# Introduction to Bioinformatics for Computer Scientists

## Lecture 14

# Plan for next lectures

- Today:
  - Questionnaire review
  - Random programming wisdom
- Next lecture: Wrap-up and exam preparation

# A review of the Questionnaire

- HPC
- Algorithms
- And some comments on programming in general

# HPC: some remarks

- HPC is nice
  - → better algorithms are nicer
  - → … and more energy-efficient
- Before parallelizing
  - → optimize the sequential code first
  - → frequently not done when comparing x86 code to some GPU or FPGA implementation
  - → learn the basics, before playing around with parallel HW

# Program Optimization

- *“The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet.”*
- *“Premature optimization is the root of all evil”*

# Why do we care about Code Quality?

- In bioinformatics software is potentially used by thousands of people
- It's not just proof of concept as frequently in CS
- Production level codes
  - Bugs & conceptual errors → thousands of papers could be wrong
  - Who maintains the software?
  - Who supports the software?

# Code Quality

- A lot of people don't care about code quality
- How can we quantify code quality?

  → difficult
- We can take some simple measures, e.g., in the programming practical, to improve code quality

# Compiler Warnings

- gcc → activate all warnings you can think of
  - Doesn't catch a lot of issues
  - Classic malloc mistake:

```
int a = 2000000, b = 3000000;
double c = (double *)malloc(a * b * sizeof(double));
```

- Clang compiler

  use `-Weverything -Wno-padded`

  → catches the above `malloc()` issue and many more issues

# Architecture Analyzers

- Tools to assess SW architecture quality

  → tell you which parts to redesign
- For instance, `pmccabe`: function complexity and line counting for C and C++
- See https://people.debian.org/~bame/pmccabe

# Use Assertions!

- Assertions

  `assert(logical op);`

  in C allow to partially implement some simple (and incomplete) form of Hoare-Logic
- Grep a code for assertions and see how many there are

  → good programmers use a lot of assertions!
- Assertions also facilitate debugging
- Users often provide incomplete bug reports, but, when an assertion fails, you directly know the line in the code that failed
- Good example for using assertions?

# Switch Statement in C

```
switch(a)
  {
    case 1:
      doSomtehing;
      break;
    case 2:
      doSomethingElse;
      break;
    default:
      assert(0);
  }
```

# Profiling

- Profiling: collecting statistics from example executions
  - Provides an idea which routines/functions are critical
  - Common profiling approaches:
    - Instrument manually/automatically all procedure call/return points (potentially expensive)
    - Sampling PC every X milliseconds -- so long as program run is significantly longer than the sampling period, the accuracy of profiling is pretty good
  - Profiling output

| Routine | % of Execution Time |
|---|---|
| function_a | 60% |
| function_b | 27% |
| function_c | 4% |
| ... | |
| function_zzz | 0.01% |

  - Often over 80% of the time spent in less than 20% of the code (80/20 rule)
    - Code locality principle
    - Data locality principle
  - More accurate profiling is possible with on-chip built-in HW counters and analysis tools

# Using Gprof

- GNU profiler
- Uses PC sampling technique
- Add the **`-pg`** flag to the compiling step!
- Add the **`-pg`** flag to the linking step!
- Then just run the program as before
  - It will execute slower though!
- Slowdown for dense matrix-matrix multiplication (**mmult**):
  - Less than 1%
- Then type `gprof mmult`

# Gprof on **mmult**

Flat profile:

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self ns/call | total ns/call | name |
|---|---|---|---|---|---|---|
| 99.83 | 156.90 | 156.90 | | | | main |
| 0.17 | 157.17 | 0.27 | 11184640 | 24.14 | 24.14 | randum |
| 0.00 | 157.17 | 0.00 | 18 | 0.00 | 0.00 | gettime |

% time — the percentage of the total running time of the program used by this function.

.....

# Using Valgrind

- Open source tool to find C/C++ memory leaks and a lot of other stuff!
- When compiling add **`-g`** to the compiler flags such that you can see the source lines where valgrind detects errors
- To look for memory leaks type:

  **`valgrind --tool=memcheck --leak-check=yes -v`**

  … and then just the executable as you called it so far
- Also works with PThreads-based programs
- Valgrind **slowdown** for **mmult almost factor 10 (188 -> 1451 secs)!**

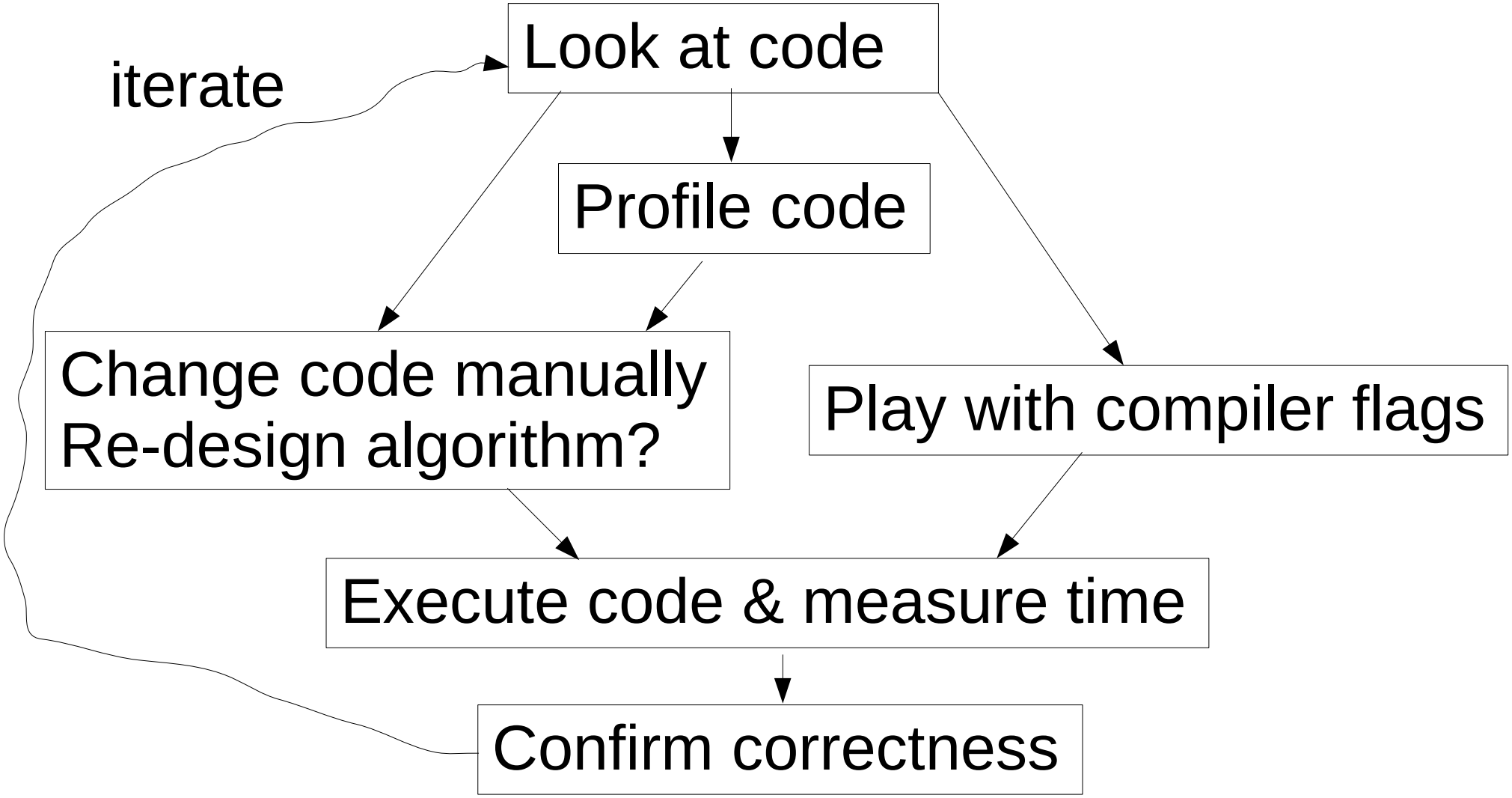
# Valgrind on **mmult**

```
Total time 1451.554931
--13304-- REDIR: 0x40cd4b0 (memset) redirected to 0x4023d50 (memset)
==13304==
==13304== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
--13304--
--13304-- supp:     13 dl-hack3-1
==13304== malloc/free: in use at exit: 0 bytes in 0 blocks.
==13304== malloc/free: 12,264 allocs, 12,264 frees, 134,264,640 bytes allocated.
```

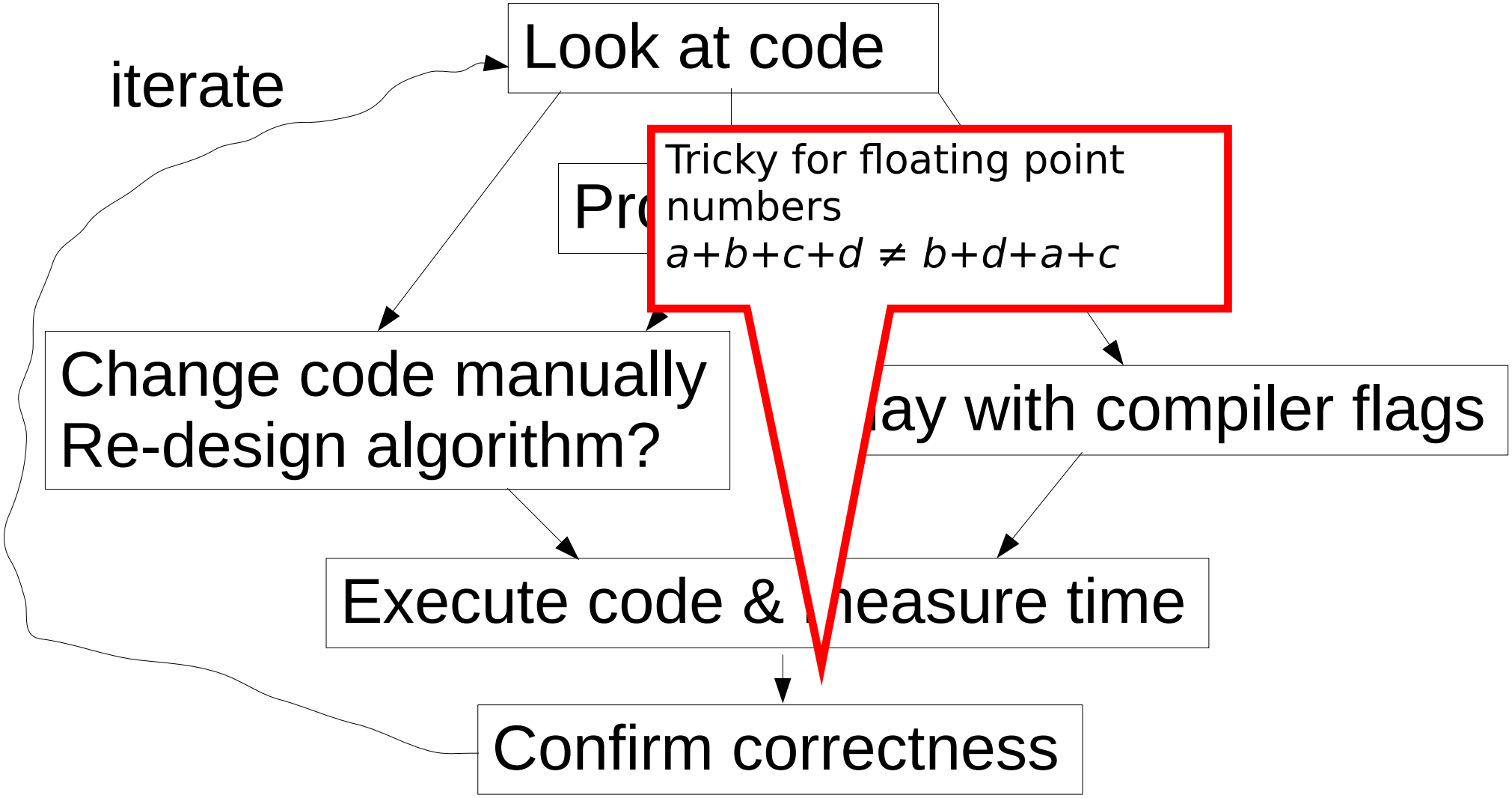
## Deliberate bug: c[matrixSize][matrixSize-1] = 0.0;

```
==14585== Invalid read of size 4
==14585==    at 0x80488AC: main (mmult.c:124)
==14585==  Address 0x41aa148 is 0 bytes after a block of size 64 alloc'd
==14585==    at 0x4022AB8: malloc (vg_replace_malloc.c:207)
==14585==    by 0x8048625: main (mmult.c:73)
==14585==
==14585== Invalid write of size 8
==14585==    at 0x80488BC: main (mmult.c:124)
==14585==  Address 0x78 is not stack'd, malloc'd or (recently) free'd
```

# Optimization Cycle



# Optimization Cycle



# BLAS: Basic Linear Algebra Subprograms

- Interface definition for linear algebra routines
- Several highly optimized implementations exist!
- Code optimization by using optimized libraries!
- Level 1 routines
  - Vector-vector ops
- Level 2 routines
  - Vector-matrix ops
- Level 3 routines
  - Matrix-matrix ops
- C-interface:
  - Functions preceded by **`cblas_`**, e.g., **`cblas_dgemm()`**
  - Arrays need to be contiguous in memory
  - Passed as pointers, **not** pointers of pointers

# BLAS

- Implementations
  - GOTO-BLAS → usually the fastest one
  - ATLAS-BLAS → easier to install
  - Intel MKL Math Kernel Library

    → also offers fast implementations of `exp()`, `log()` and other functions
  - AMD AMCL
- Interfaces: matrix linearization:

  → by rows or

  → by columns?
- ATLAS has a C interface

# De-Normalized Floating Point Numbers

- On-line demo of my benchmark
- Micro-Benchmark available at: https://github.com/stamatak/denormalizedFloatingPointNumbers

# De-Normalized Floating Point Numbers

- Internal representation > *64* bit, that deviates from IEEE 754 standard
- Allows for de-normalized floating point numbers to represent values near 0 → directly done by the hardware
- CPU issues floating point exception when using de-normalized floats

  → this exception delays CPU by several cycles!
- J. Björndalen and O. Anshus: "Trusting floating point benchmarks-are your benchmarks really data-independent?" http://www.hpc2n.umu.se/sites/default/files/events/para06/papers/paper_195.pdf

# Floating Point Numbers are the Root of all Evil!

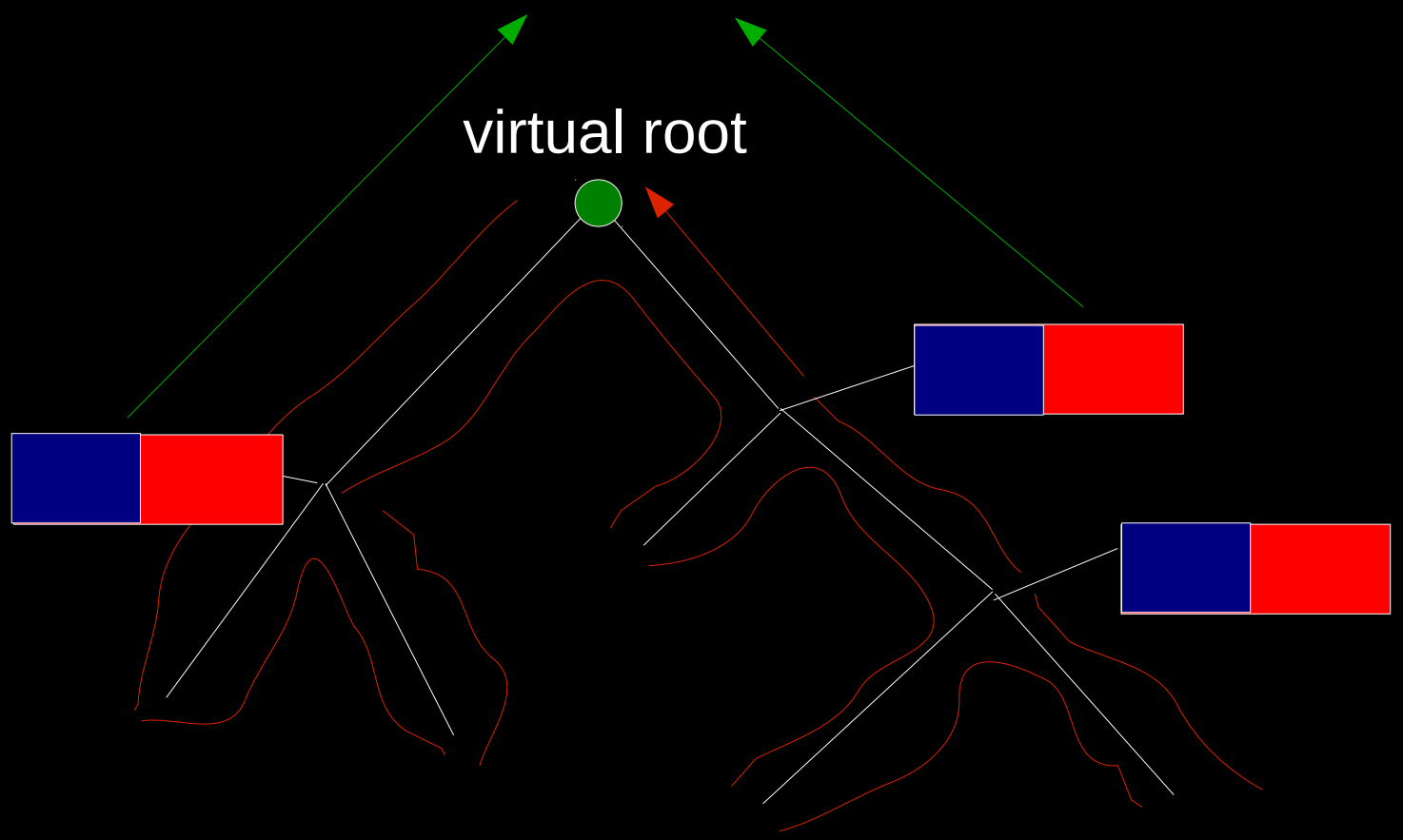
- Read the classic paper: *What Every Computer Scientist Should Know About Floating Point Arithmetic*
- Link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6768
- What can happen when we do parallel numerics?

# Floating Point Numbers are the Root of all Evil!

- Read the classic paper: *What Every Computer Scientist Should Know About Floating Point Arithmetic*
- Link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6768
- What can happen when we do parallel numerics?
- For instance if we do likelihood calculations and run the same code twice with 2 and 4 cores?
- What could happen?

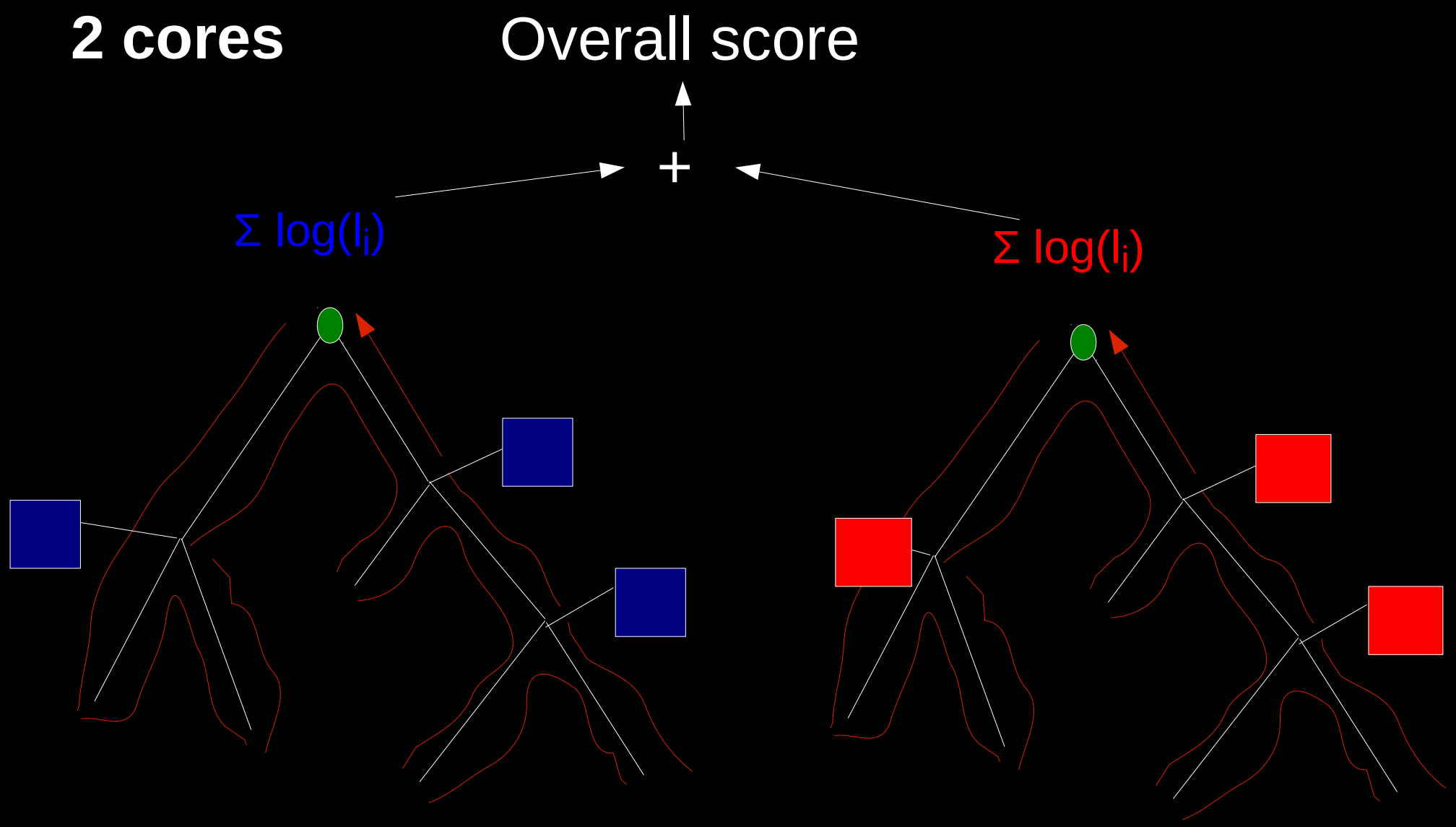
# An Example
## *Parallel Numerics*

$\Sigma \log(l_j) + \Sigma \log(l_i)$



# *Parallel Numerics*

**2 cores**

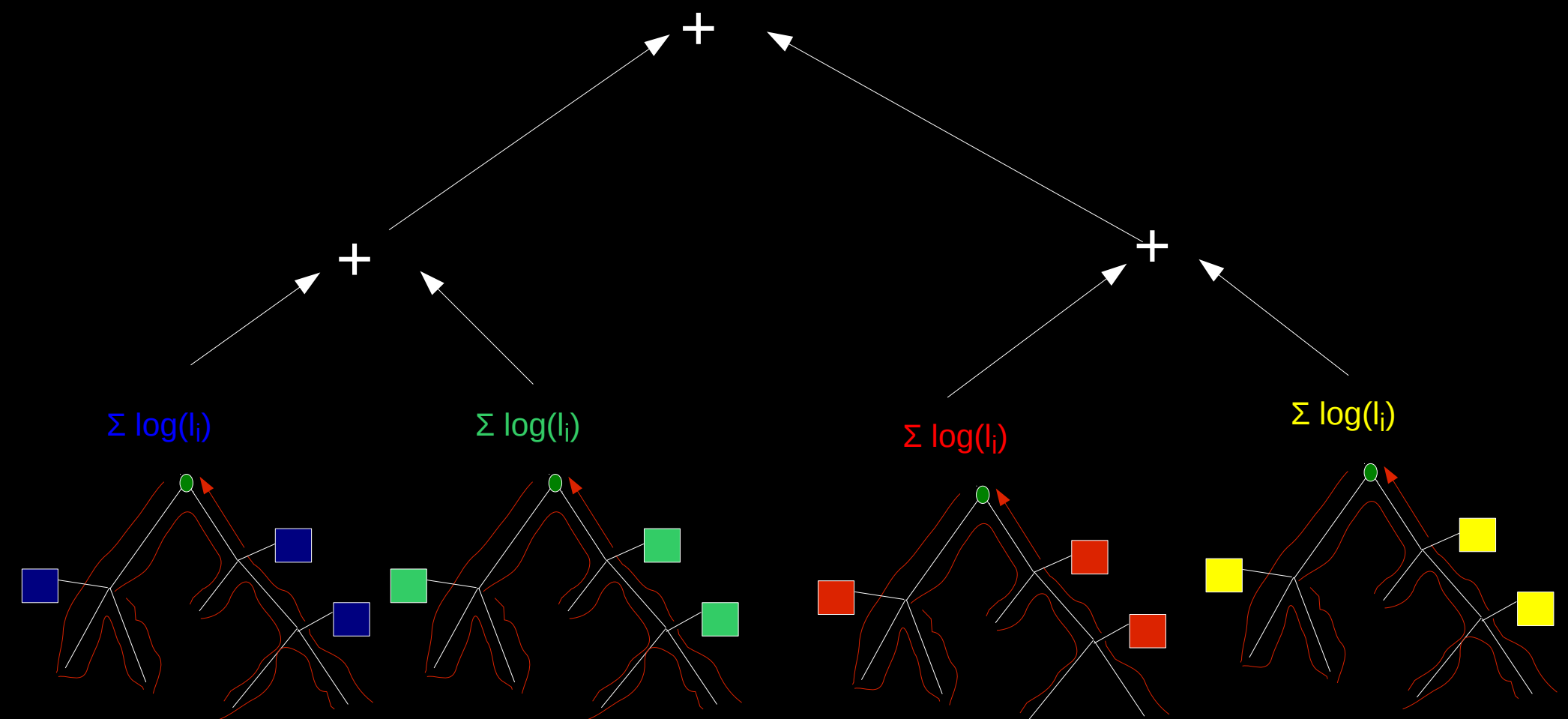


# *Parallel Numerics*

## 4 cores

Overall score

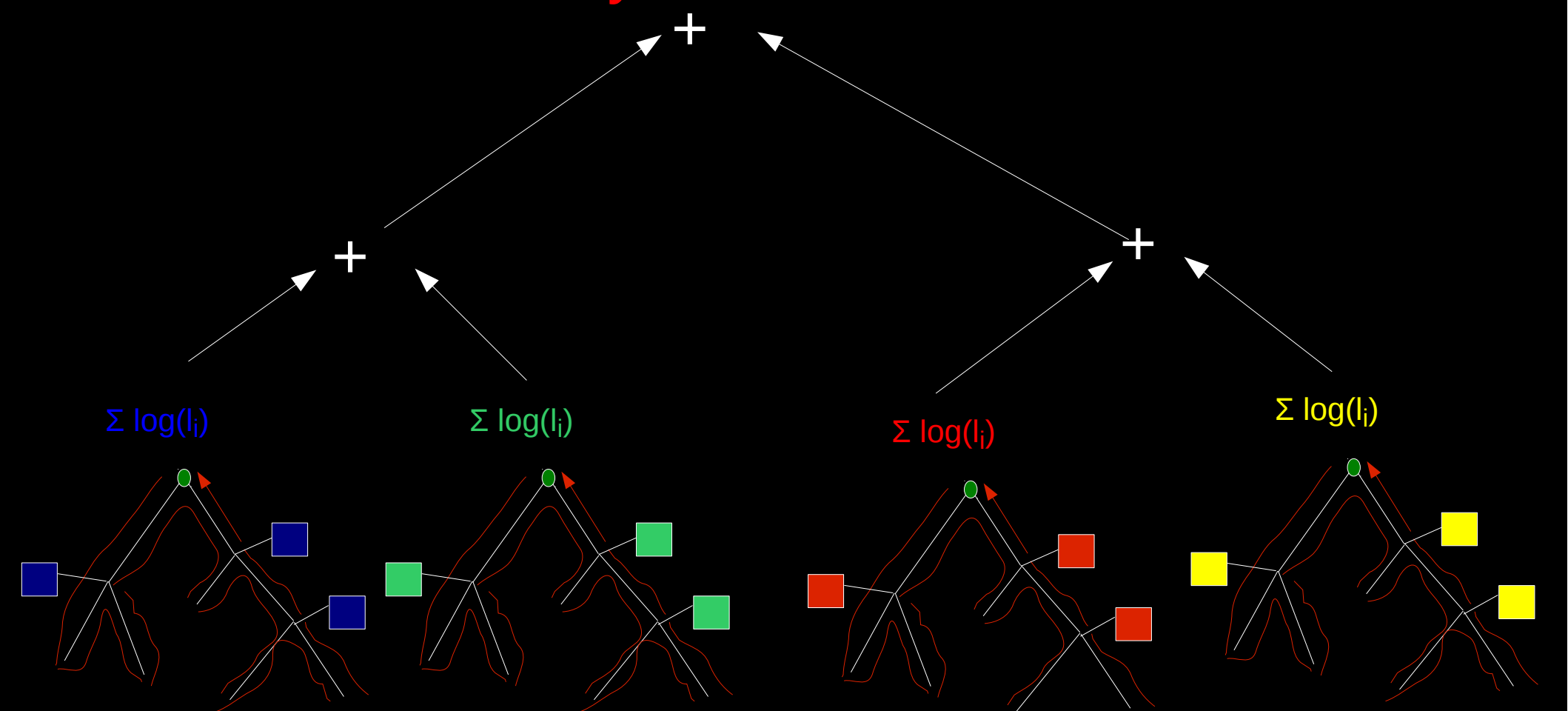
Can be different!



# *Parallel Numerics*

**4 cores**

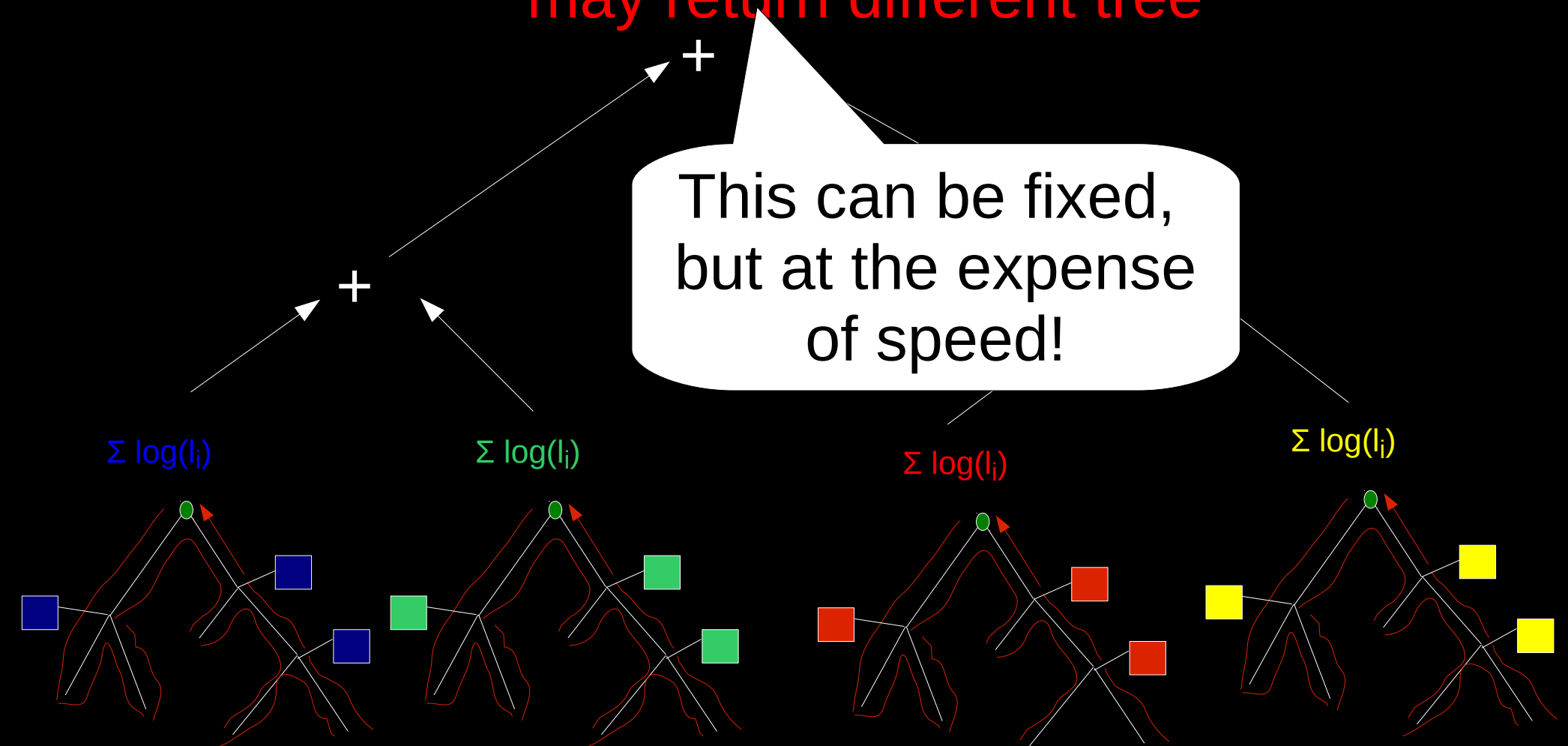
Overall score
Can be different →
may return different tree



# *Parallel Numerics*

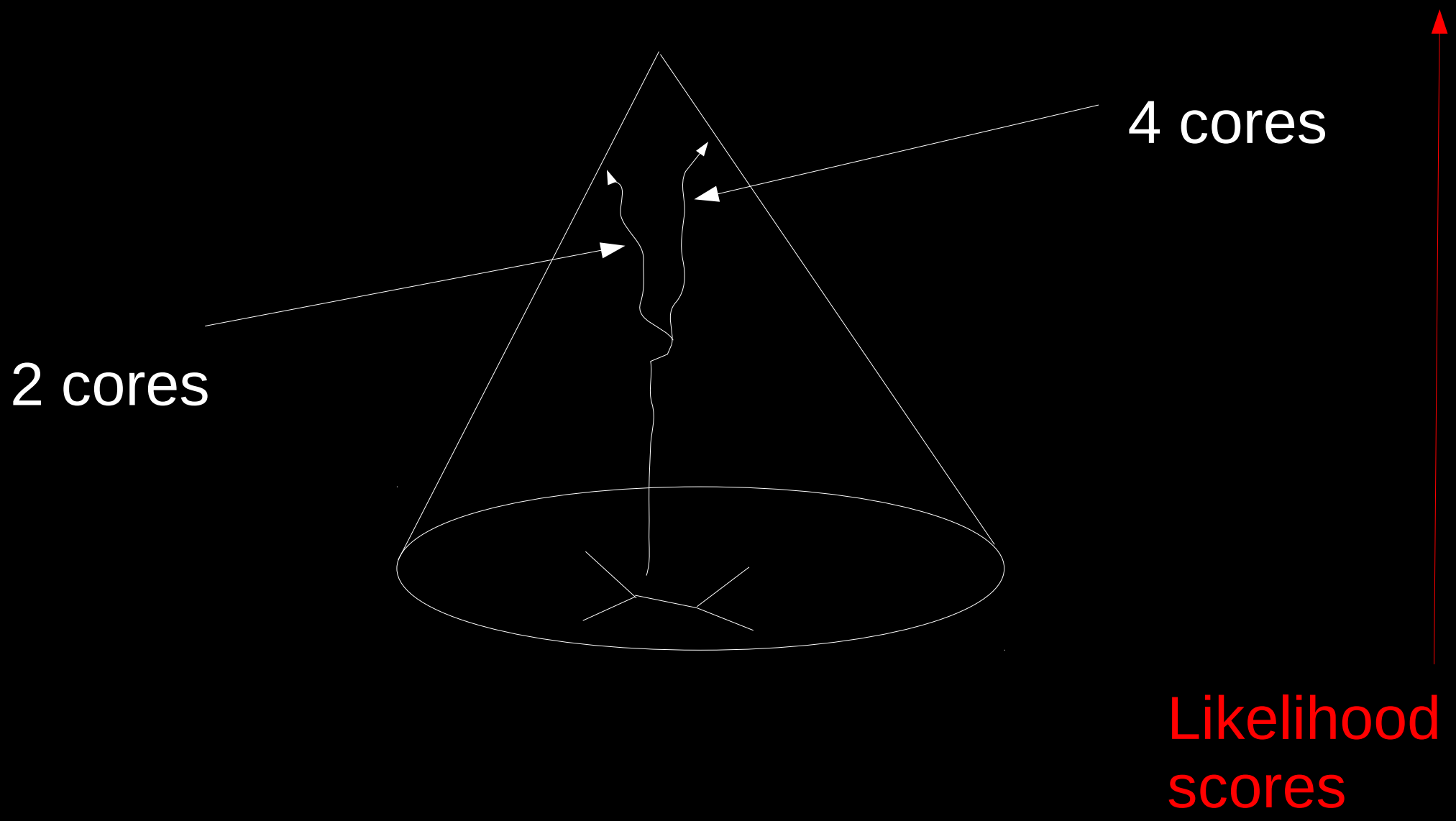
**4 cores**

Overall score
Can be different →
may return different tree

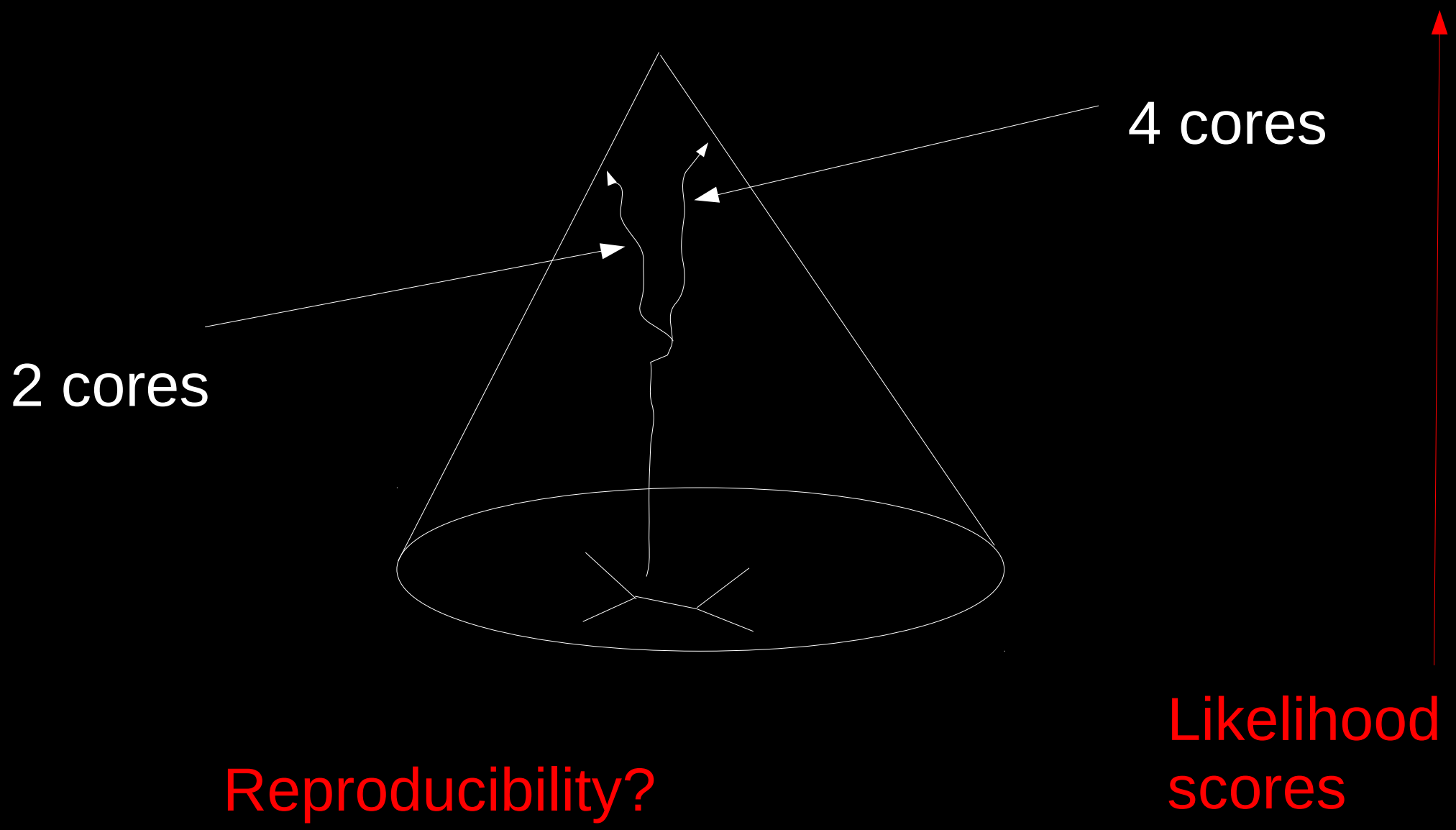



This can be fixed, but at the expense of speed!

# Diverging tree Searches



# Diverging tree Searches



# Instruction Level Parallelism

- All processors since around 1985 use pipelining to overlap (and hence parallelize and accelerate) the execution of instructions
- Goal: increase amount of parallelism and hence speed at a low (mostly HW) level
- Two distinct approaches:
  - HW-based & dynamic

    → RISC: **R**educed **I**nstruction **S**et **C**omputer/**C**omputing
  - SW (compiler)-based very long instruction words VLIW & static branch prediction

    → Intel Itanium processor
- ILP represents very fine-grained low-level parallelism
- When tuning performance you must keep in mind that there is a pipelined processor!

bottom-up

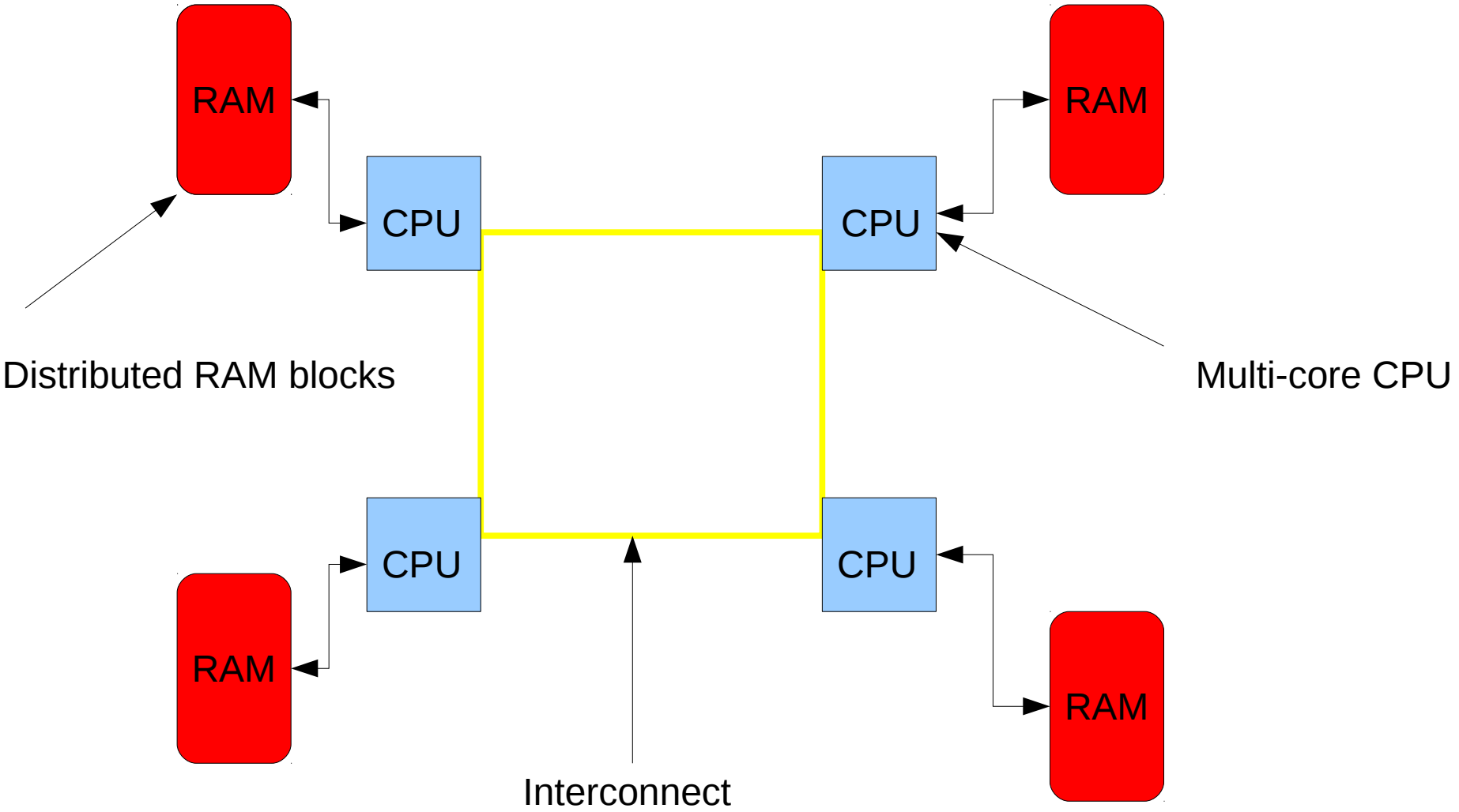
# Levels of Parallelism

- ILP in a single pipeline
- SSE & AVX vector instructions on a single core
  - → has anybody ever used vector intrinsics?
  - → compilers are supposed to vectorize automatically
  - → often this doesn't work
  - → our experience: plain, good C code versus manually vectorized code: speedup factor 4.5 (with 256 bit AVX instructions)
- Simultaneous multi-threading/Hyper-threading
- Thread-based (OpenMP/Pthreads) on shared memory
  - → multi-cores
  - → mostly
    - loop-level parallelism
    - Fork-join paradigm
  - → shared memory supercomputers
- MPI across nodes
  - → high bandwidth/low latency network interconnect
  - → current state-of-the art: Infiniband technology
  - → closely coupled parallel codes
- Distributed/Cloud computing, e.g., seti@home → no need for fast interconnect

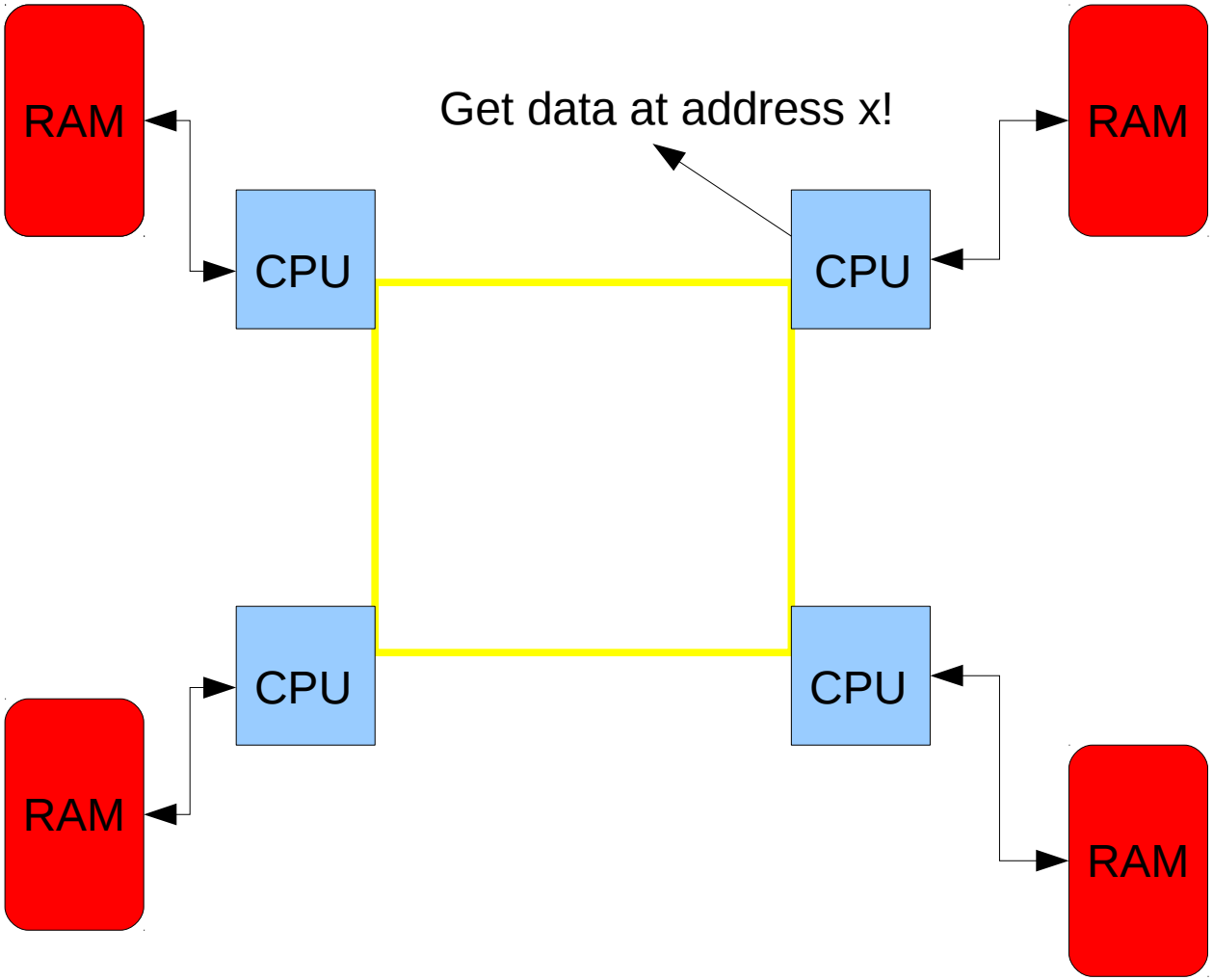
# NUMA

- Taxonomy of shared memory computers based on RAM address access times
- UMA: Uniform Memory Access
- NUMA: Non-Uniform Memory Access
  - → all multi-cores
  - → we actually have distributed shared memory!

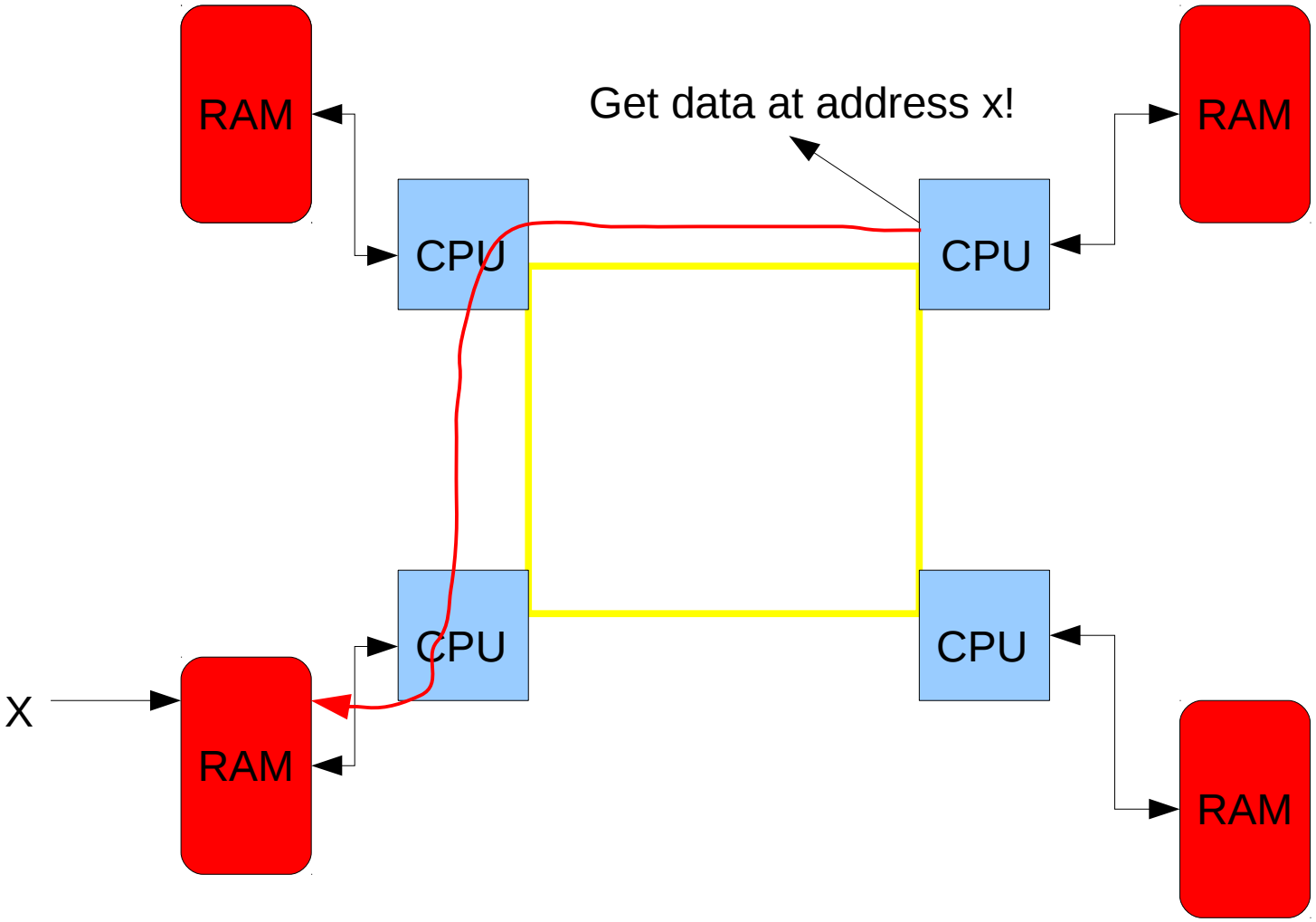
# NUMA


RAM
RAM
CPU
CPU
Distributed RAM blocks
Multi-core CPU
CPU
CPU
RAM
RAM
Interconnect

# NUMA



# NUMA



# NUMA

- We need to think about data locality when programming on a NUMA architecture!
- What happens when a thread on a NUMA system allocates memory?

# NUMA

- We need to think about data locality when programming on a NUMA architecture!
- What happens when a thread on a NUMA system allocates memory?

  → first touch policy!
- Page gets allocated to RAM block closest to the first core that does a writing access

  → there also exist performance issues associated to *thread-to-core* pinning

  → can you imagine what could be the problem here

# Top500 list

- List of top 500 supercomputer systems in the world
- Benchmarking based on LINPACK linear algebra package

| RANK | SITE | SYSTEM | CORES | RMAX (TFLOP/S) | RPEAK (TFLOP/S) | POWER (KW) |
|---|---|---|---|---|---|---|
| 1 | National Super Computer Center in Guangzhou<br>China | **Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P<br>NUDT | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 2 | DOE/SC/Oak Ridge National Laboratory<br>United States | **Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x<br>Cray Inc. | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 3 | DOE/NNSA/LLNL<br>United States | **Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom<br>IBM | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 4 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect<br>Fujitsu | 705,024 | 10,510.0 | 11,280.4 | 12,660 |
| 5 | DOE/SC/Argonne National Laboratory<br>United States | **Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM | 786,432 | 8,586.6 | 10,066.3 | 3,945 |
| 6 | Swiss National Supercomputing Centre (CSCS)<br>Switzerland | **Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x<br>Cray Inc. | 115,984 | 6,271.0 | 7,788.9 | 2,325 |
| 7 | Texas Advanced Computing Center/Univ. of Texas<br>United States | **Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P<br>Dell | 462,462 | 5,168.1 | 8,520.1 | 4,510 |
| 8 | Forschungszentrum Juelich (FZJ)<br>Germany | **JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect<br>IBM | 458,752 | 5,008.9 | 5,872.0 | 2,301 |
| 9 | DOE/NNSA/LLNL<br>United States | **Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect<br>IBM | 393,216 | 4,293.3 | 5,033.2 | 1,972 |
| 10 | Government<br>United States | Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40<br>Cray Inc. | 72,800 | 3,577.0 | 6,131.8 | 1,499 |
| 11 | NASA/Ames Research Center/NAS<br>United States | **Pleiades** - SGI ICE X, Intel Xeon E5-2670/E5-2680v2/E5-2680v3 2.6/2.8/2.5 GHz, Infiniband FDR<br>SGI | 160,768 | 3,375.6 | 3,987.9 | 3,141 |
| 12 | Exploration & Production - Eni S.p.A.<br>Italy | **HPC2** - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x<br>IBM | 72,000 | 3,188.0 | 4,605.0 | 1,227 |
| 13 | Government<br>United States | Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect<br>Cray Inc. | 225,984 | 3,143.5 | 4,881.3 | |
| 14 | Leibniz Rechenzentrum<br>Germany | **SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR<br>IBM | 147,456 | 2,897.0 | 3,185.1 | 3,423 |

# Top500 list

- List of top 500 supercomputer systems in the world
- Benchmarking based on LINPACK linear algebra package

Who knows what weak scaling is?

| RANK | SITE | | CORES | RMAX (TFLOP/S) | RPEAK (TFLOP/S) | POWER (KW) |
|---|---|---|---|---|---|---|
| 1 | National Super Computer Center in Guangzhou<br>China | **lkyWay-2)** - TH-IVB-FEP<br>eon E5-2692 12C 2.200GHz,<br>el Xeon Phi 31S1P | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| | | | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| | | C | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 4 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect<br>Fujitsu | 705,024 | 10,510.0 | 11,280.4 | 12,660 |
| 5 | DOE/SC/Argonne National Laboratory<br>United States | **Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom<br>IBM | 786,432 | 8,586.6 | 10,066.3 | 3,945 |
| 6 | Swiss National Supercomputing Centre (CSCS)<br>Switzerland | **Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x<br>Cray Inc. | 115,984 | 6,271.0 | 7,788.9 | 2,325 |
| 7 | Texas Advanced Computing Center/Univ. of Texas<br>United States | **Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P<br>Dell | 462,462 | 5,168.1 | 8,520.1 | 4,510 |
| 8 | Forschungszentrum Juelich (FZJ)<br>Germany | **JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect<br>IBM | 458,752 | 5,008.9 | 5,872.0 | 2,301 |
| 9 | DOE/NNSA/LLNL<br>United States | **Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect<br>IBM | 393,216 | 4,293.3 | 5,033.2 | 1,972 |
| 10 | Government<br>United States | Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40<br>Cray Inc. | 72,800 | 3,577.0 | 6,131.8 | 1,499 |
| 11 | NASA/Ames Research Center/NAS<br>United States | **Pleiades** - SGI ICE X, Intel Xeon E5-2670/E5-2680v2/E5-2680v3 2.6/2.8/2.5 GHz, Infiniband FDR<br>SGI | 160,768 | 3,375.6 | 3,987.9 | 3,141 |
| 12 | Exploration & Production - Eni S.p.A.<br>Italy | **HPC2** - iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, NVIDIA K20x<br>IBM | 72,000 | 3,188.0 | 4,605.0 | 1,227 |
| 13 | Government<br>United States | Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect<br>Cray Inc. | 225,984 | 3,143.5 | 4,881.3 | |
| 14 | Leibniz Rechenzentrum<br>Germany | **SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR<br>IBM | 147,456 | 2,897.0 | 3,185.1 | 3,423 |

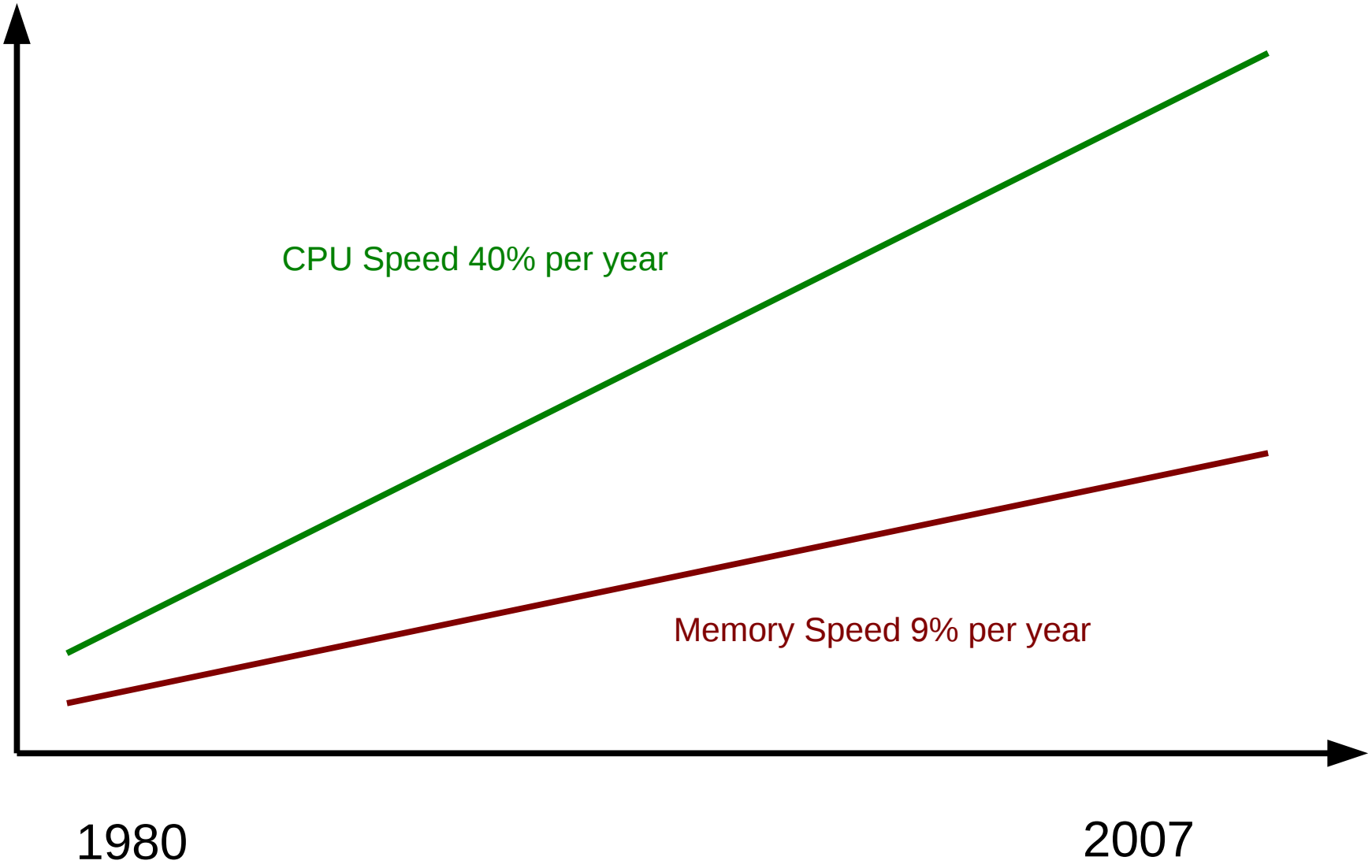
# OpenMP

- **Open** specifications for **M**ulti-**P**rocessing
- Easy semi-automatic parallelization of codes for shared memory systems
- Just insert so-called **pragmas** into the code to parallelize loops (mostly)
- Based upon PThreads
- Needs a dedicated OpenMP-enabled compiler
- In comparison to direct usage of PThreads
  - faster implementation
  - provides the programmer less control over what is happening!

# OpenMP Hello World

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int id,i;
    omp_set_num_threads(4);


#pragma omp parallel for private(id)
    for (i = 0; i < 4; ++i)
    {
        id = omp_get_thread_num();

        printf("Hello World from thread %d\n", id);
    }

    return 0;
}
```

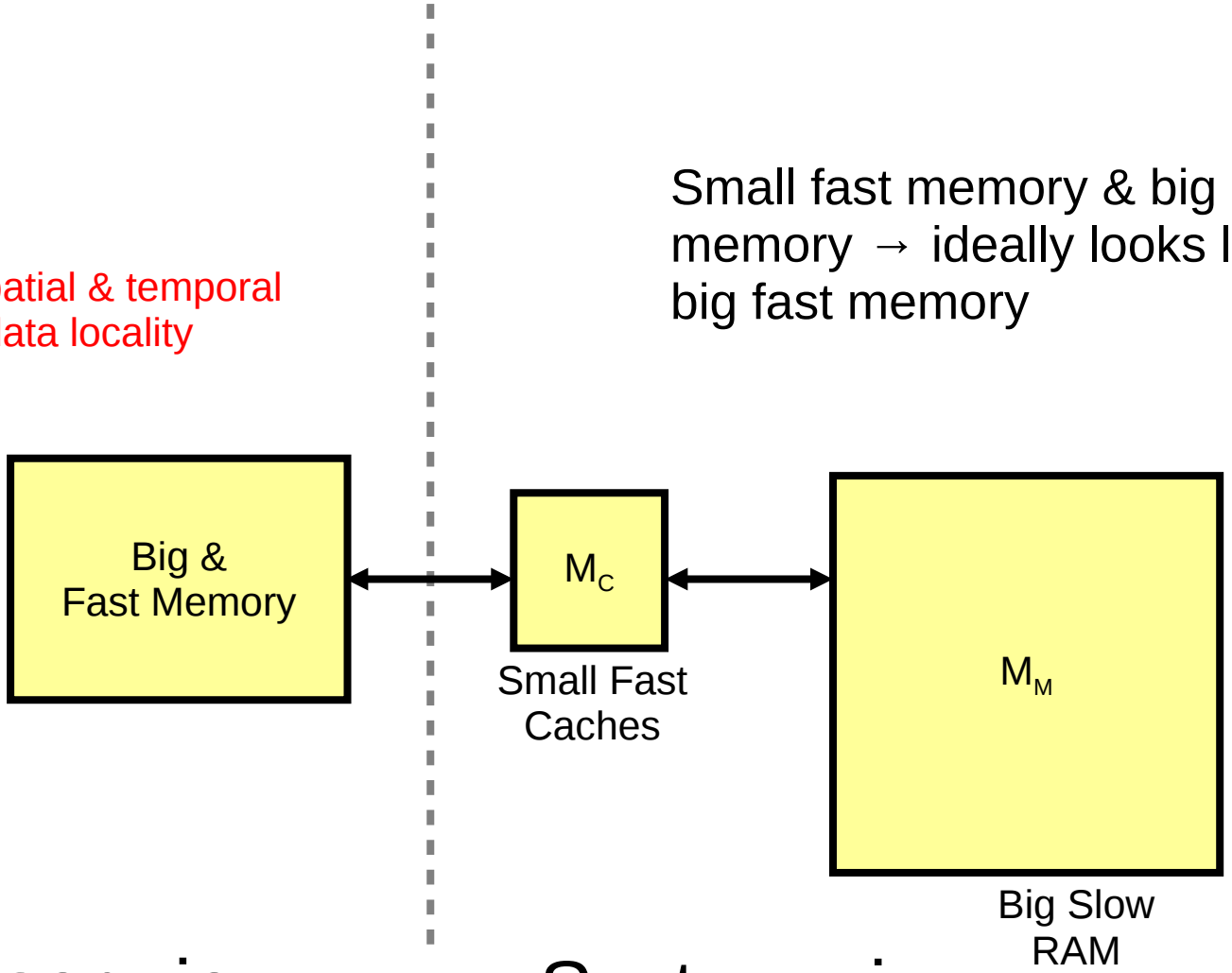
# The Memory Gap



# What we hope for

Given spatial & temporal code & data locality

Small fast memory & big slow memory → ideally looks like a big fast memory




User view

System view

# Principles of Cache Memories & Paging Algorithms

- What I wanted to hear:
  - Temporal locality of data accesses
  - Spatial locality of data accesses
- The term cache comes from the French verb *cacher* (hiding)

  → Caches hide the memory latency
- If temporal and spatial data locality are not given, caches have no effect

  → random memory access patterns

  → e.g., accessing hash tables!

# Cache Coherence

- In a shared memory system with multiple caches
- How do we make sure that copies of the same address *x* in RAM that may reside in two or more caches are consistent when one cache copy is changed?

  → cache coherency protocols
- We often talk about *ccNUMA*

  → cache-coherent NUMA

# Measuring Parallel Performance

- *Speedup*: S = T(1)/T(p)
- It can't be stated frequently enough

  → T(1) actually refers to the fastest sequential implementation/algorithm available, everything else should be reported as **relative** speedup!

- *Scalability*: Up to how many CPUs do we get good speedups?
- We distinguish between:

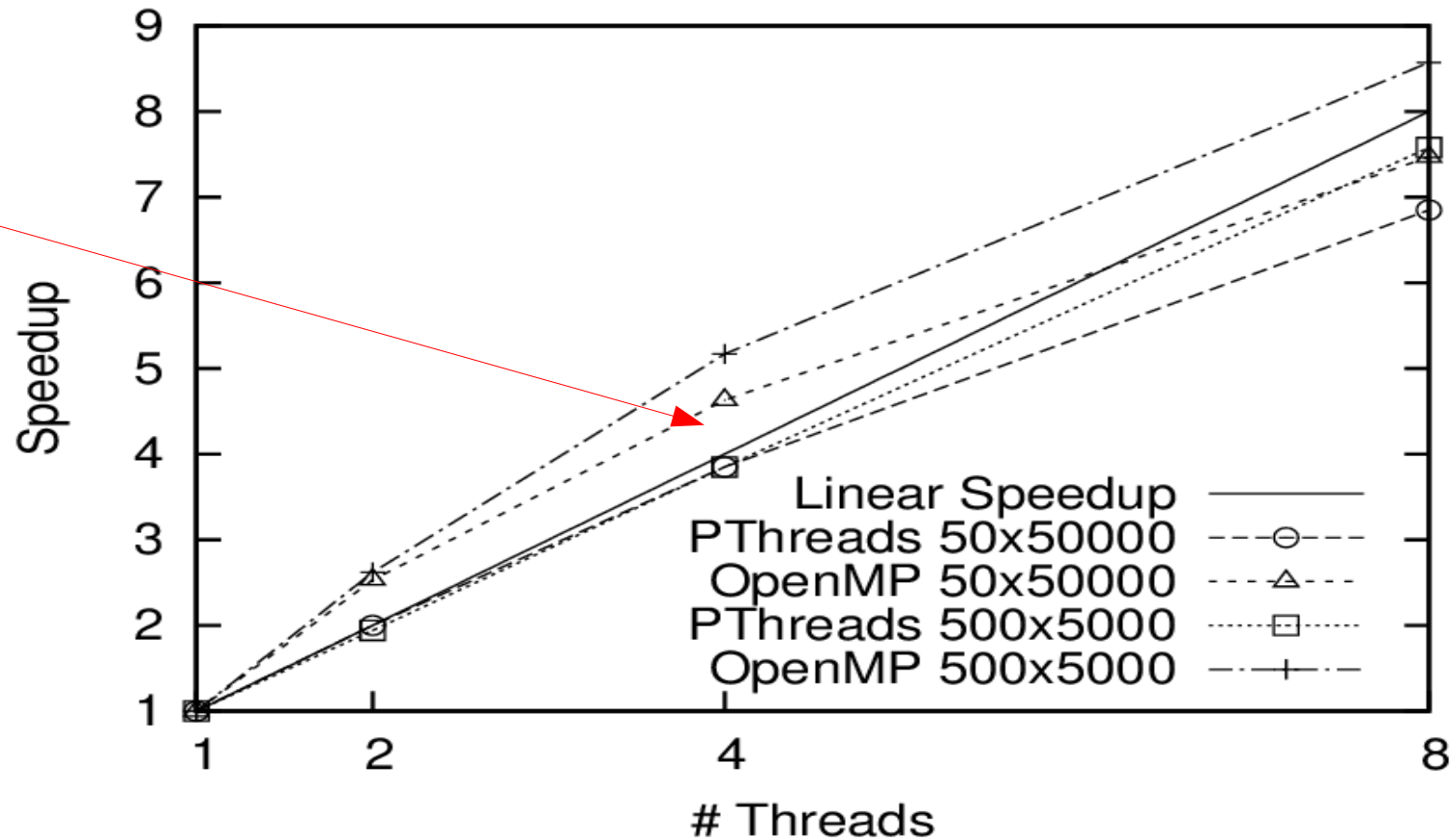
  → *weak scaling*: scale up the problem size as we add cores

  → *strong scaling*: keep problem size fixed as we add cores

# Super-linear speedups

- Reducing the per-CPU memory footprint of parallel programs via data distribution can yield super-linear speedups due to increased cache efficiency due to increased total cache size
- Below: RAxML on an AMD Barcelona multi-core system:

Super-linear speedups



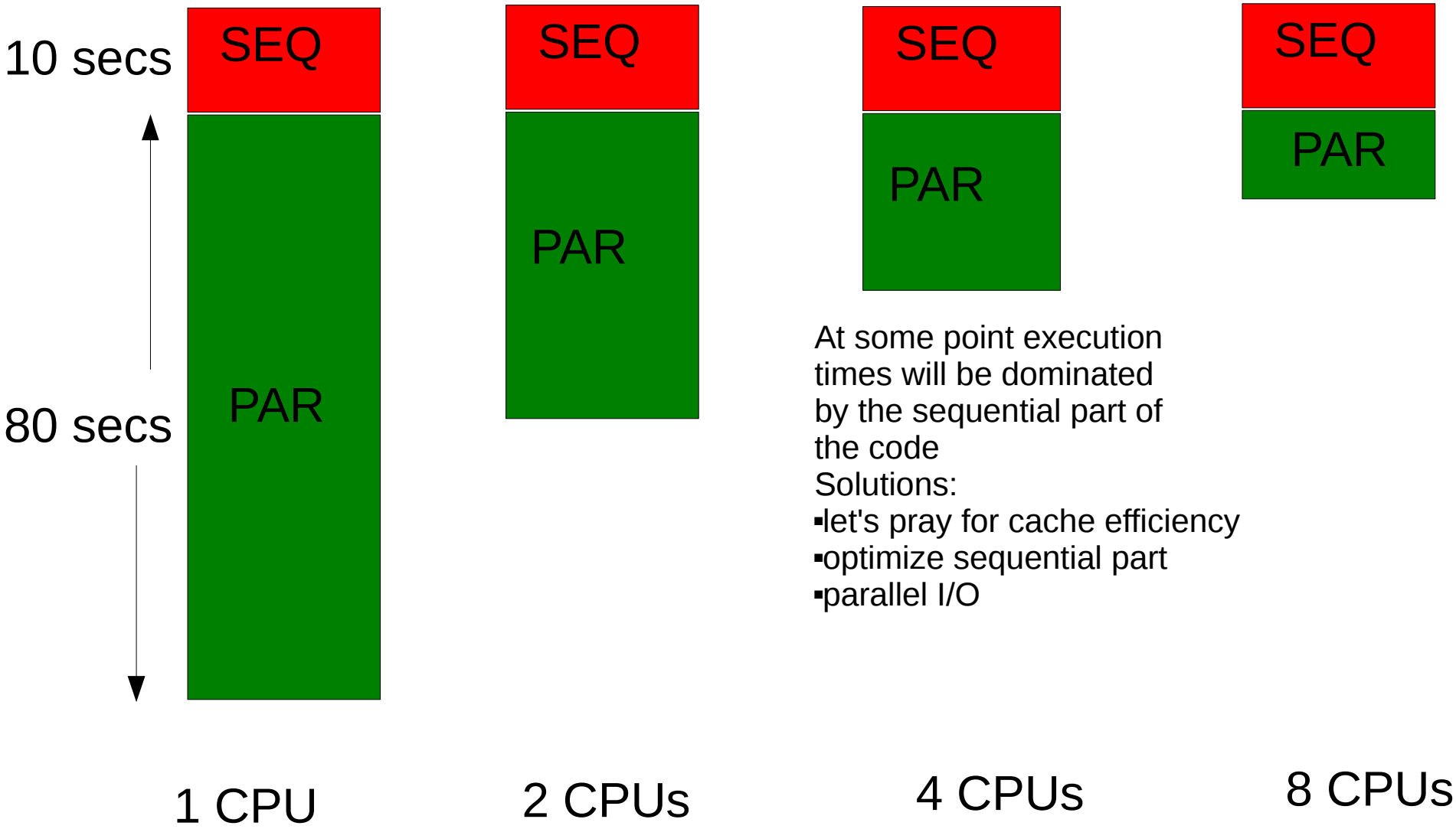
# Example

- Sequential code total memory footprint of 20 MB
- Assume a machine with 2 cores and two caches of 10MB each

  → if we use one core, the working set of 20MB will not fit into cache

  → if we use two cores, the working set of 20 MB will entirely fit into the two caches

# Amdahl's Law




At some point execution times will be dominated by the sequential part of the code

Solutions:

- let's pray for cache efficiency
- optimize sequential part
- parallel I/O

# Amdahl's Law

- Scalability to large number of processors is limited by sequential part of program
- Every program has a sequential portion, even if it is just the time needed to start all the threads or MPI processes!
- More formally:

  $speedup \leq 1 /( f + ((1-f)/p)))$

  where $f$ is the fraction $f$ with $0 \leq f \leq 1.0$ of the program that must be executed sequentially
- Thus for $p \rightarrow infinity$ the maximum speedup $S_{max} \leq 1/f$
- If the $f:=0.01 \rightarrow S_{max} \leq 100$ :-(
- IMPORTANT: here we are assuming linear speedups for the part that *can* be parallelized!
- Solutions
  - Hope that $f$ is small
  - Make $f$ small
  - May be counter-balanced by cache-efficiency!

# Amdahl's law

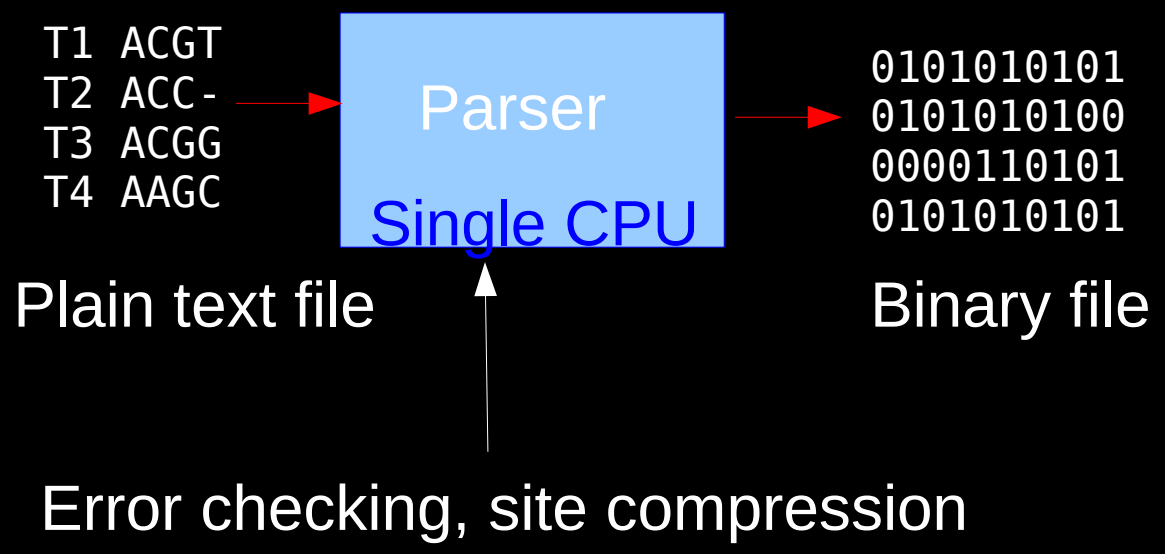
- This is a real issue
- Encountered it while developing our codes

  ExaBayes and ExaML
- Reduced parallel I/O times at start-up from 15 to 1 minute on about 4000 cores!

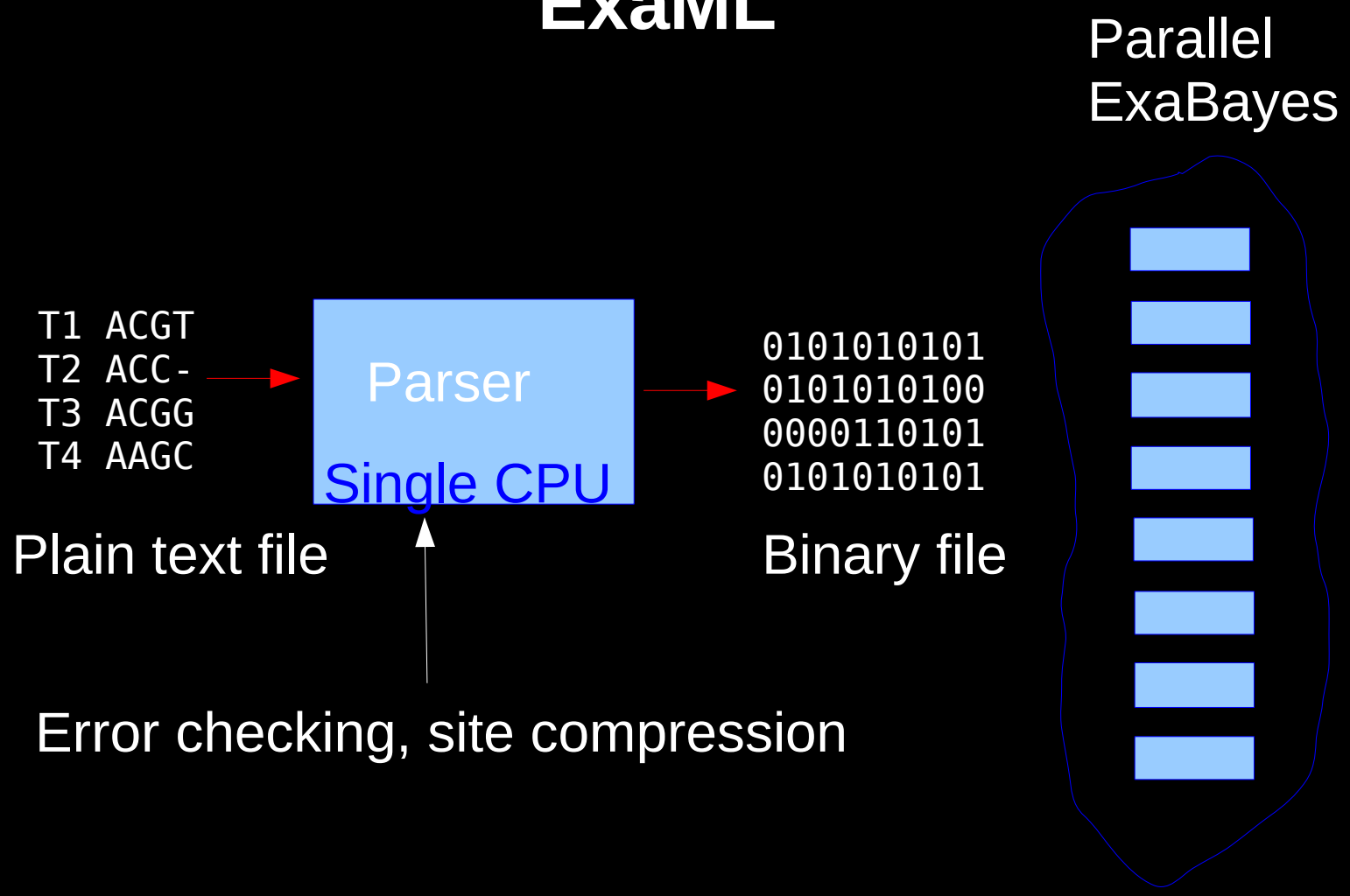
# ExaBayes and ExaML

- Exascale Bayesian and Maximum Likelihood Phylogenetic Inference
- Codes for phylogenetic inference on supercomputers
- Scale up to 32,000 cores on SuperMUC x86 system
- Analyze datasets with 200 taxa and 100,000,000 sites

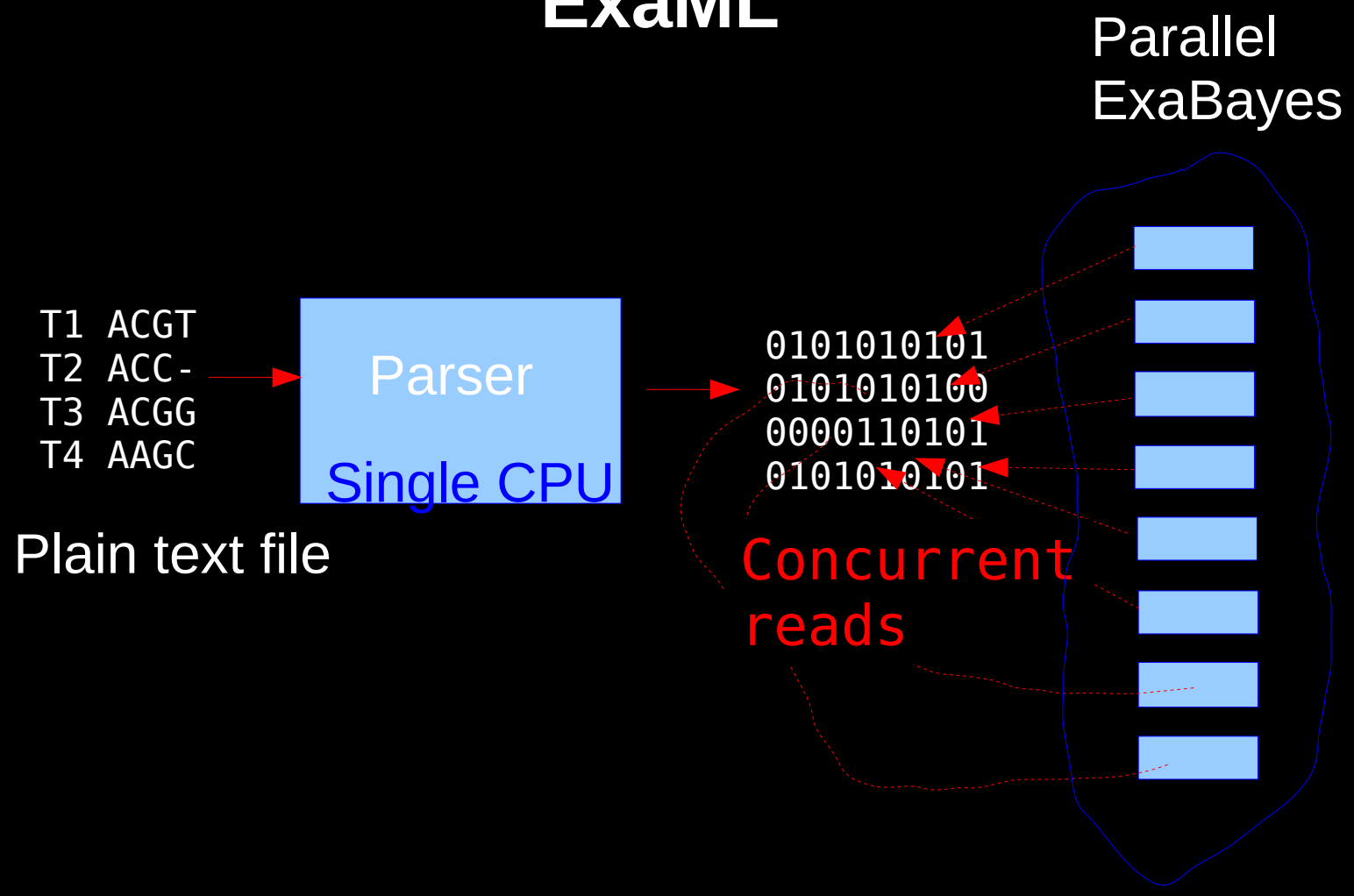
# Handling I/O in ExaBayes and ExaML



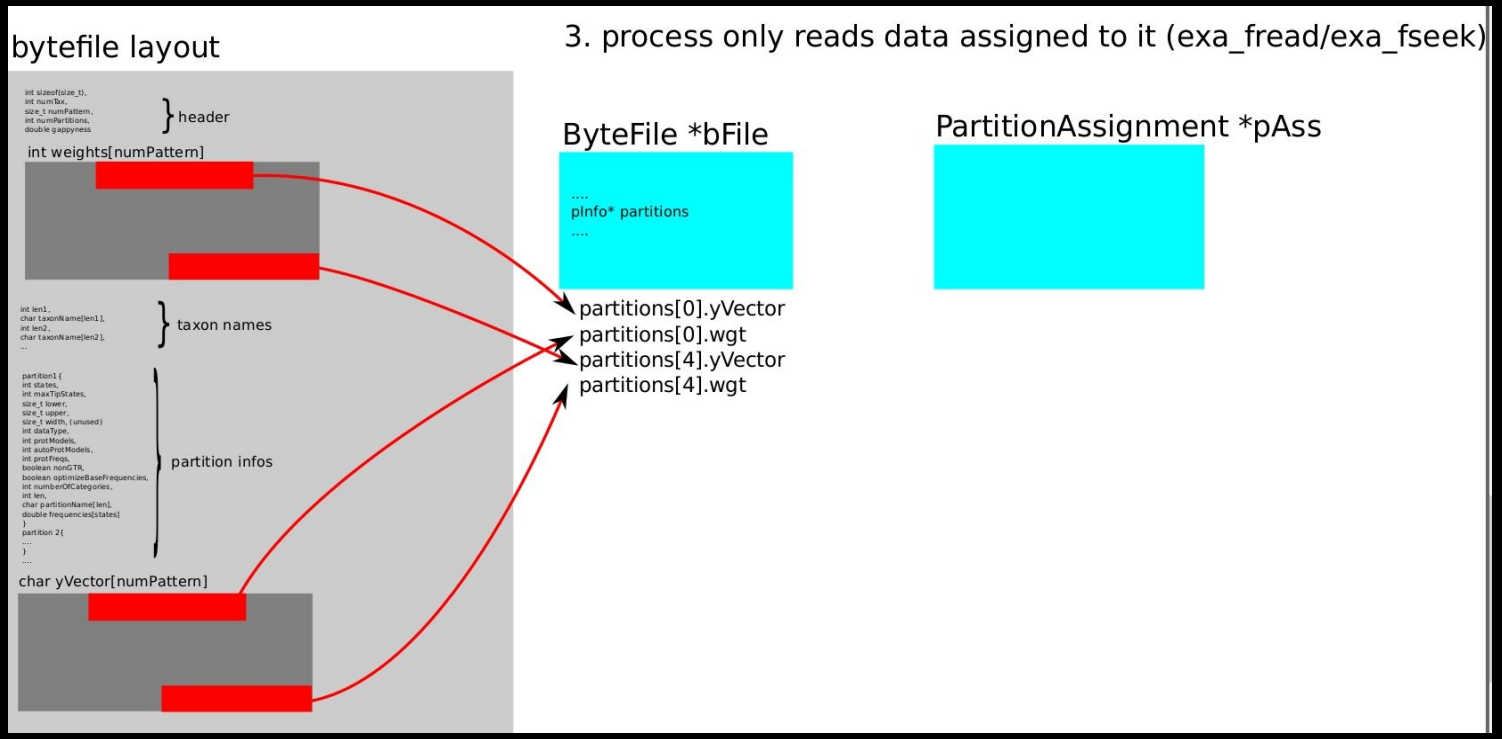
# Handling I/O in ExaBayes and ExaML



# Handling I/O in ExaBayes and ExaML



# Concurrent I/O: The index war



# Twelve Ways to Fool the Masses when Giving Performance Results on Parallel Computers
by David H. Bailey

**Who cares about Amdahl's law anyway:**

2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

It is quite difficult to obtain high performance on a complete large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution to this dilemma is to present results for an inner kernel of an application, which can be souped up with artificial tricks. Then imply in your presentation that these rates are equivalent to the overall performance of the entire application.

# Twelve Ways to Fool the Masses when Giving Performance Results on Parallel Computers
by David H. Bailey

**Who cares about Amdahl's law anyway:**

2. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.

It is quite difficult to obtain high performance on a complete large-scale scientific application, timed from beginning of execution through completion. There is often a great deal of data movement and initialization that depresses overall performance rates. A good solution to this dilem is to present results for an inner kernel of an application, whic e souped up with artificial tricks. Then imply in your prese these rates are equivalent to the overall performan application.

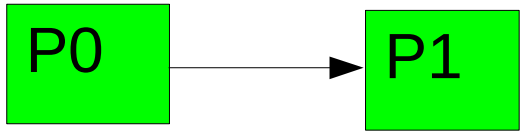
Full list at:
http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf

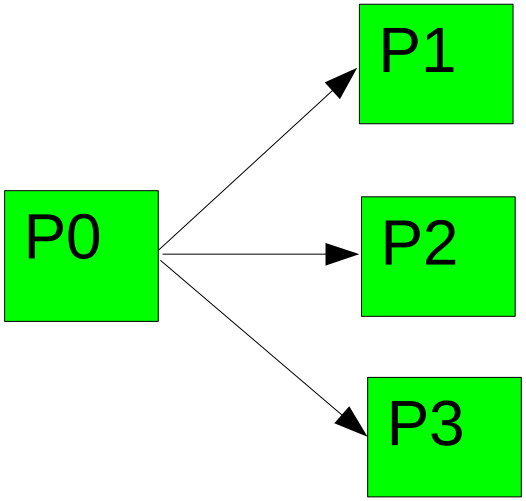
# MPI API for parallelizing Code on **Distributed Memory Architectures**

- **M**essage **P**assing **I**nterface
- Send-Receive Paradigm
- Point to Point Communication
- Collective Communication

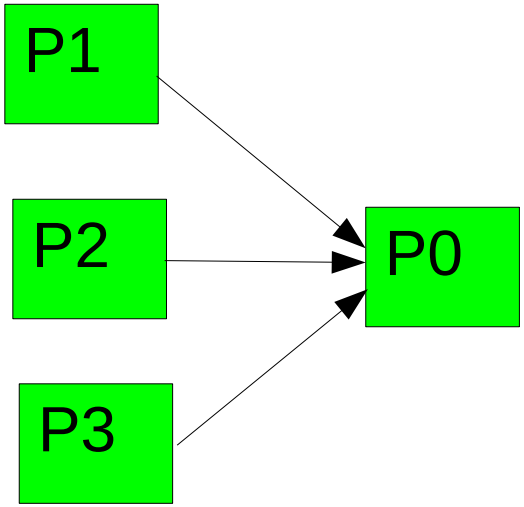
# Communication Schemes

Direct/point-to-point




broadcast/multicast




indirect/reduction

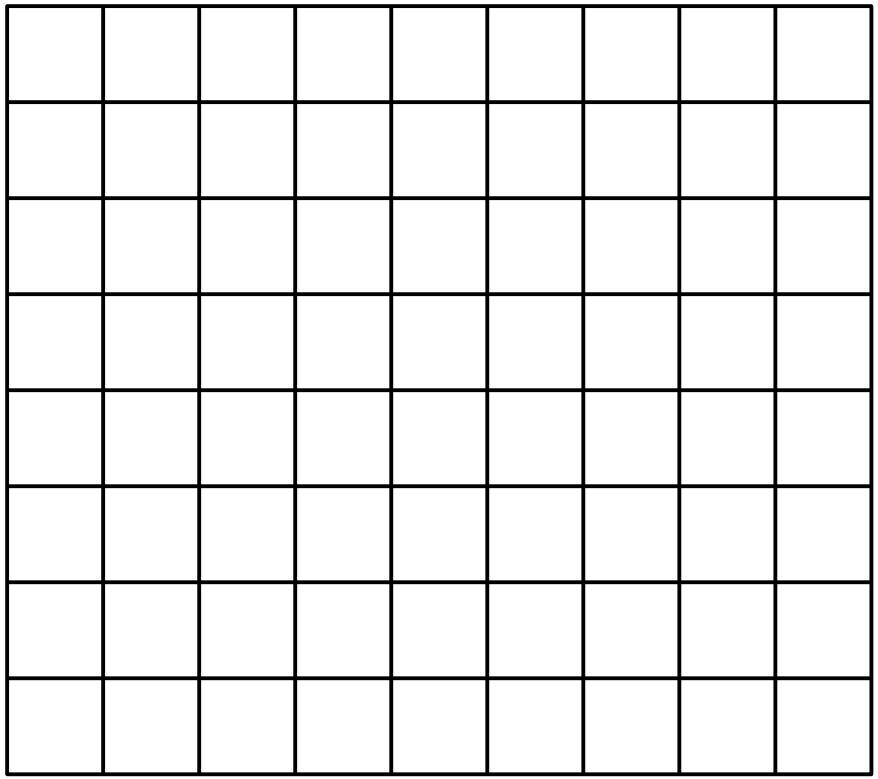
# Classic MPI Programming Error

```
MPI_Init (&argc, &argv);

/* starts MPI */

MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* get current process id */

MPI_Comm_size (MPI_COMM_WORLD, &size);

/* get number of processes */
```

- Those three commands must be the very first at the beginning of `main()` !

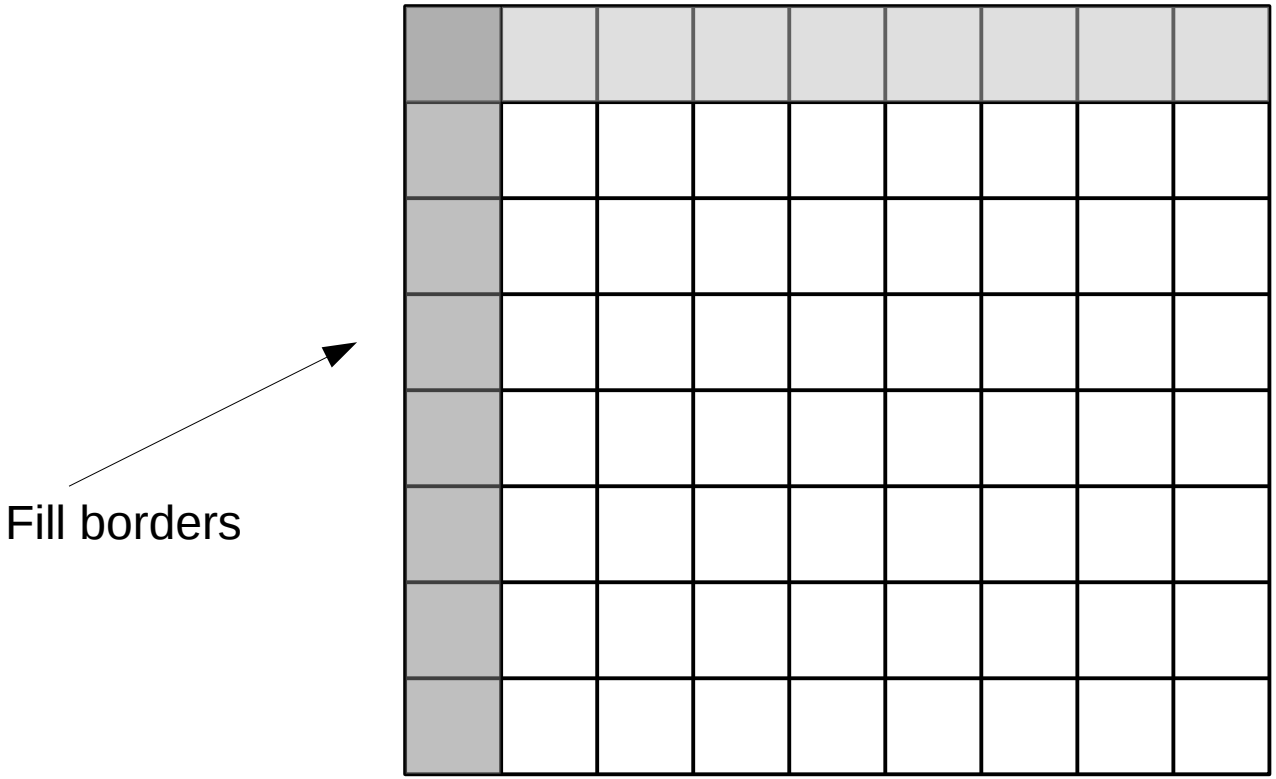
# Wave-Front Parallelism

- Dynamic Programming algorithms



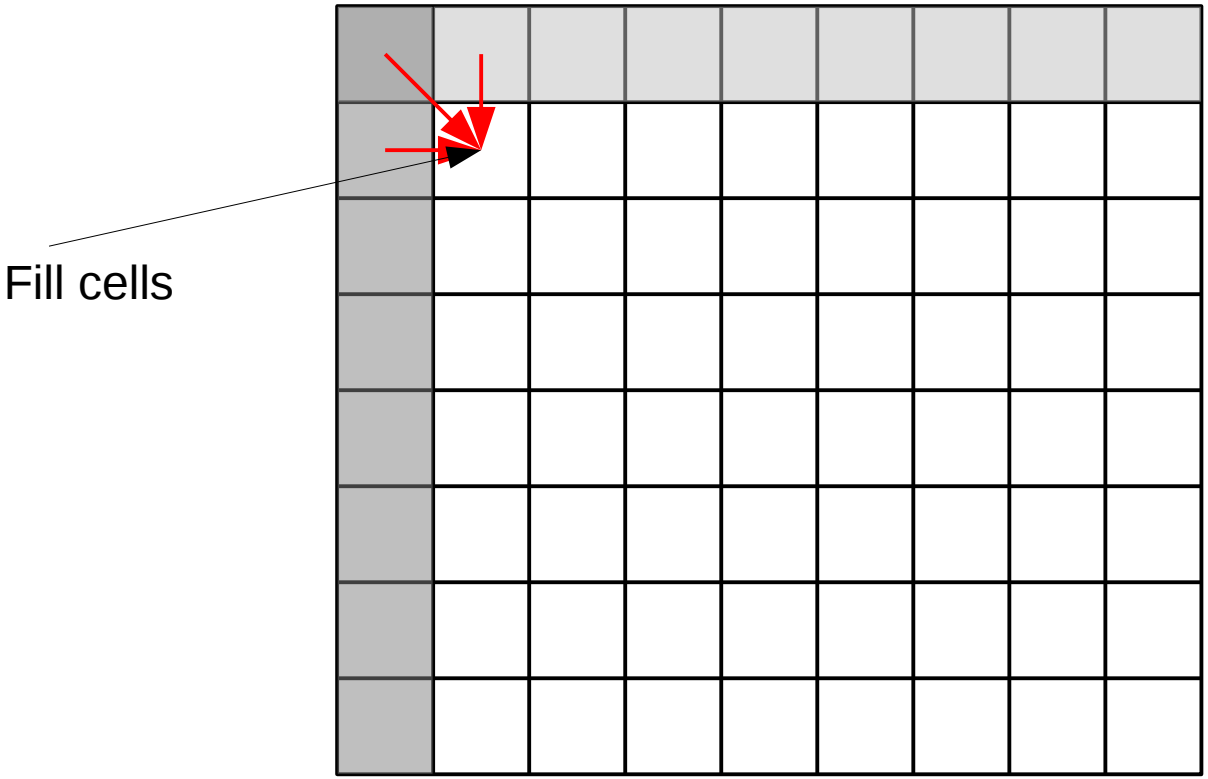
# Wave-Front Parallelism

- Dynamic Programming algorithms



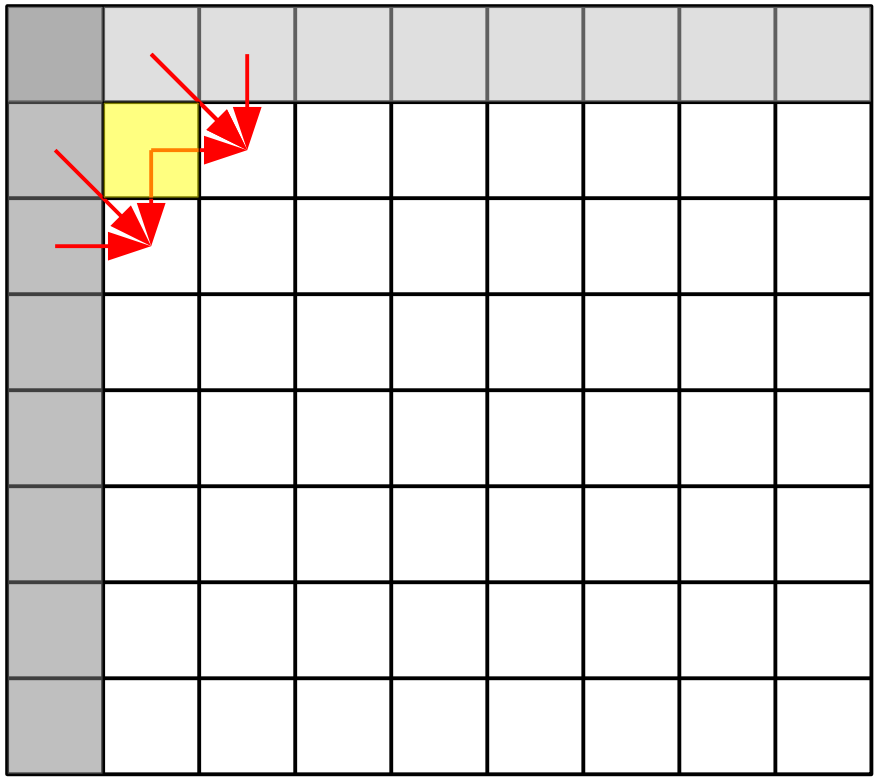
# Wave-Front Parallelism

- Dynamic Programming algorithms



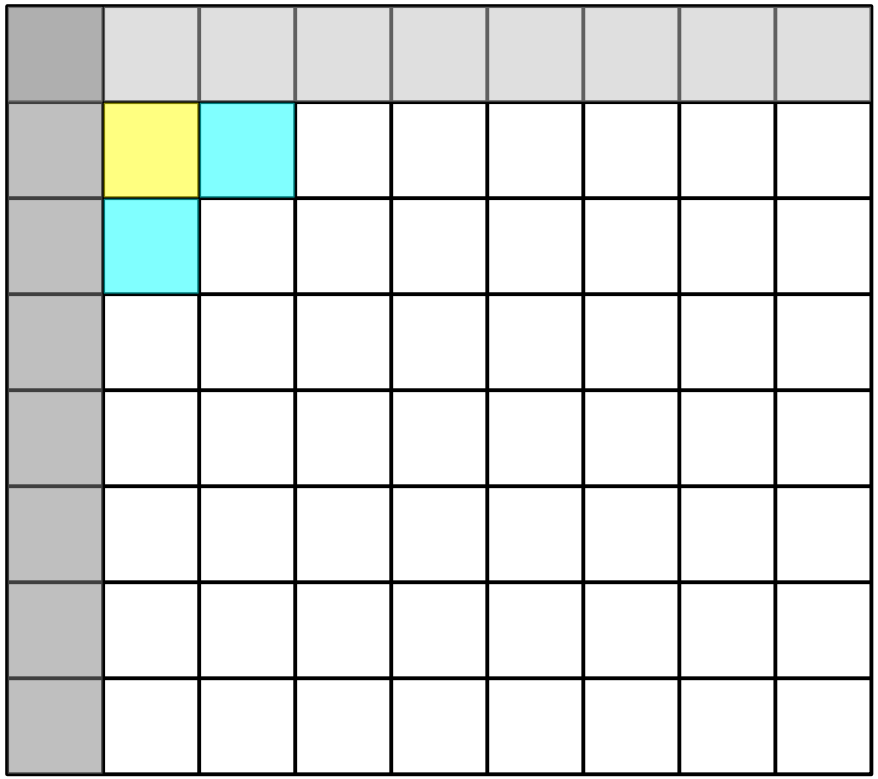
# Wave-Front Parallelism

- Dynamic Programming algorithms



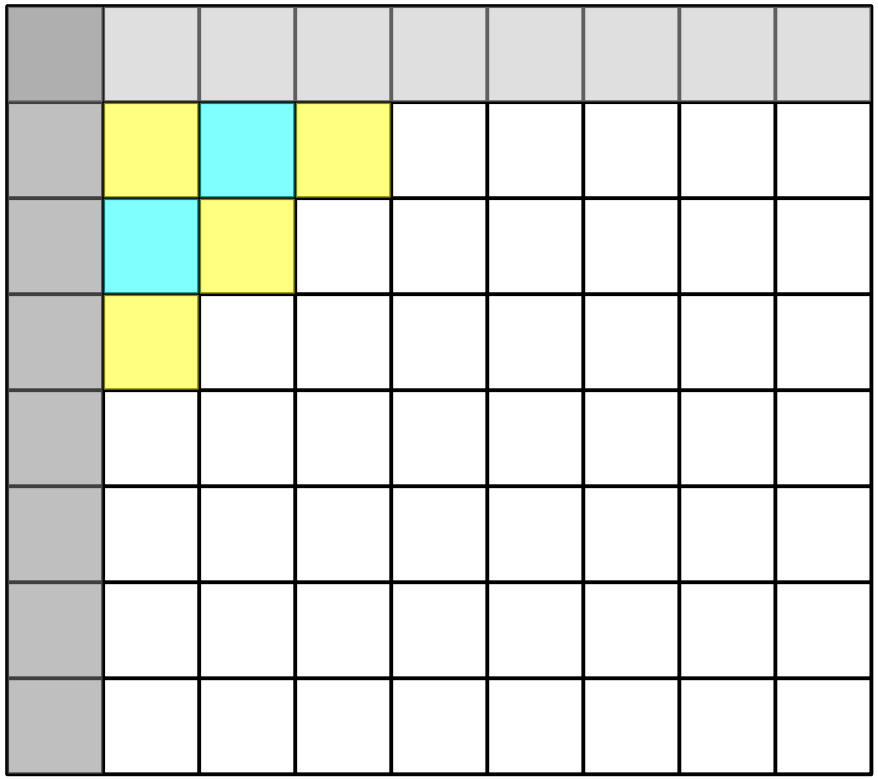
# Wave-Front Parallelism

- Dynamic Programming algorithms



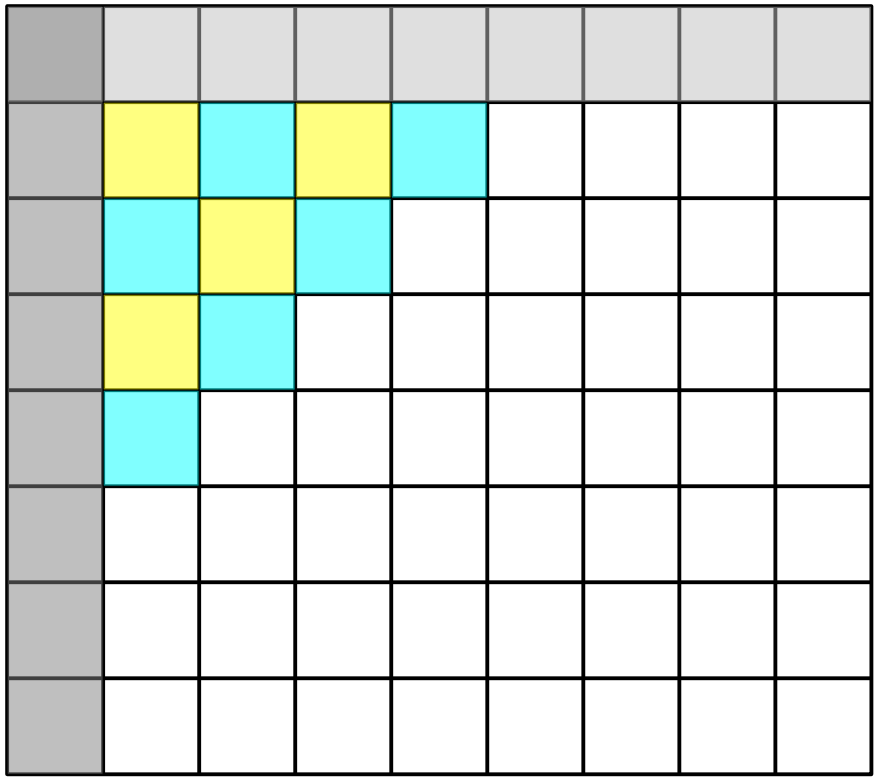
# Wave-Front Parallelism

- Dynamic Programming algorithms



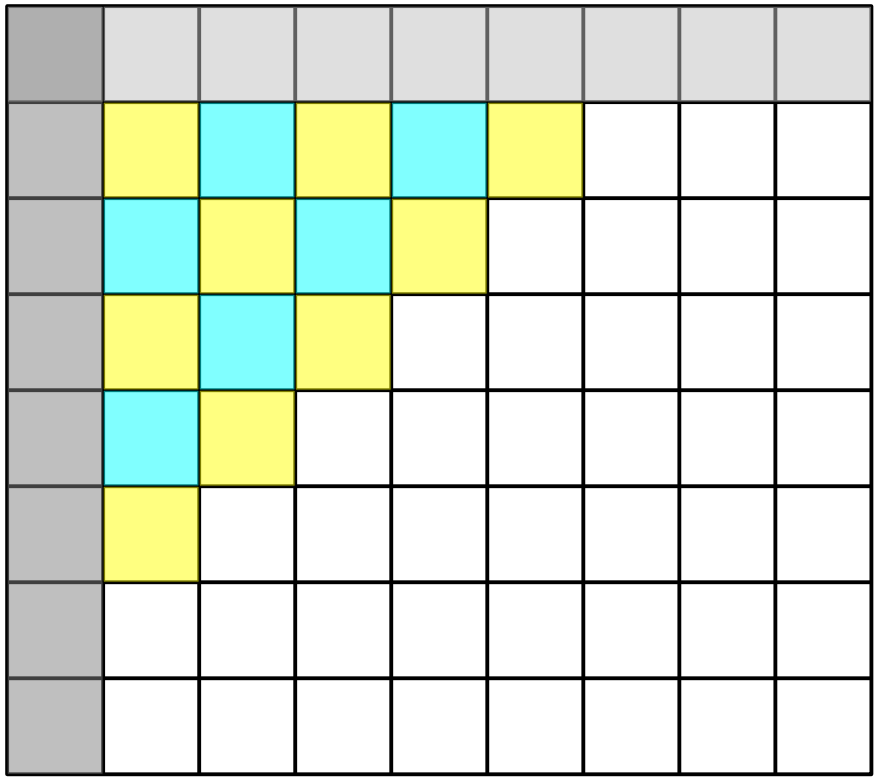
# Wave-Front Parallelism

- Dynamic Programming algorithms



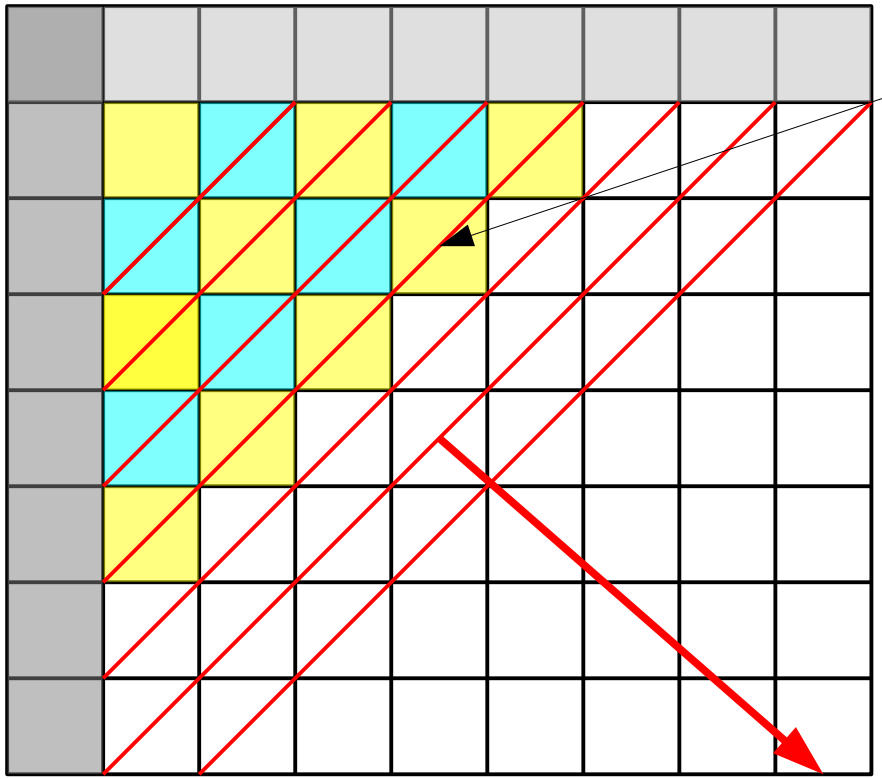
# Wave-Front Parallelism

- Dynamic Programming algorithms



# Wave-Front Parallelism

- Dynamic Programming algorithms



Number of cells
That can be computed
In parallel proceeds
Like a wave-front through
The matrix

# Other ways to vectorize a Dynamic Programming Matrix?

- Assume we have a reference sequence *r* and a lot of query sequences $q_1,\ldots,q_n$ that we want to align to *r*
- Is there an option, other than wavefront, to parallelize this?

# Other ways to vectorize a Dynamic Programming Matrix?

- Assume we have a reference sequence *r* and a lot of query sequences $q_1,\ldots,q_n$ that we want to align to *r*
- Is there an option, other than wavefront, to parallelize this?

  → yes, it's called intra-sequence vectorization :-)

# Hash Tables

- Map a universe of keys U to a much smaller integer-based index table
- Lookup of elements in *O(1)*
- Challenge: define hash function
  - → such that: it is fast to compute
  - → such that: it does not map all keys to the same integer
- Handling collisions: two distinct keys are mapped to the same integer
  - → resolve collisions by chaining
  - → resolve collisions by re-hashing
- A sequence of hash table lookups will generally produce a sequence of random memory accesses

# Parallel Hash-Tables

- Some literature available
- Some recent papers too using new HW capabilities (transaction intrinsics)
- I'd like to have a look at this at some point

# Binary Searches

- Search by comparison & bisection
- Able to find an element in $O(\log n)$

# Kruskal & Prim

- Both invented a minimum spanning tree algorithm!