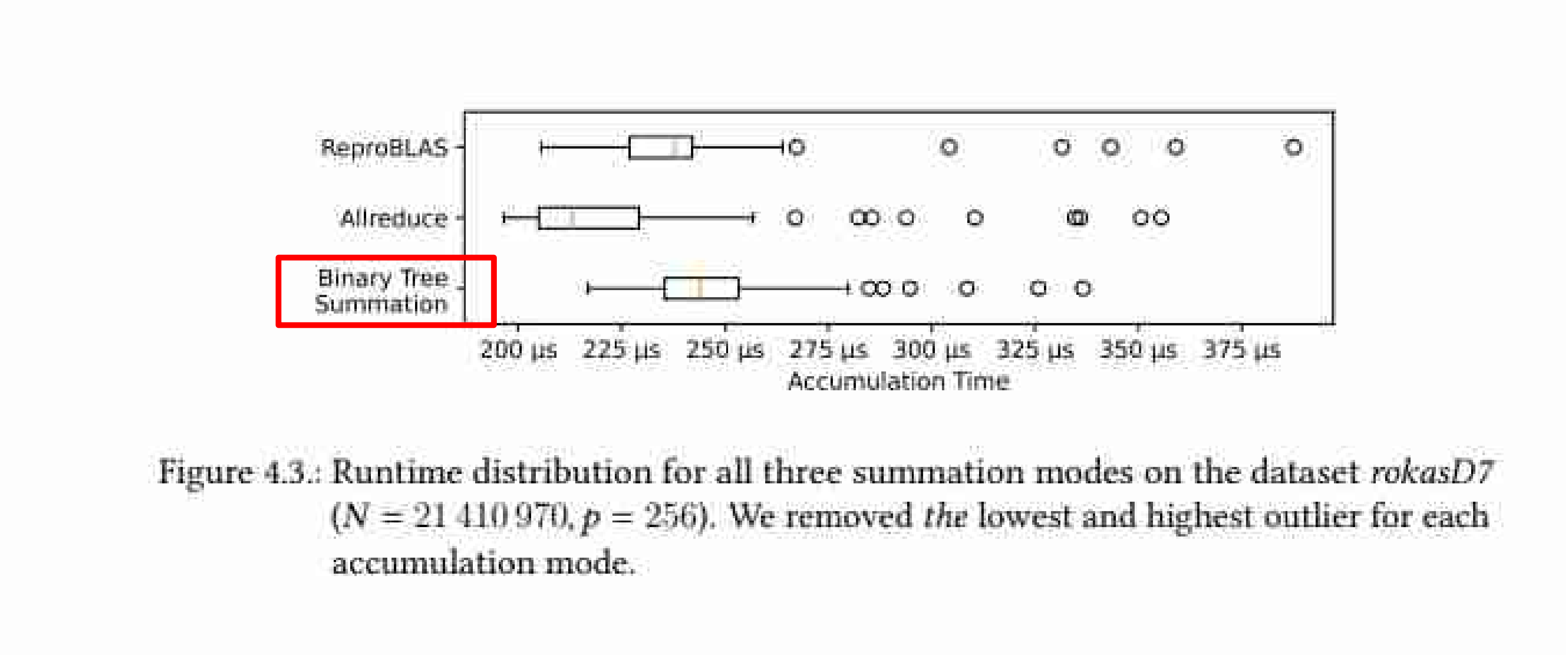


Figure 4.3.: Runtime distribution for all three summation modes on the dataset *rokasD7* ($N = 21\,410\,970$, $p = 256$). We removed *the* lowest and highest outlier for each accumulation mode.

Outline

- The root of all evil
- Sequential Computations
- Parallel Computations
- **Software Quality**

SW Engineering

- As a student I thought that SW engineering is a sub-discipline of philology – and didn’t care
- Many very hard lessons learned with those real-world production level codes !!!!

Project Complexity

the good old days

Sequence → Align

```
T1 ACGT  T1 ACGT
T2 ACC   T2 ACC-
T3 ACGG  T3 ACGG
T4 AAGC  T4 AAGC
```

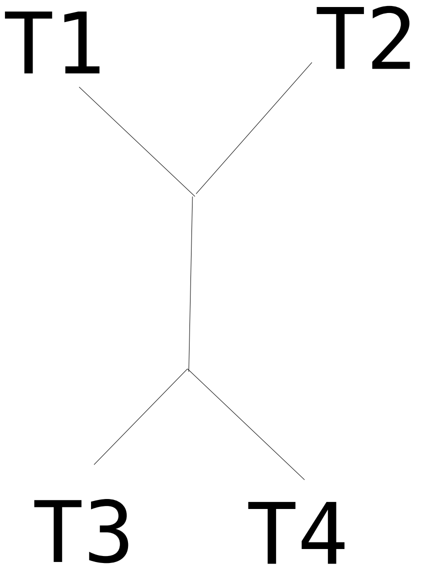
Project Complexity

the good old days

Sequence → Align → Infer Tree

```
T1 ACGT
T2 ACC
T3 ACGG
T4 AAGC
```

```
T1 ACGT
T2 ACC-
T3 ACGG
T4 AAGC
```



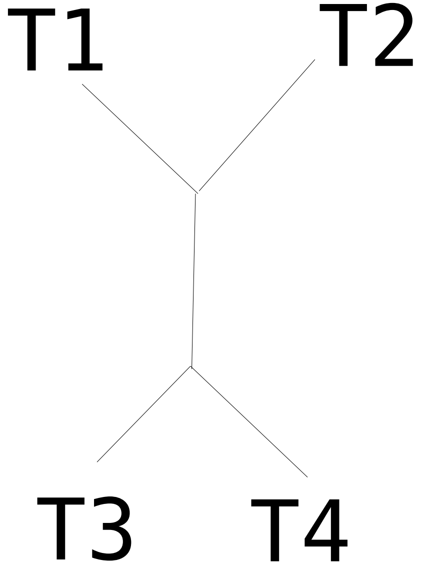
Project Complexity

the good old days

Sequence → Align → Infer Tree → Publish

T1	ACGT
T2	ACC
T3	ACGG
T4	AAGC

T1	ACGT
T2	ACC-
T3	ACGG
T4	AAGC



NO, NO, IF YOU MAKE THE PAPER TOO EASY TO READ, EVERYONE WILL KNOW HOW YOU GOT THE RESULTS!

Project Complexity Today

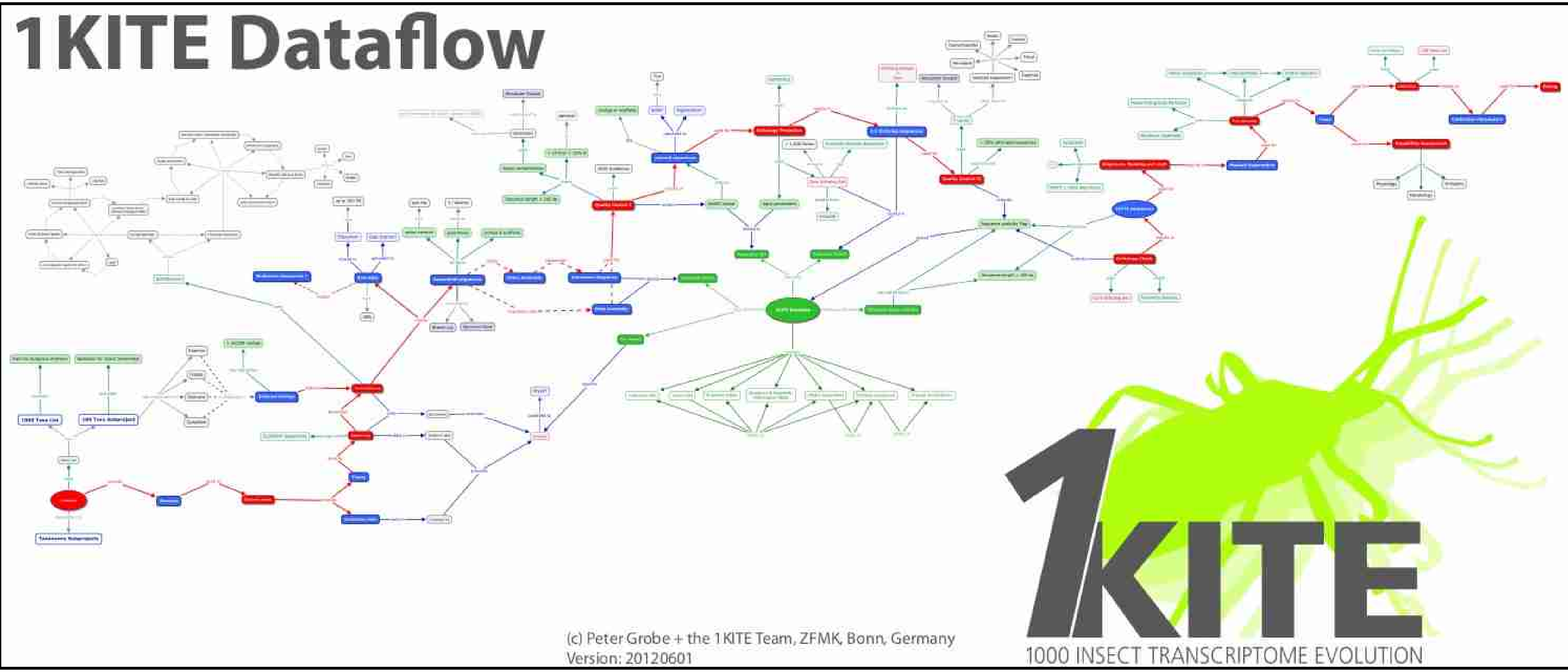
All aTwitter over an Internet study p. XXX
The extragalactic background's uneven glow pp. XXX & XXX
A cellular target for human norovirus pp. XXX & XXX
Science
$10
7 NOVEMBER 2014
sciencemag.org
AAAS
Insect phylogeny resolved
Molecular insights into insect origins and evolution
p. XXX

150 insect transcriptomes

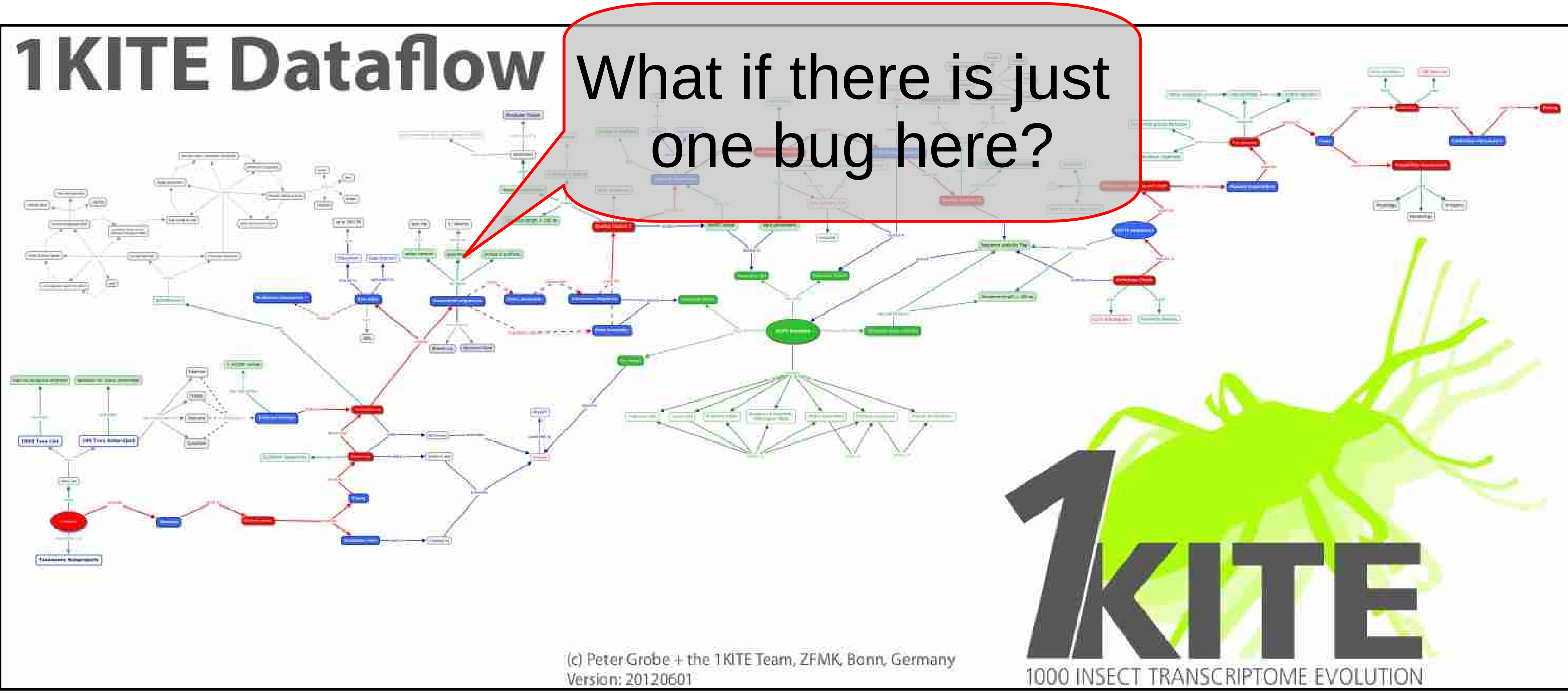
Does DNA act as a telephone line? p. 1284
Mutations enhancing leukemia development pp. 1291 & 1373
A stable gold support gives sharper resolution p. 1377
Science
$10
12 DECEMBER 2014
sciencemag.org
AAAS
SPECIAL ISSUE
Avian genomes
Sequencing across the bird species tree
p. 1308

50 bird genomes

Project Complexity Today

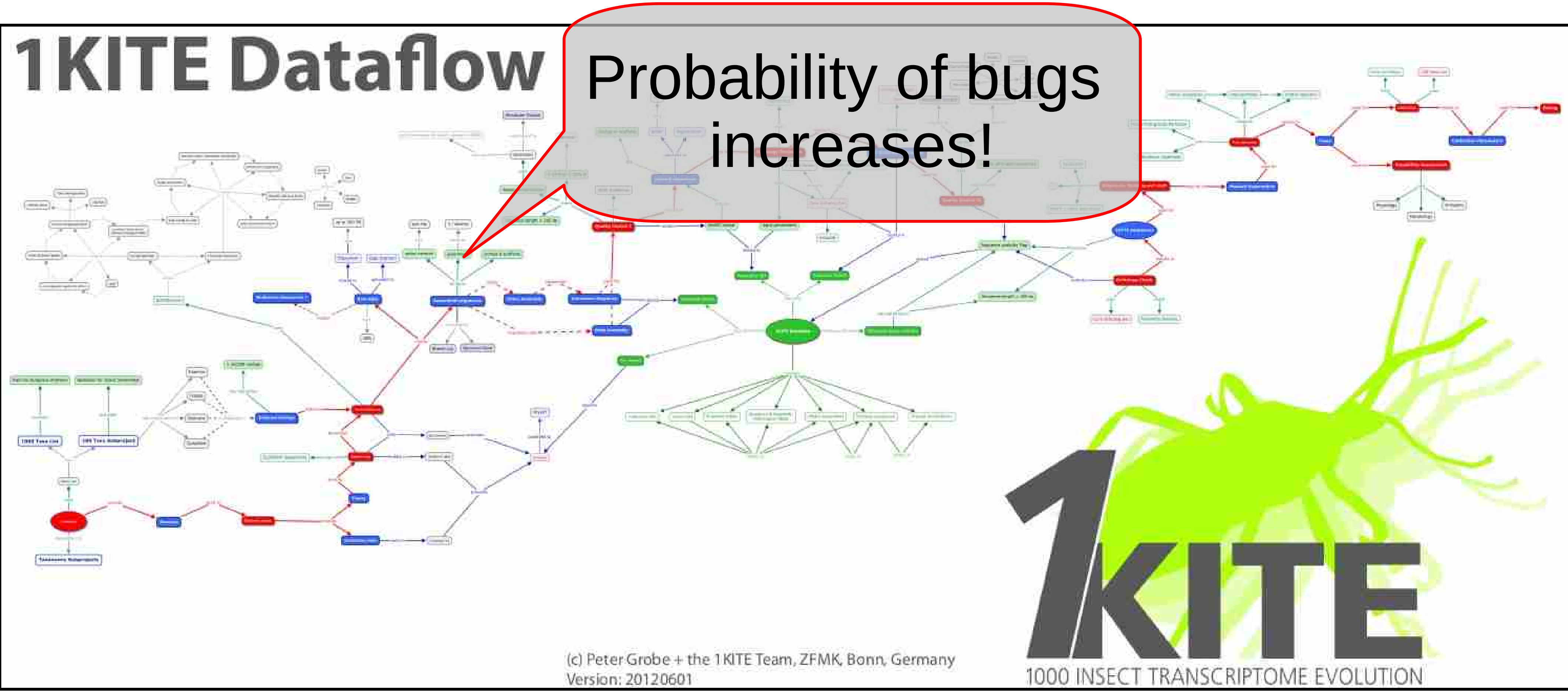

1KITE Dataflow
(c) Peter Grobe + the 1KITE Team, ZFMK, Bonn, Germany
Version: 20120601
1KITE
1000 INSECT TRANSCRIPTOME EVOLUTION

Project Complexity Today


1KITE Dataflow
(c) Peter Grobe + the 1KITE Team, ZFMK, Bonn, Germany
Version: 20120601
1KITE
1000 INSECT TRANSCRIPTOME EVOLUTION

What if there is just one bug here?

Project Complexity Today

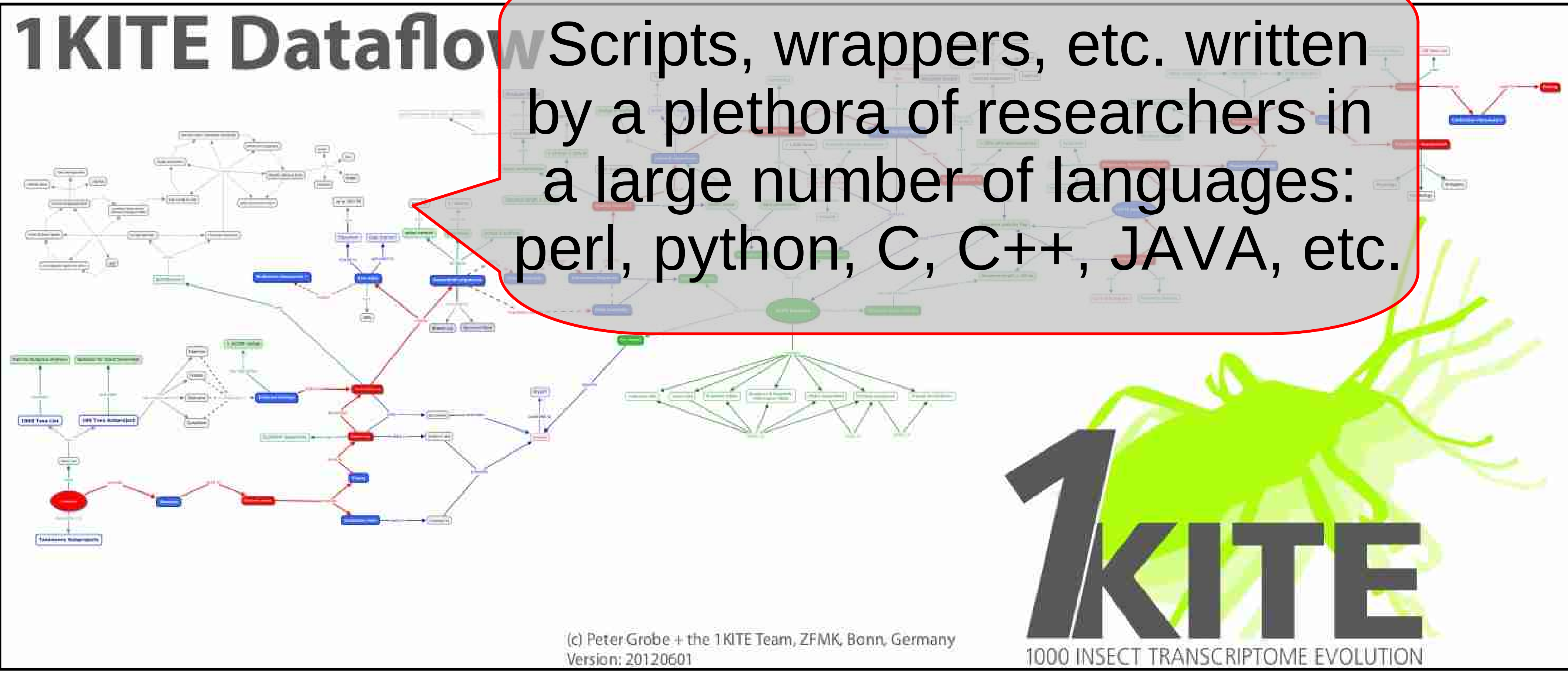


1KITE Dataflow

Probability of bugs increases!

(c) Peter Grobe + the 1KITE Team, ZFMK, Bonn, Germany
Version: 20120601

Project Complexity Today

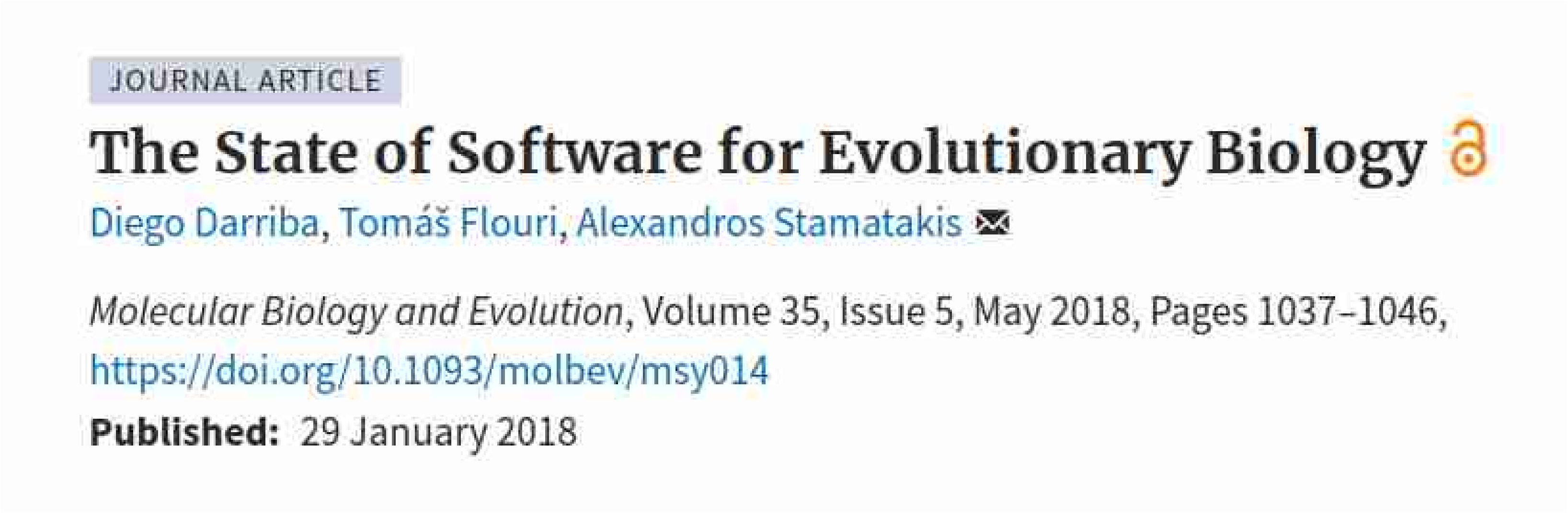


(c) Peter Grobe + the 1KITE Team, ZFMK, Bonn, Germany
Version: 20120601

Scripts, wrappers, etc. written by a plethora of researchers in a large number of languages: perl, python, C, C++, JAVA, etc.

Bioinformatics Tools

- We knew many tools are pretty awful
- Numerous self-taught programmers from application domains
- So we did some manual analyses and started ranting



JOURNAL ARTICLE

The State of Software for Evolutionary Biology

Diego Darriba, Tomáš Flouri, Alexandros Stamatakis

Molecular Biology and Evolution, Volume 35, Issue 5, May 2018, Pages 1037–1046, https://doi.org/10.1093/molbev/msy014

Published: 29 January 2018

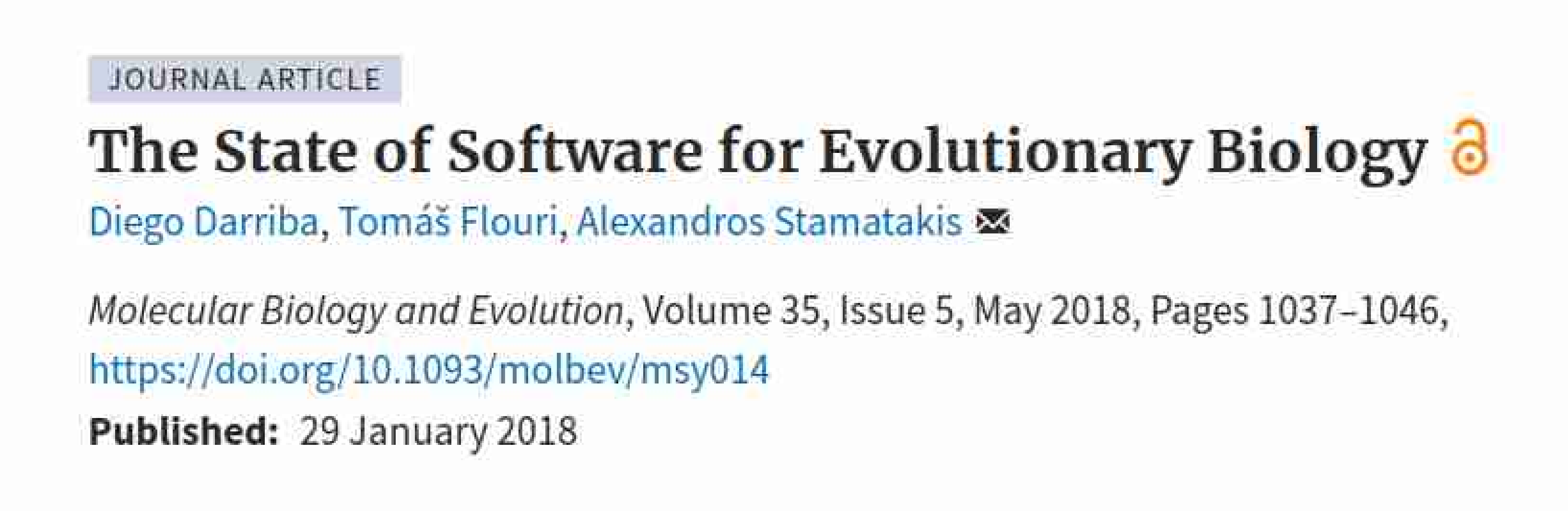
The 'crappy' software project

Internal name of the project
in our lab

The 'crappy' software project

- Analyzed *15* widely-used evolutionary biology tools ≈ *65,000* citations
- Analyses performed
  - Compiled with `gcc` and `clang` with all warnings enabled
  - Memory check with `valgrind`
  - Checked if assertions are used via `assert()`
  - Analyzed degree of code duplication
- **Caution:** “bad” quality does not induce that a tool is faulty, but the probability of it being faulty is higher!

The birth of `SoftWipe`



JOURNAL ARTICLE

The State of Software for Evolutionary Biology

Diego Darriba, Tomáš Flouri, Alexandros Stamatakis

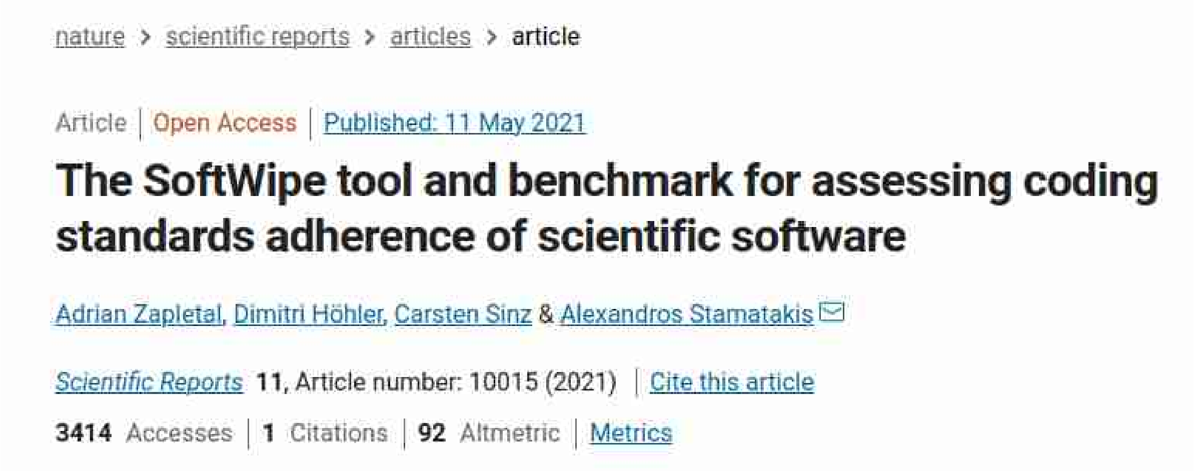
Molecular Biology and Evolution, Volume 35, Issue 5, May 2018, Pages 1037–1046, https://doi.org/10.1093/molbev/msy014

Published: 29 January 2018

- Chatting with a science journalist about the above paper he asked me if this code analysis process can be automated → the start of the `SoftWipe` project

SoftWipe

- Development of SoftWipe - An automated tool and benchmark for **relative** quality ranking of scientific software
- Ranking of *53* open source tools written in C or C++ from a wide range of research areas
    - Astrophysics



nature > scientific reports > articles > article

Article | Open Access | Published: 11 May 2021

The SoftWipe tool and benchmark for assessing coding standards adherence of scientific software

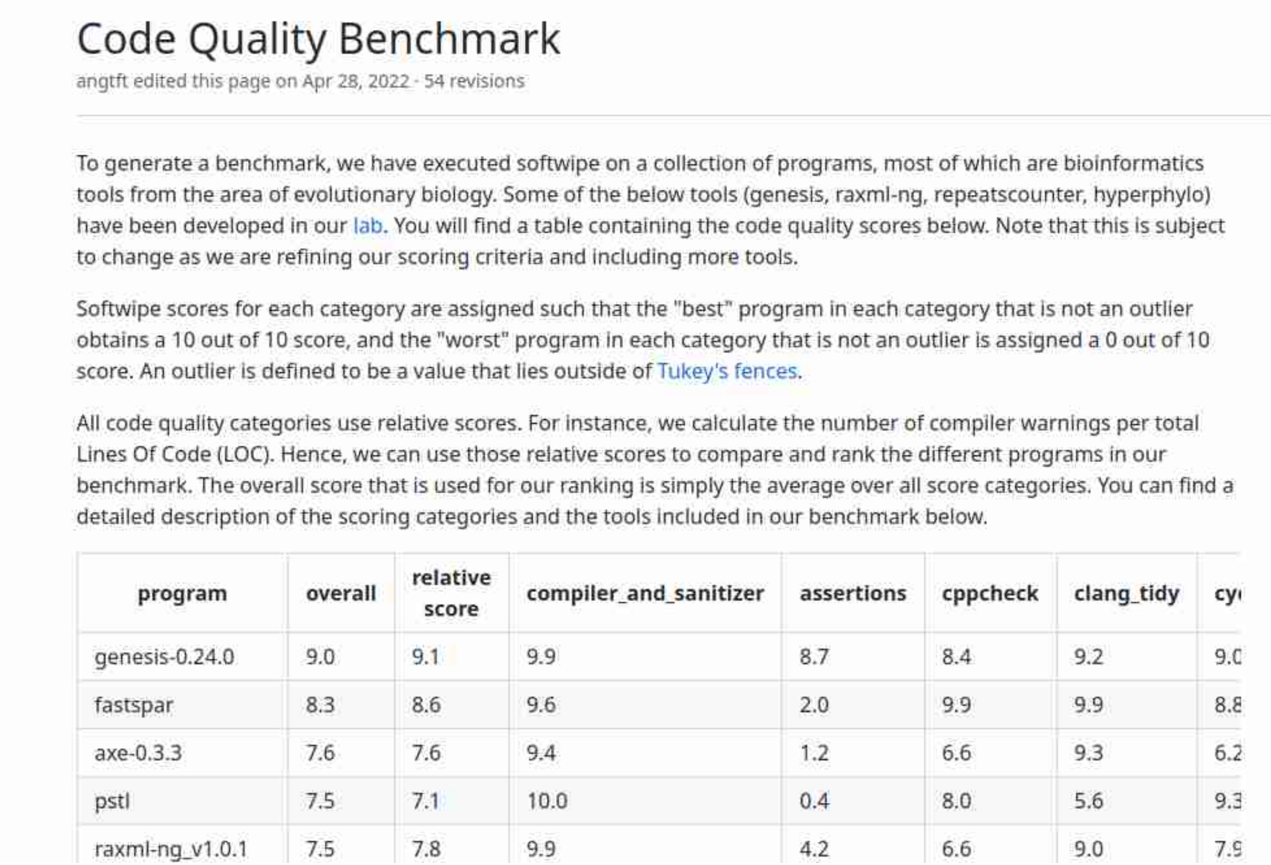
Adrian Zapletal, Dimitri Höhler, Carsten Sinz & Alexandros Stamatakis

Scientific Reports **11**, Article number: 10015 (2021) | Cite this article

3414 Accesses | **1** Citations | **92** Altmetric | Metrics

Benchmark

- Available at https://github.com/adrianzap/softwipe/wiki/Code-Quality-Benchmark



Code Quality Benchmark

angtft edited this page on Apr 28, 2022 · 54 revisions

To generate a benchmark, we have executed softwipe on a collection of programs, most of which are bioinformatics tools from the area of evolutionary biology. Some of the below tools (genesis, raxml-ng, repeatscounter, hyperphylo) have been developed in our lab. You will find a table containing the code quality scores below. Note that this is subject to change as we are refining our scoring criteria and including more tools.

Softwipe scores for each category are assigned such that the "best" program in each category that is not an outlier obtains a 10 out of 10 score, and the "worst" program in each category that is not an outlier is assigned a 0 out of 10 score. An outlier is defined to be a value that lies outside of Tukey's fences.

All code quality categories use relative scores. For instance, we calculate the number of compiler warnings per total Lines Of Code (LOC). Hence, we can use those relative scores to compare and rank the different programs in our benchmark. The overall score that is used for our ranking is simply the average over all score categories. You can find a detailed description of the scoring categories and the tools included in our benchmark below.

program	overall	relative score	compiler_and_sanitizer	assertions	cppcheck	clang_tidy	cy
genesis-0.24.0	9.0	9.1	9.9	8.7	8.4	9.2	9.0
fastspar	8.3	8.6	9.6	2.0	9.9	9.9	8.8
axe-0.3.3	7.6	7.6	9.4	1.2	6.6	9.3	6.2
pstl	7.5	7.1	10.0	0.4	8.0	5.6	9.3
raxml-ng_v1.0.1	7.5	7.8	9.9	4.2	6.6	9.0	7.9

`SoftWipe` Criteria I

- Per criterion, calculate & assign score 0-10 such that
  - "best" program under criterion *that is not an outlier* gets 10/10
  - "worst" program under criterion *that is not an outlier* gets 0/10
  - Outliers: values that are outside of Tukey's fences.
- Then just take the unweighted average over all criteria to get an overall `SoftWipe` score
- We apply some corrections such that the global score does not change when more tools are added to the benchmark (details omitted)

`SoftWipe` Criteria II

- **compiler and sanitizer**: use `clang` compiler and count the number of warnings - we activate almost all warnings for this. Warnings are weighted - each warning has a *weight* of *1, 2,* or *3*, where *3* is most dangerous (this is totally subjective).

  We also use `clang sanitizers` (`ASan` and `UBSan`) - if they yield warnings, we add them to the weighted warning sum above with weight *3*. The compiler and sanitizer score is calculated from the weighted sum of warnings per total *LOC*.

- **assertions:** The count of assertions (C-Style `assert()`, `static_assert()`, or custom `assert macros`, if defined) per total *LOC*.

- **cppcheck:** #warnings found by the static code analyzer `cppcheck` per total *LOC*. `cppcheck` also categorizes warnings → analogous weighting as for compiler warnings.

- **clang-tidy:** #warnings found by the static code analyzer `clang-tidy` per total *LOC*. `clang-tidy` also categorizes warnings → analogous weigthing as for compiler warnings.

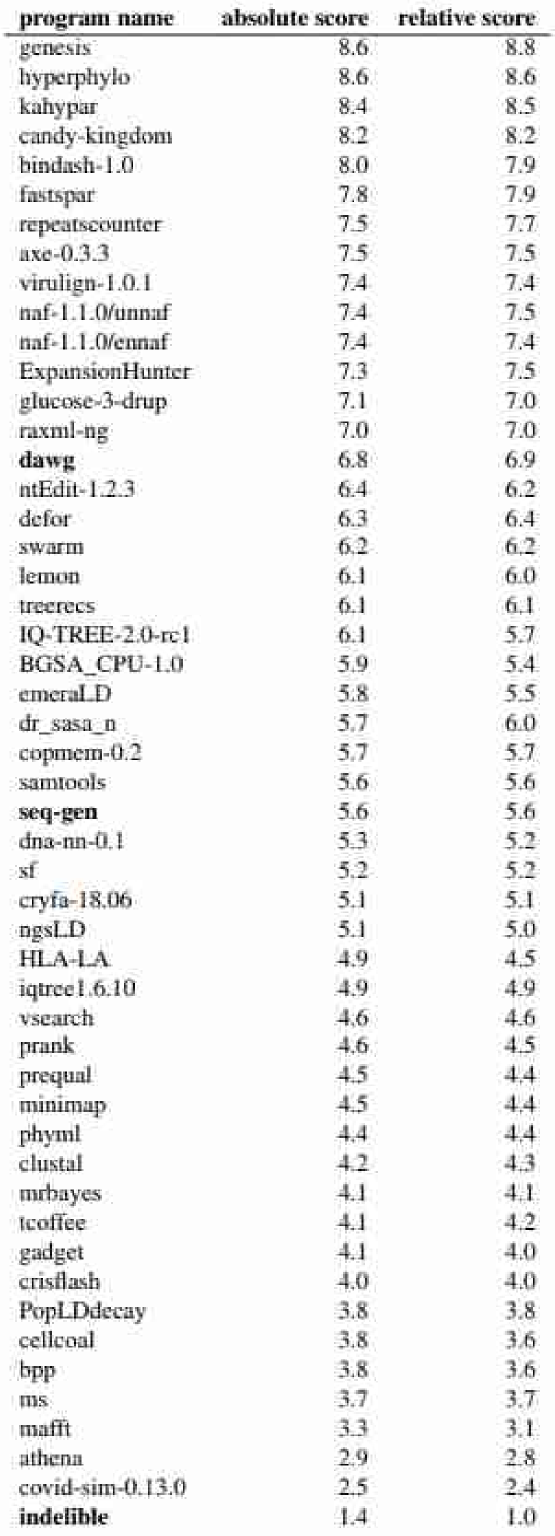
SoftWipe Criteria III

- **cyclomatic complexity:** software metric to quantify the complexity/modularity of a program.

  We use lizard to assess the cyclomatic complexity of a source code.
- **lizard warnings:** Number of functions that are considered too complex, relative to the total number of functions - lizard considers a function as "too complex" if its cyclomatic complexity, its length, or its parameter count exceeds a certain threshold value.
- **unique rate:** amount of unique code; a higher amount of duplicated code yields a lower value. Also computed with lizard.
- **kwstyle:** #warnings found by the static code style analyzer KWStyle per total *LOC*. We configure KWStyle using the KWStyle.xml file that ships with SoftWipe.
- **infer**: we weight the warnings found by the static analyzer Infer and use the weighted warnings per *LOC* rate to calculate a score.
- **test count:** We try to relate the unit test *LOC* in with overall *LOC* count by compting: *test_code_loc / overall_loc* . The detection of unit test *LOC* should be improved – at present we interpret source files containing the keyword "test" in their path as unit test files.

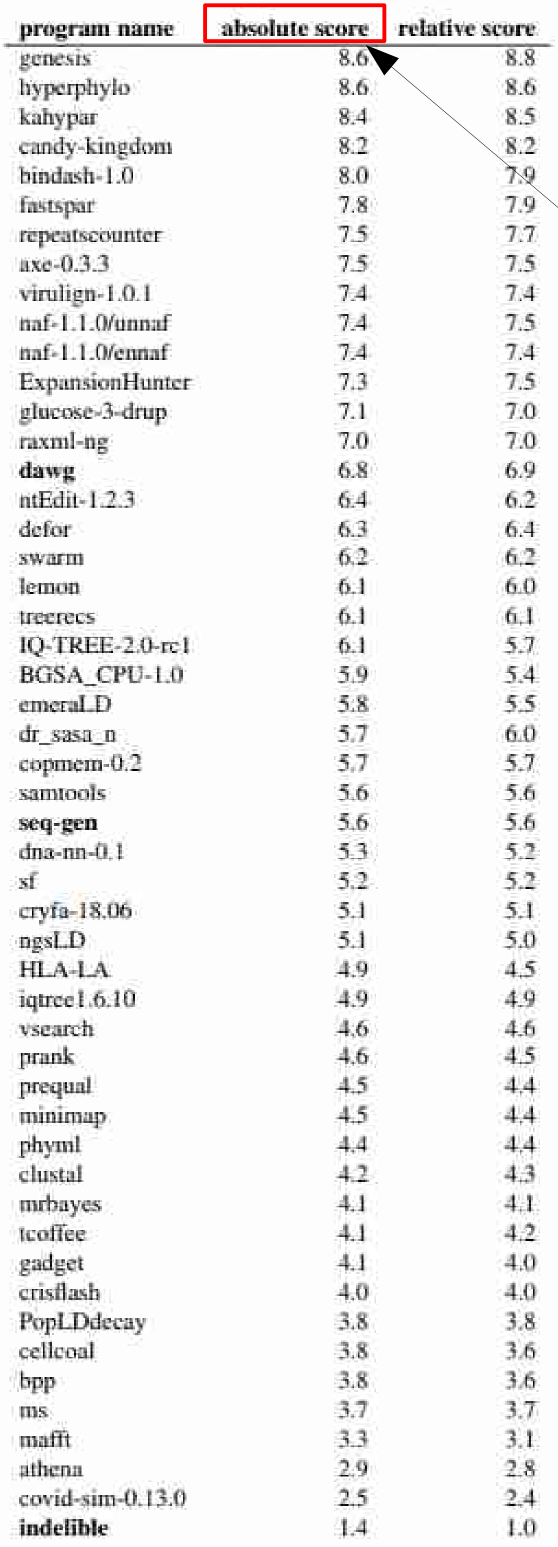
Let’s have a look at the benchmark

SoftWipe Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

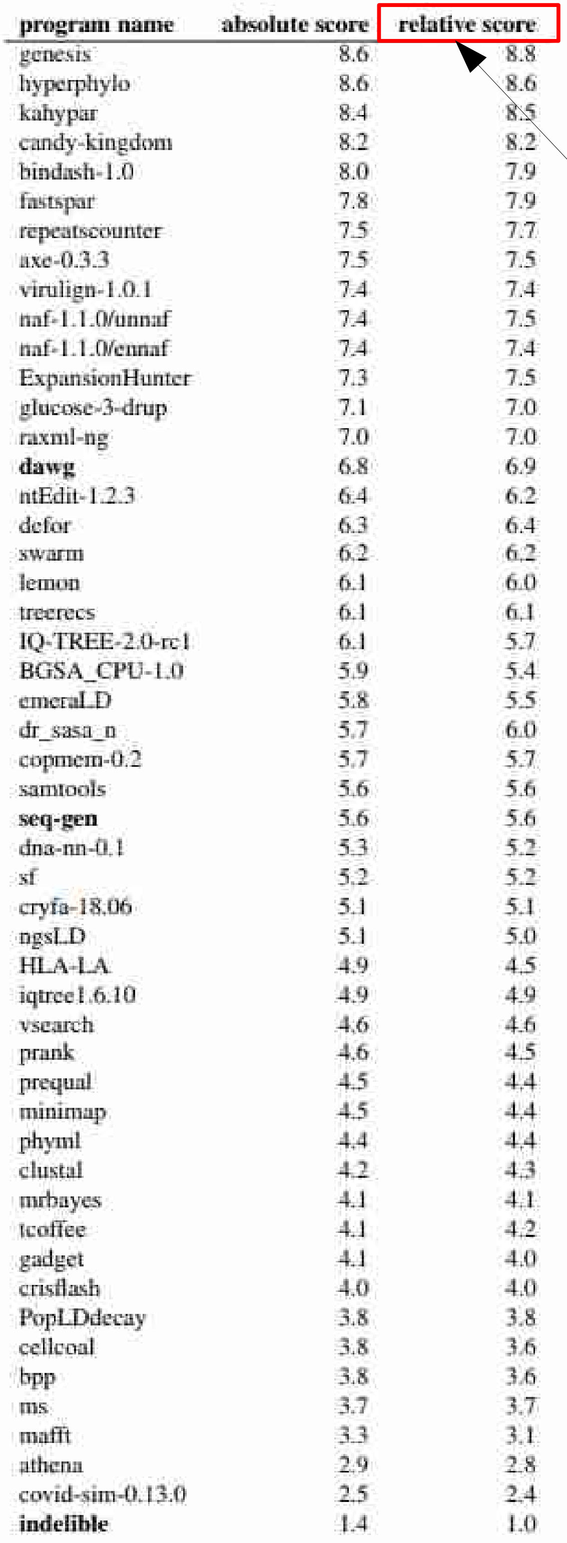
SoftWipe
Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Does not change over time as more tools are added → can easily be referenced

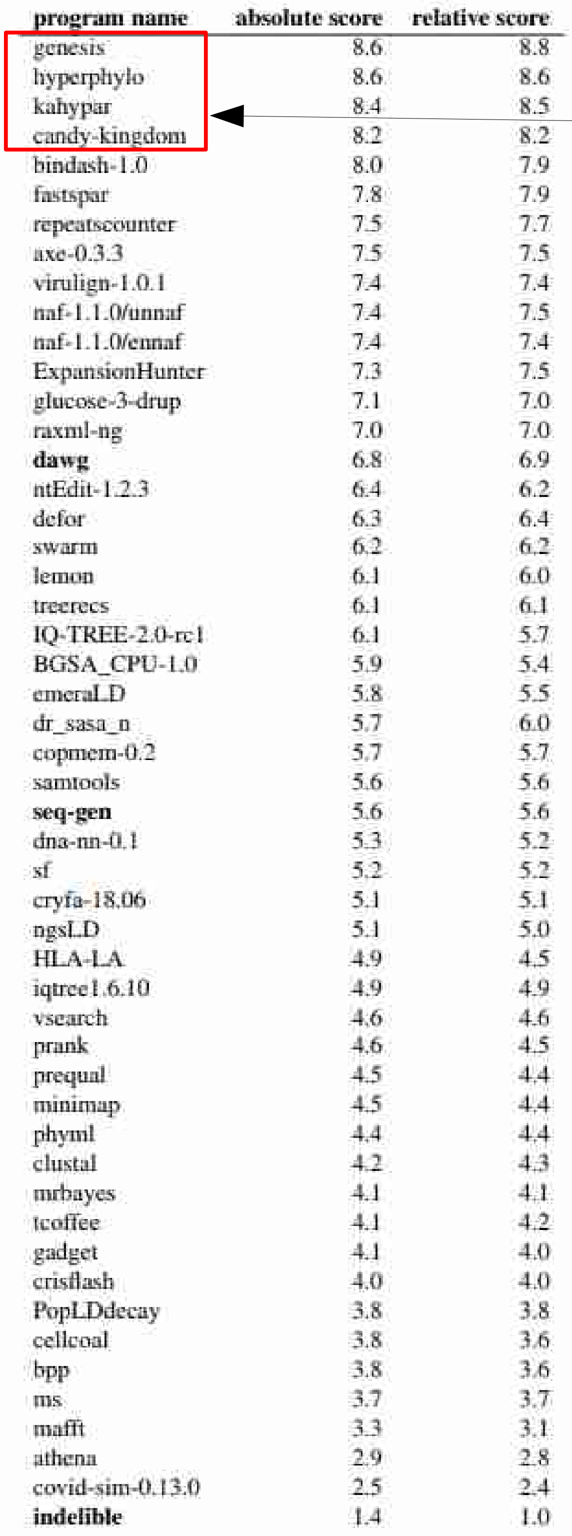
SoftWipe Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Does change over time as more tools are added →
Difficult to be referenced

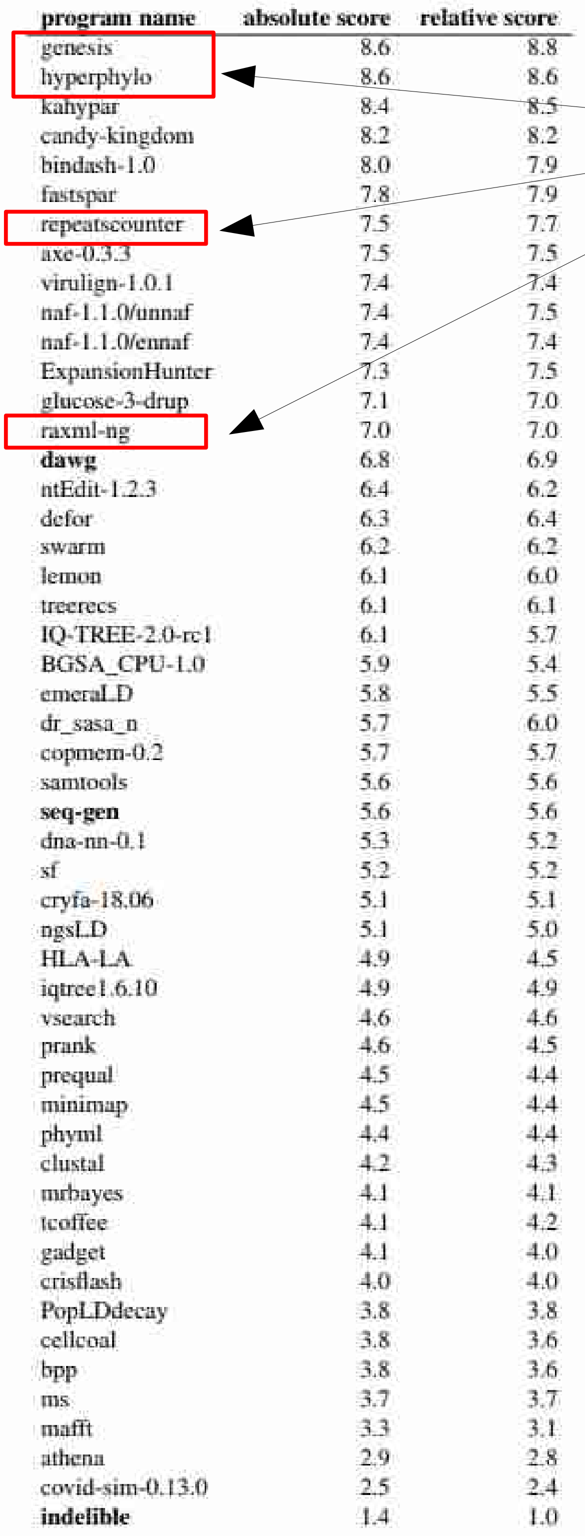
SoftWipe Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Written by computer Scientists :-)

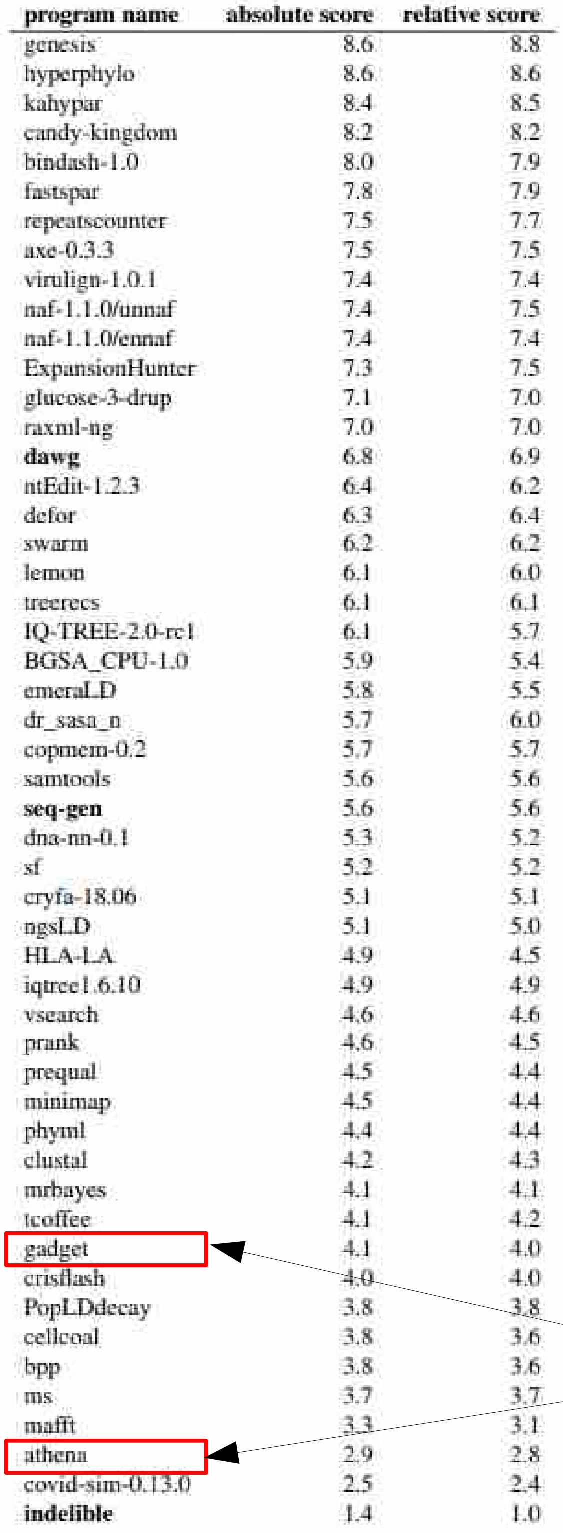
SoftWipe
Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

My lab in Germany :-)

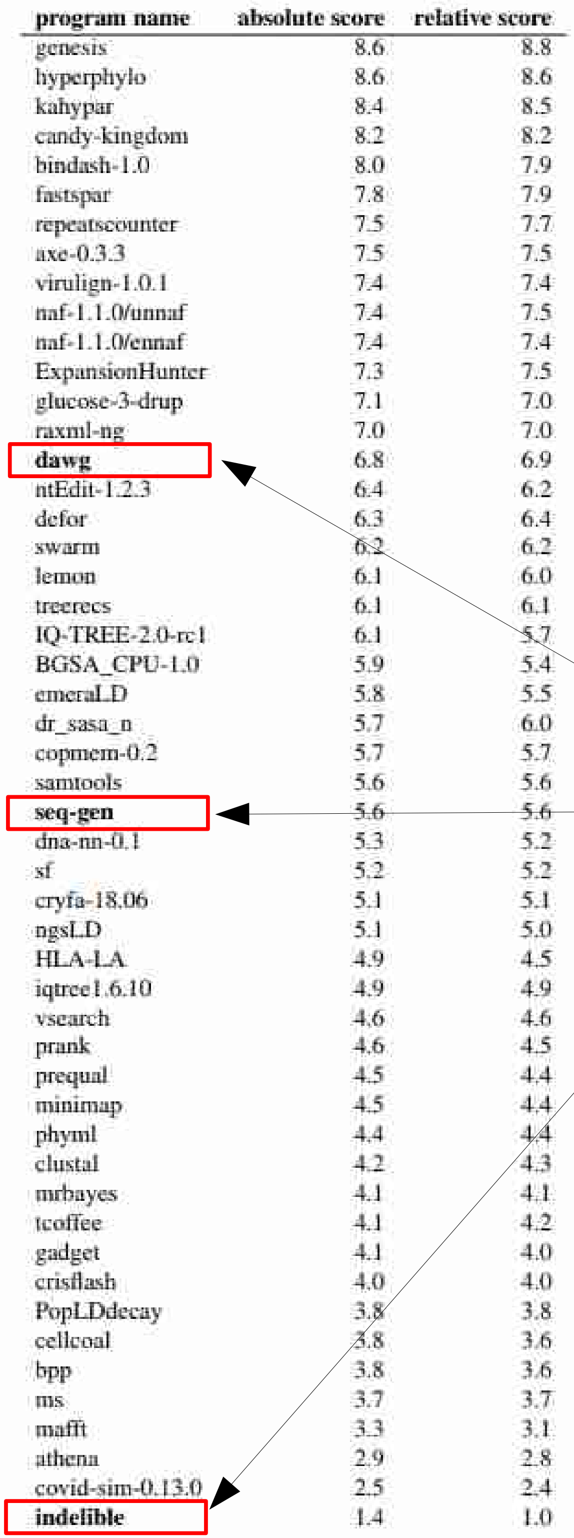
SoftWipe Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Astrophysics

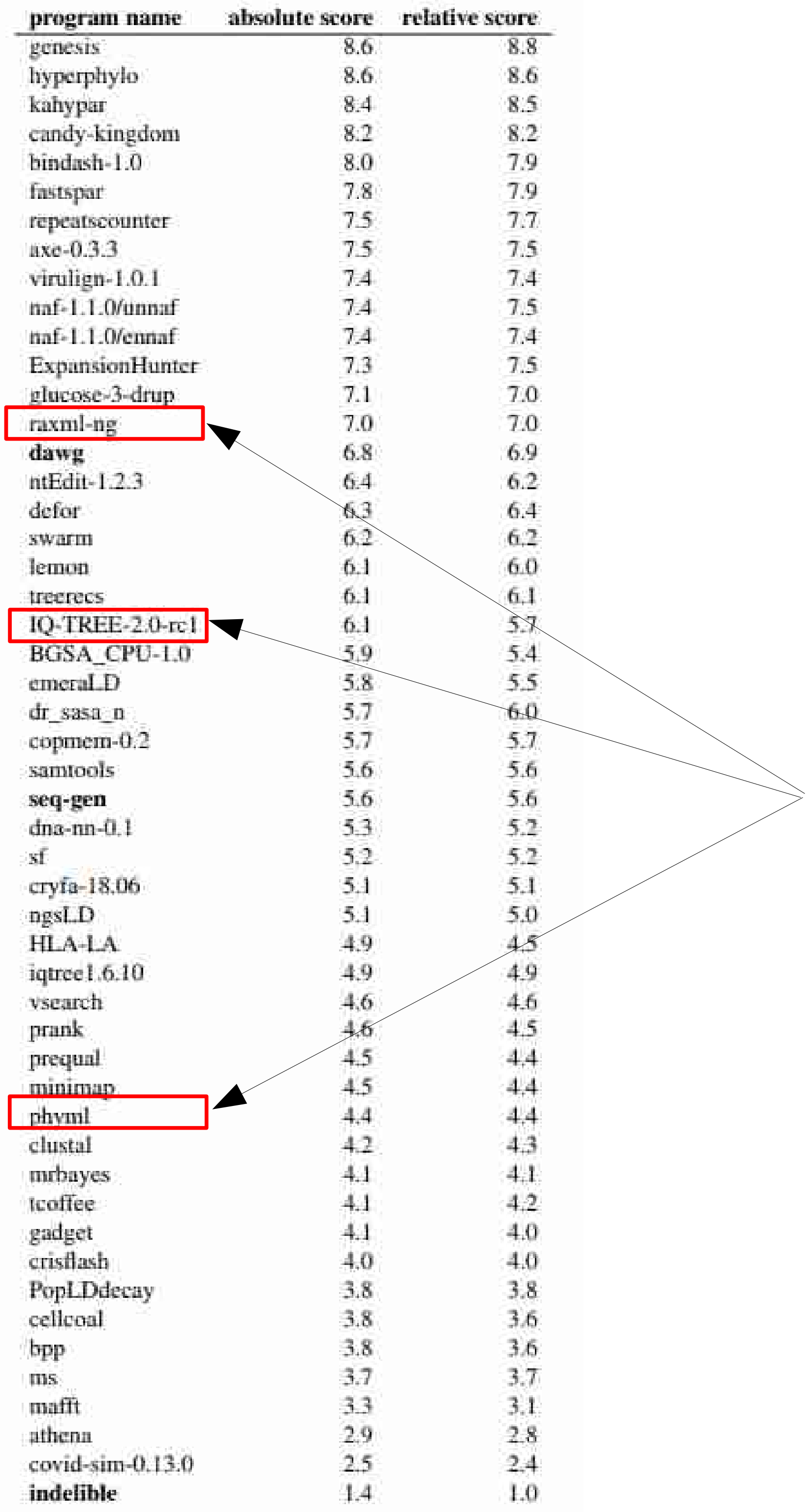
SoftWipe Benchmark



program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Tools with highly similar functionality

SoftWipe
Benchmark

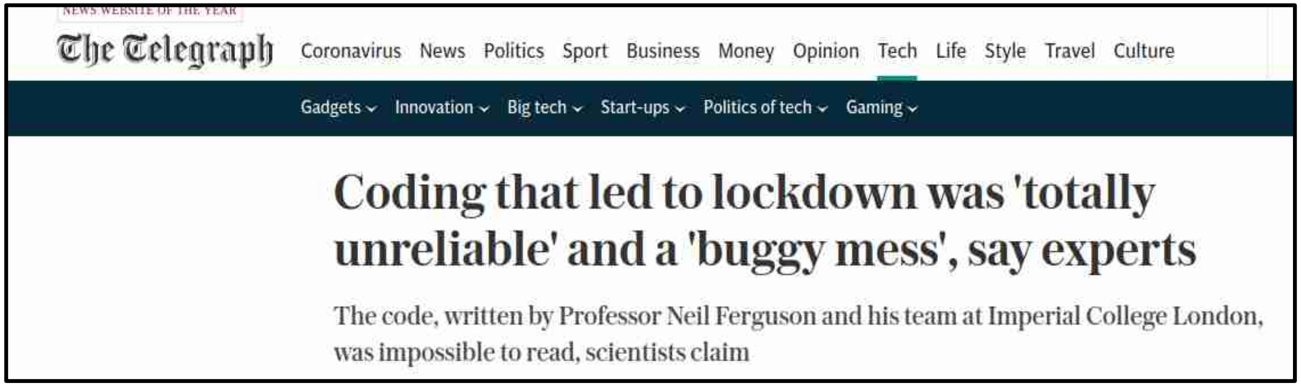


program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0

Tools with highly similar functionality

SoftWipe Benchmark

program name	absolute score	relative score
genesis	8.6	8.8
hyperphylo	8.6	8.6
kahypar	8.4	8.5
candy-kingdom	8.2	8.2
bindash-1.0	8.0	7.9
fastspar	7.8	7.9
repeatscounter	7.5	7.7
axe-0.3.3	7.5	7.5
virulign-1.0.1	7.4	7.4
naf-1.1.0/unnaf	7.4	7.5
naf-1.1.0/ennaf	7.4	7.4
ExpansionHunter	7.3	7.5
glucose-3-drup	7.1	7.0
raxml-ng	7.0	7.0
dawg	6.8	6.9
ntEdit-1.2.3	6.4	6.2
defor	6.3	6.4
swarm	6.2	6.2
lemon	6.1	6.0
treerecs	6.1	6.1
IQ-TREE-2.0-rc1	6.1	5.7
BGSA_CPU-1.0	5.9	5.4
emeraLD	5.8	5.5
dr_sasa_n	5.7	6.0
copmem-0.2	5.7	5.7
samtools	5.6	5.6
seq-gen	5.6	5.6
dna-nn-0.1	5.3	5.2
sf	5.2	5.2
cryfa-18.06	5.1	5.1
ngsLD	5.1	5.0
HLA-LA	4.9	4.5
iqtree1.6.10	4.9	4.9
vsearch	4.6	4.6
prank	4.6	4.5
prequal	4.5	4.4
minimap	4.5	4.4
phyml	4.4	4.4
clustal	4.2	4.3
mrbayes	4.1	4.1
tcoffee	4.1	4.2
gadget	4.1	4.0
crisflash	4.0	4.0
PopLDdecay	3.8	3.8
cellcoal	3.8	3.6
bpp	3.8	3.6
ms	3.7	3.7
mafft	3.3	3.1
athena	2.9	2.8
covid-sim-0.13.0	2.5	2.4
indelible	1.4	1.0


NEWS WEBSITE OF THE YEAR
The Telegraph
Coronavirus News Politics Sport Business Money Opinion Tech Life Style Travel Culture
Gadgets Innovation Big tech Start-ups Politics of tech Gaming

Coding that led to lockdown was 'totally unreliable' and a 'buggy mess', say experts

The code, written by Professor Neil Ferguson and his team at Imperial College London, was impossible to read, scientists claim

Covid simulation tool

`SoftWipe` in Practice

- Leads to healthy competition among lab members → everyone wants to write the cleanest code
- Used by researchers inside and outside of the lab during the development process

  → potential bugs identified and avoided (e.g., bug that yielded plausible results and would have gone undetected)

  → yielded improved performance (inlining warnings fixed)

  → used in *Continuous Integration* tool
- Used as teaching tool in programming practicals
- `SoftWipe` score already used by us and others in Bioinformatics software paper submissions

  From the Abstract: *Finally, Lagrange-NG exhibits substantially higher adherence to coding quality standards. It improves a respective software quality indicator as implemented in the SoftWipe tool from average (5.5; Lagrange) to high (7.8; Lagrange-NG)*
- **Vision:** Establish software quality indicators as a necessary prerequisite for (Bioinformatics) software paper submissions

Software Quality and Maintainability

- The Next Generation (**-NG**) projects:
  - Re-design, re-factoring, from scratch re-implementation of flagship tools to ensure maintainability, sustainability, and extensibility & increase scalability/performance
  - `ModelTest-NG` – model testing of evolutionary models for phylogenetic inference
  - `RAxML-NG` – phylogenetic inference
  - `EPA-NG` – phylogenetic placement of environmental reads
  - `Lagrange-NG` – Biogeography tool

A Bachelor Thesis

- One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality.

A Bachelor Thesis

- One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality.
- To investigate this profound matter we crawled and analyzed over 3800 C open source code containing English swearwords and over 7600 C open source code not containing swearwords from GitHub.

A Bachelor Thesis

- One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality.
- To investigate this profound matter we crawled and analyzed over 3800 C open source code containing English swearwords and over 7600 C open source code not containing swearwords from GitHub.
- We find that open source code containing swearwords exhibit significantly better code quality than those not containing swearwords under several statistical tests.

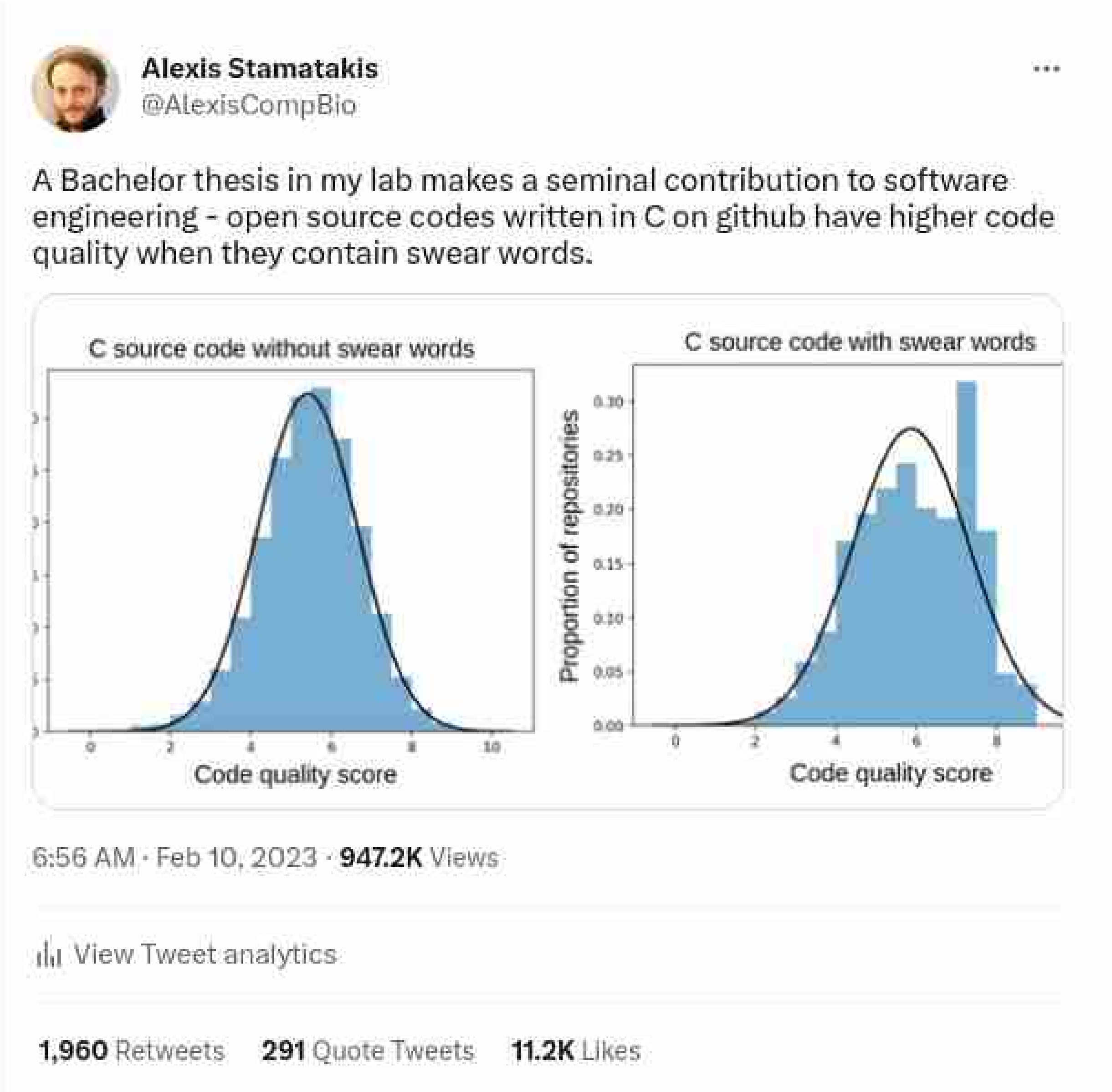
A Bachelor Thesis

- One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality.
- To investigate this profound matter we crawled and analyzed over 3800 C open source code containing English swearwords and over 7600 C open source code not containing swearwords from GitHub.
- We find that open source code containing swearwords exhibit significantly better code quality than those not containing swearwords under several statistical tests.
- We hypothesise that the use of swearwords constitutes an indicator of a profound emotional involvement of the programmer with the code and its inherent complexities, thus yielding better code based on a thorough, critical, and dialectic code analysis process.

A Bachelor Thesis

- One of the most fundamental unanswered questions that has been bothering mankind during the Anthropocene is whether the use of swearwords in open source code is positively or negatively correlated with source code quality.
- To investigate this profound matter we crawled and analyzed over 3800 C open source code containing English swearwords and over 7600 C open source code not containing swearwords from GitHub.
- We find that open source code containing swearwords exhibit significantly better code quality than those not containing swearwords under several statistical tests.
- We hypothesise that the use of swearwords constitutes an indicator of a profound emotional involvement of the programmer with the code and its inherent complexities, thus yielding better code based on a thorough, critical, and dialectic code analysis process.
- Caution: if you swear in source code it doesn’t automatically get better !!!

The Results



Alexis Stamatakis
@AlexisCompBio

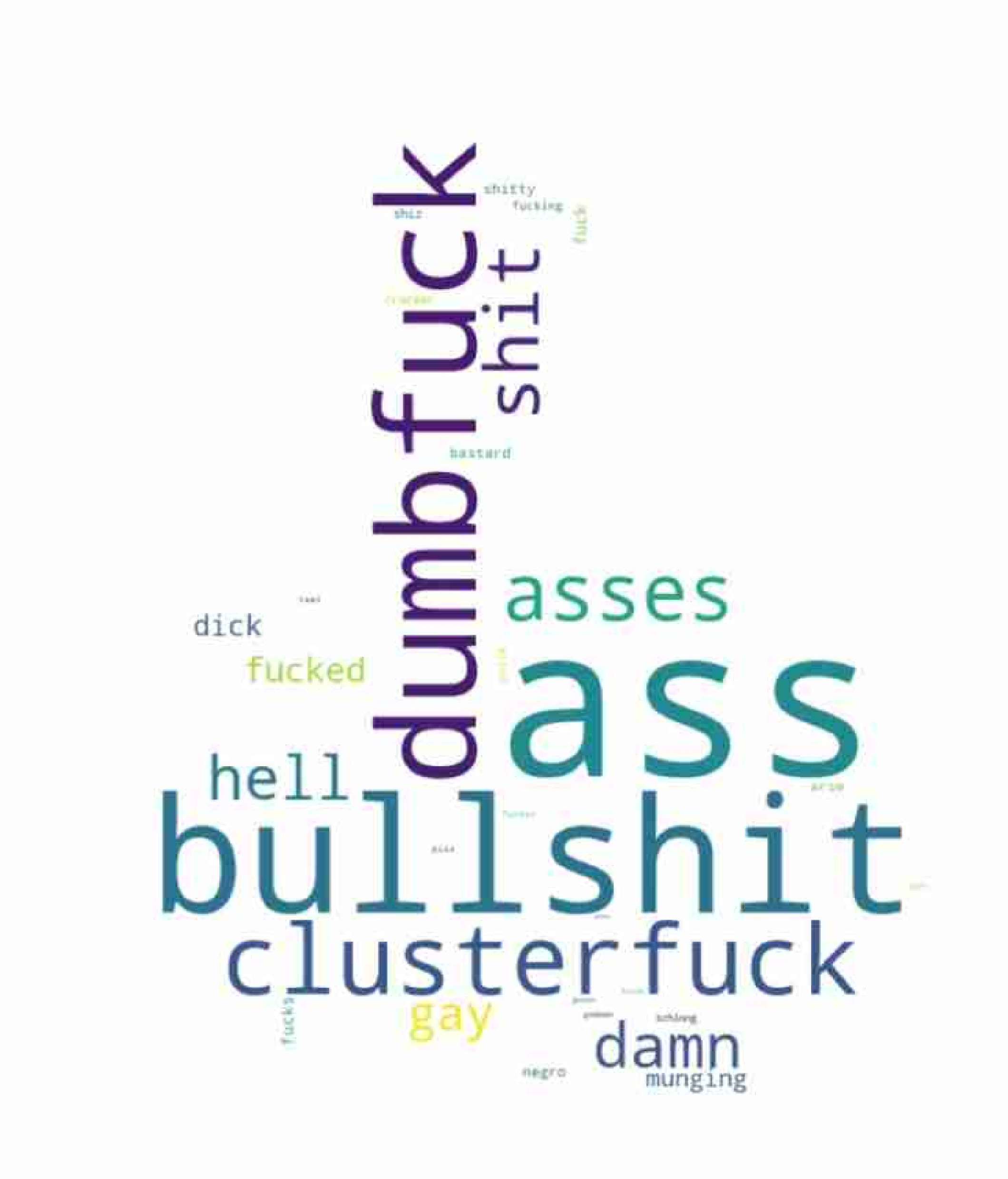
A Bachelor thesis in my lab makes a seminal contribution to software engineering - open source codes written in C on github have higher code quality when they contain swear words.

C source code without swear words
0
2
4
6
8
10
Code quality score
C source code with swear words
Proportion of repositories
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
Code quality score

6:56 AM · Feb 10, 2023 · **947.2K** Views

1,960 Retweets **291** Quote Tweets **11.2K** Likes

The Word Cloud
(10% of swear repos)



Thank you for your attention