Infrastructure for LISA Data Analysis

Antoine Basset, Antoine Tran, Matthieu Marseille, Hugues Larat, Hong-Nga Nguyen

LIDA Workshop

25 November 2022

Overview

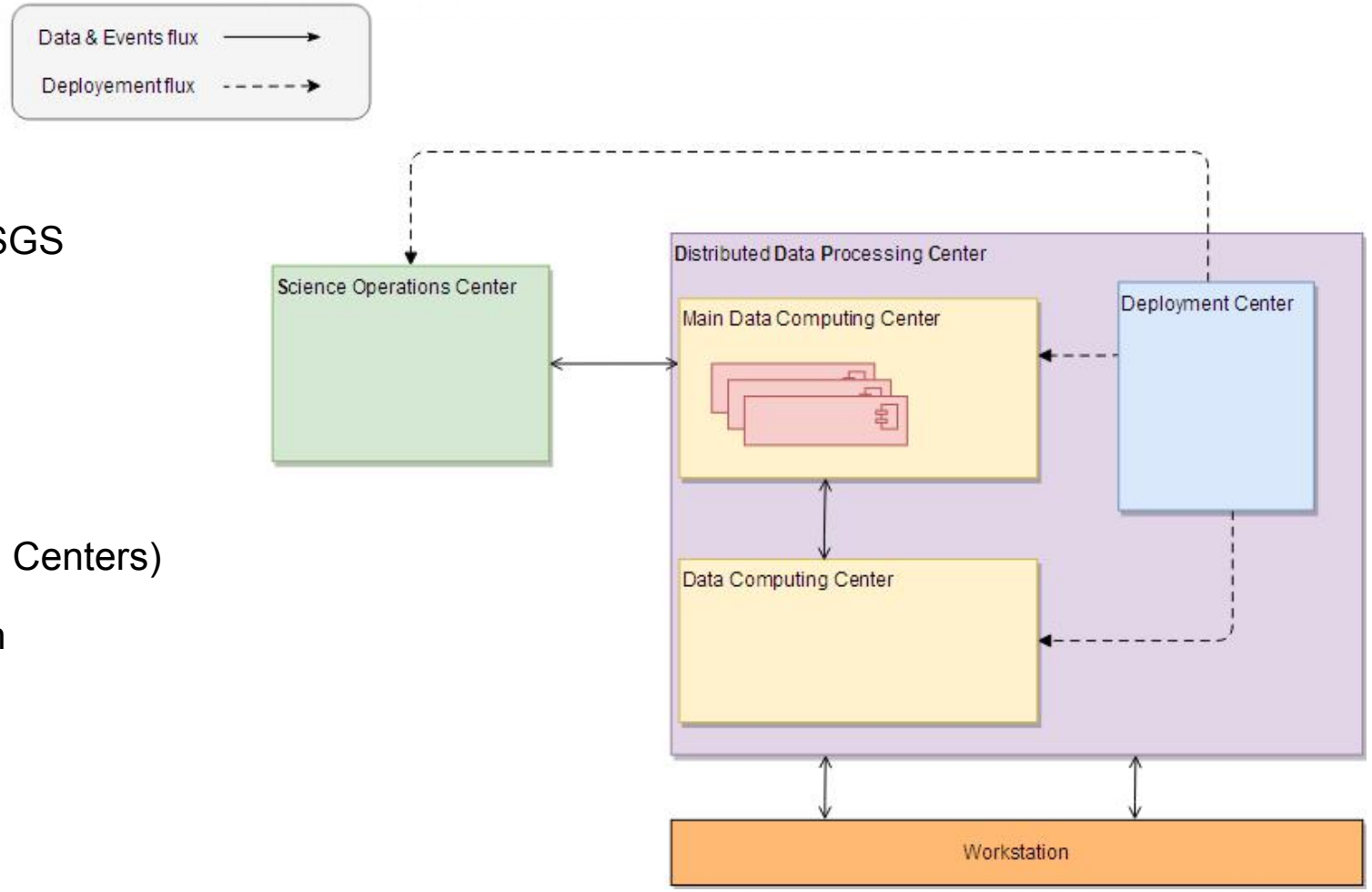
- Current assumptions
- Practical approach with an embryonic GlobalFit
- Coding for Computing Centers

1

Current Assumptions

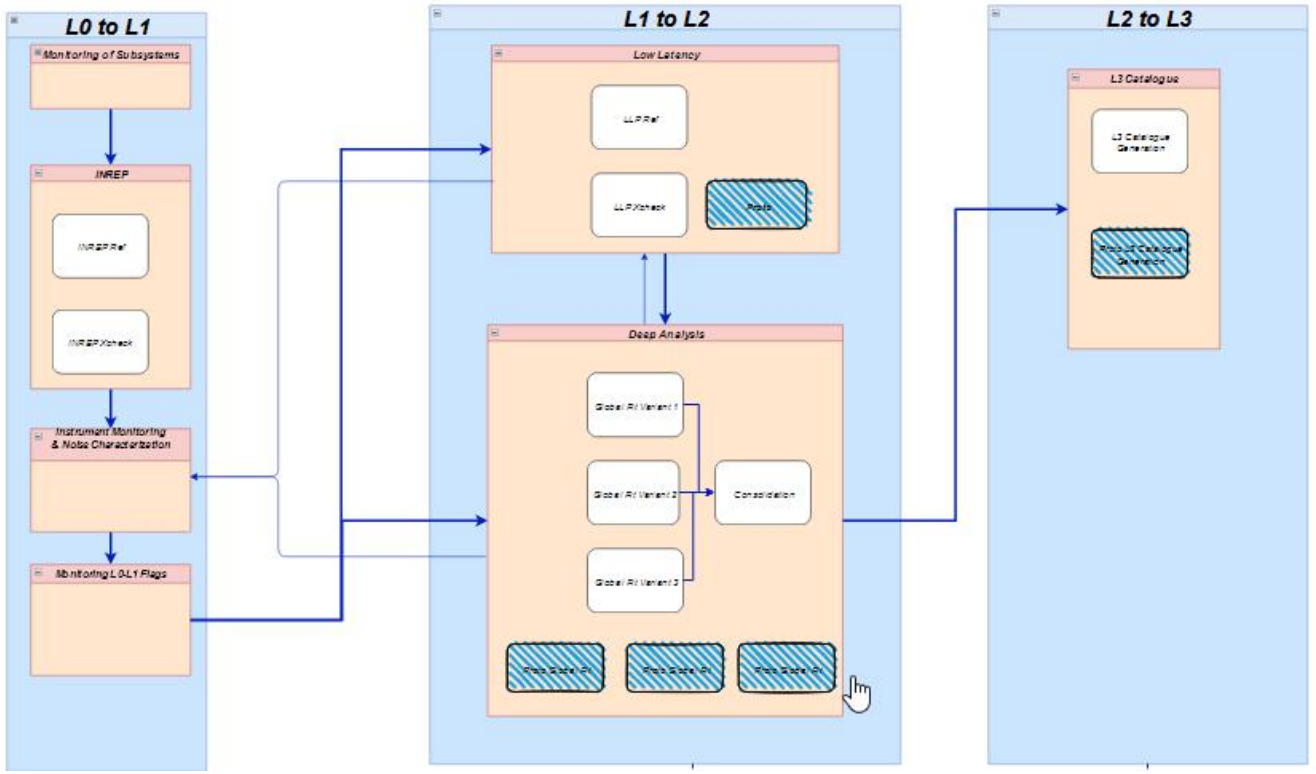
Distributed Data Processing Center

- DDPC = Consortium contribution to SGS
- Science pipeines
- System components
- Infrastructure
  - Several DCCs (Data Computing Centers)
  - One main DCC for orchestration
  - One CI/CD platform



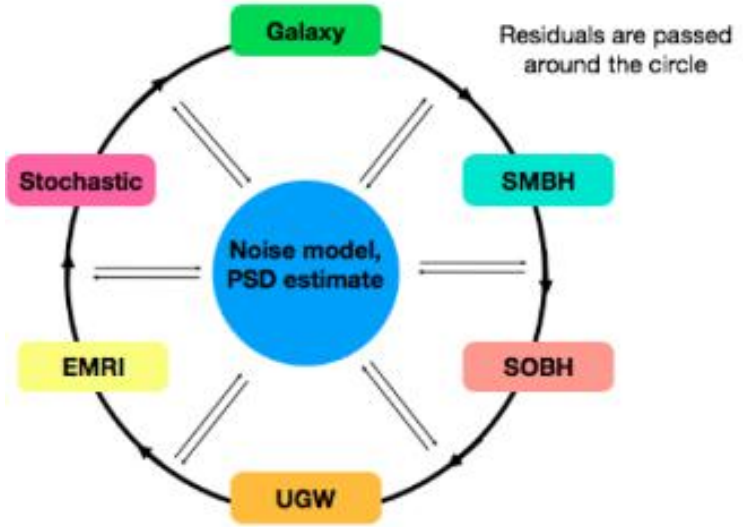
Two Main Pipeline Types

- Low Latency Alert
  - Real time-like constraints
- **GlobalFit**, the heavy one
  - CPU-, (intermediate) data storage-greedy
  - Different implementations will run in production to crosscheck the results
  - For a given implementation, different instances will run on different data segments (1, 3, 6, 12 months)



GlobalFit Computation Model

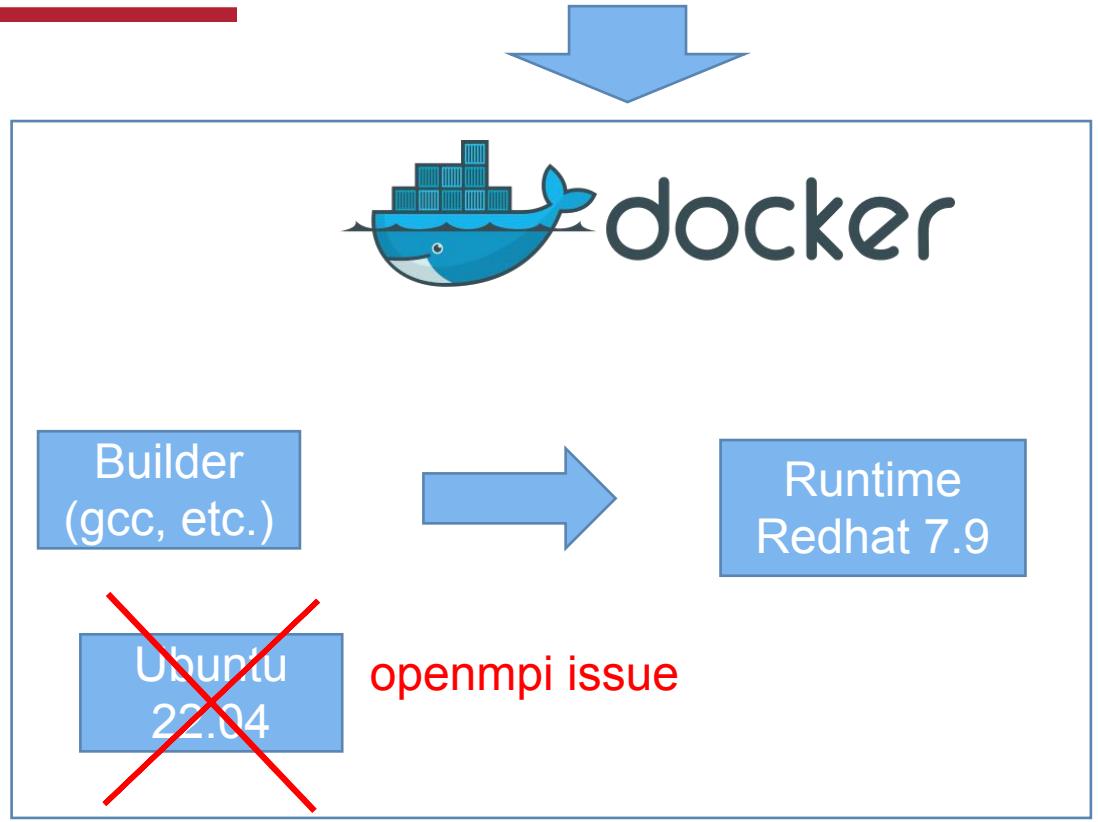
- GlobalFit is too complex to be distributed over several Computing Centers
- Need to better understand concretely a GlobalFit
  - How to parallelize?
  - How to distribute a GlobalFit in a center?
  - **How to build the DDPC?**



2

Practical Approach with an Embryonic GlobalFit

MBH https://github.com/eXtremeGravityInstitute/LISA-Massive-Black-Hole.git
Globalfit https://github.com/tlittenberg/ldasoft



SINGULARITYCE

apptainer.org

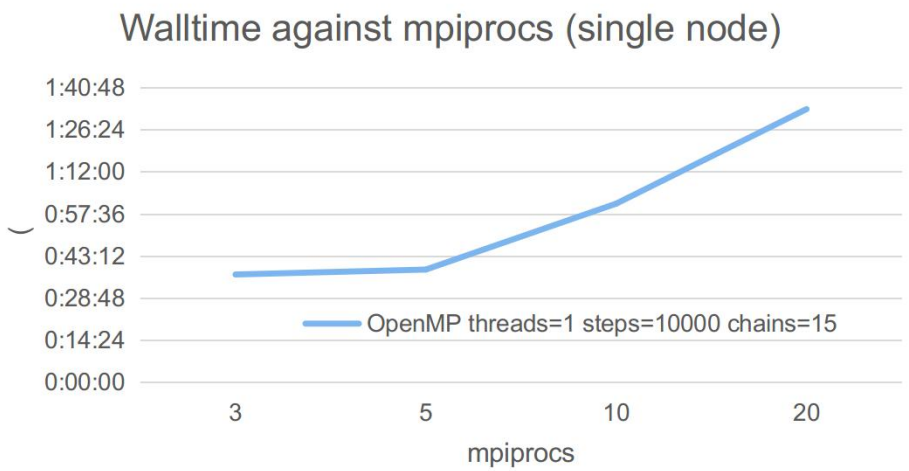
With InfinityBand
drivers (network)

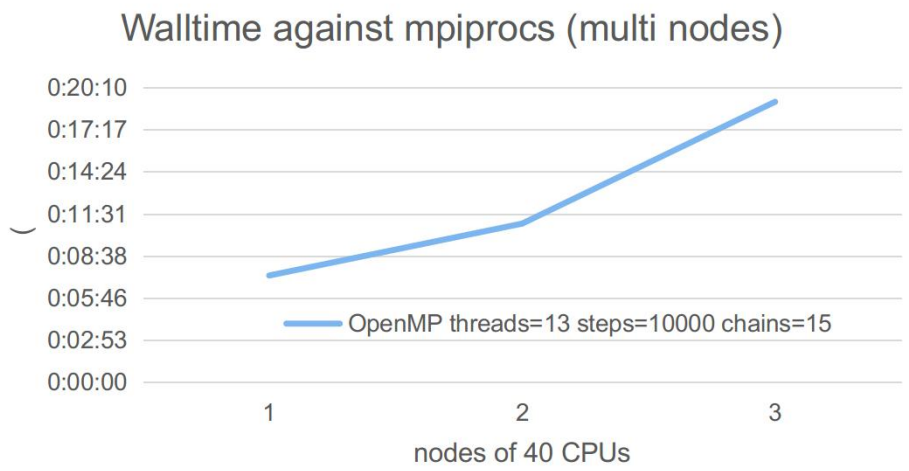
https://gitlab.in2p3.fr/LISA/LDPG/globalfit_prototyping

CNES HPC

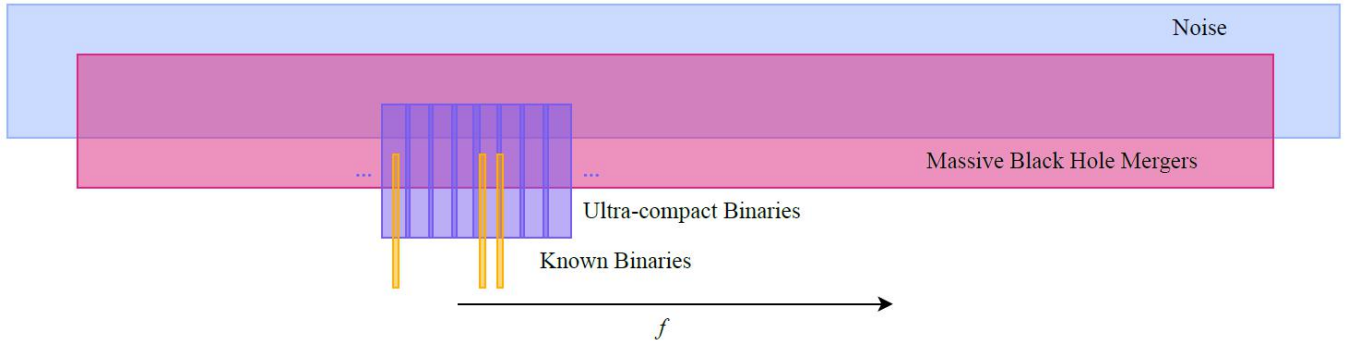
OpenPbs submit

Performance of mcmc (1 step = 100 likelihood calculations * 15 chain = 1500):
* simple mode: no mbh / vgb , only noise + ucb





Conclusion: increasing number of MPI processes/nodes increases walltime



More GB MCMC mpiprocs = more f analyze

Eg: mpiprocs = 5

========== Global Fit Analysis ==========

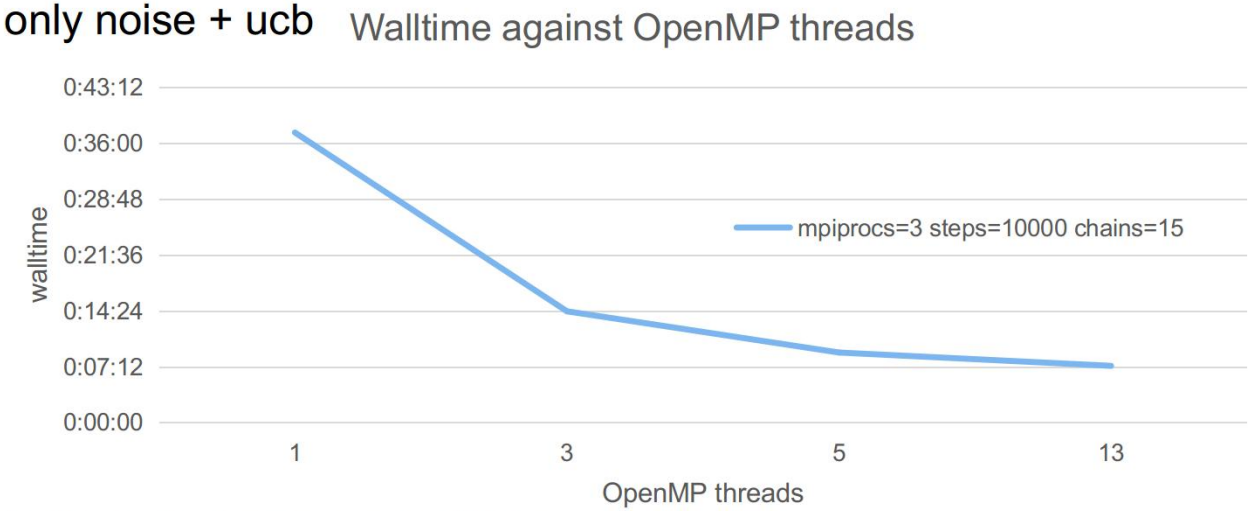
1 noise processes (pid 0)	= noise model
1 vbmcmc processes (pid 1)	= Verification Galactic Binaries
1 mbh processes (pid 2-2)	= Massive Black Hole
2 gbmcmc processes (pid 3-4)	= Ultra Compact Binaries

Conclusion: increasing the number of MPI processes/nodes, because the code allocates the additional cores to more frequencies bands to analyse, thus increasing walltime

CNES HPC

OpenPbs submit

Performance of mcmc (1 step = 100 likelihood calculations * 15 chain = 1500):
* simple mode: no mbh / vgb , only noise + ucb

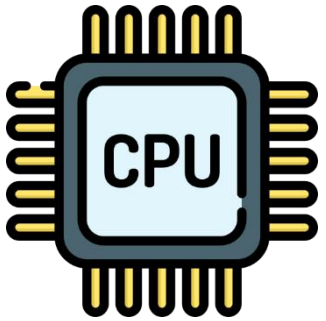


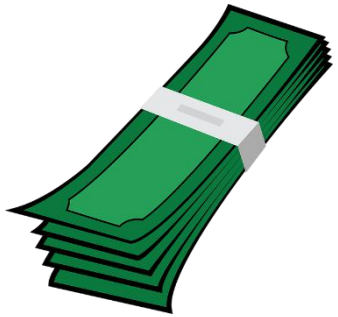
Conclusion: increasing OpenMP threads decrease walltime.

Tyson's Environment

- Cloud: 18 * 72 cpus = 1296 cpus
  - 12 mpiprocs, 6 openmp threads
  - sangria training v2
  - 1 year input => walltime 7 days, **218k cpu.hour**
  - 3 months input => walltime 3 days
- Cloud: 78 * 96 cpus = 7500 cpus
  - 8 mpiprocs / node, 12 openmp threads
  - 1 year input => walltime 7 days, **1260k cpu.hour**

- 5400 CPUs for 1 execution of 24h
  - 12 * 448 vCPU = 294,783.54 USD / month
  - 42 * 128 vCPU = 99,424.54 USD / month
- 1000 CPUs for 1 execution of 7days
  - 15 * 64 vCPU = 15,056.35 USD / month
  - 8 * 128 vCPU = 18,938.01 USD / month






HPC Constraints

- Big number of nodes and CPUs/GPUs
- Shared ressources
- Eg for CSC (Finland)
  - large : 2-1040 cores 3 days max runtime
  - longrun : 1-40 cores 14 days max runtime

Conclusion: in HPC, jobs can be submitted with large ressources and small walltime, or small resources and large walltime, but not large resources and large walltime. There are exceptions but to negotiate and other projects might suffer from exceptions.

Multiple Approaches to Make GlobalFit More HPC Compliant

1. Reduce dataset reading from 1 year to 3 month analysis => reduce walltime from 7 to 3 days with 1300cpus

   => not enough!

2. Generate a new dataset with smaller number of Gravitational Wave sources

   => in progress to generate a new LDC dataset

3. GPU use?

   => work to do, to confirm or not if MCMC can benefit

4. Introduce Checkpoint/Resume

   => see next slide

Checkpoint / Resume (CR)

- At application level: Globalfit does not implement yet CR
- At HPC framework level: very very experimental and in development
- Berkeley Lab Checkpoint/Restart (BLCR)
- Alternative MPI (MPICH2, MVAPICH, IntelMPI) implements CR using BLCR
  Old OpenMPI implemented CR using BLCR, but not maintained in latest
  Too old (last stable in 2013) and might not work in latest kernel
- CRIU
- Great integration in recent kernel/OS
  Currently not integrated in OpenMPI (stalled development)
- DMTCP
- Runs in userland so no need of kernel integration
  Trouble with infiniband TBC + singularity integration

Conclusion: not easy to implement CR at HPC level, but possible. Anyway, application level CR is always more efficient (less memory state to save) but requires more work.

Difficulties During GlobalFit Integration

1. Run with enough CPUs in HPC
2. To have quick scientific validation (post-processing, human verification)

   => generates quick results for automatic verification
3. Benchmark minimum globalfit steps to reduce walltime and have valid results

   => create another metrics called « quality/precision »?

   => generates intermediate output data (like each 10000 steps) to avoid the same calculations but for 10000, 50000, 100000, etc.

3

Coding for Computing Centers

Two Side-by-Side Points of View

Developer

- **Minimize walltime**
- Parallelize as much as possible
- Book unlimited resources!
- Get dedicated hardware (GPU, FPGA...)

Computing Center

- **Maximize resource usage efficiency...**
- Share resources between users
- Limit walltime (target a few hours)
- Limit memory per core (target a few GBs)
- Provide generic/homogeneous hardware
- **... ≈ Minimize sum of all walltimes**

Impact on Pipelines

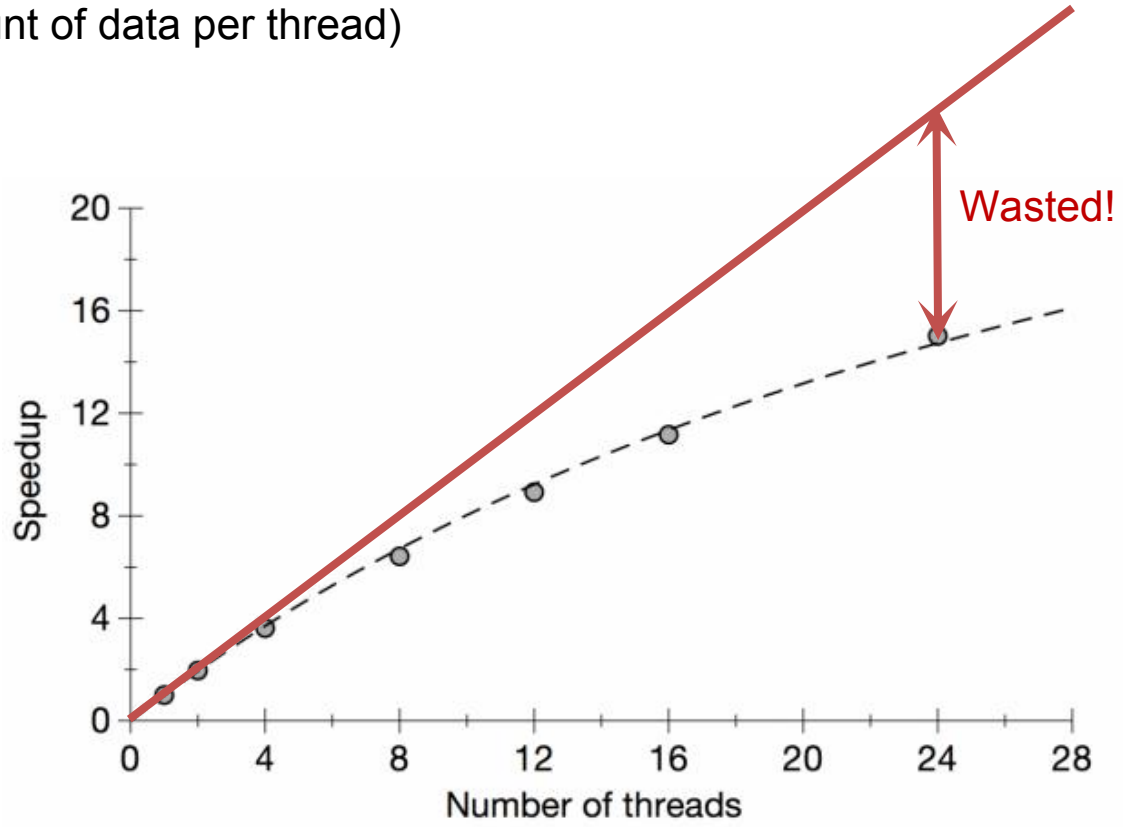
- Long pipelines must be **resumable**
  - Regular checkpoints where the **global** state is saved
  - Restart from a previously saved global state
- Pipelines must have good **scalability** (= speedup vs # threads, see next slide)
  - **Profile** and then optimize and then parallelize! (and loop)
  - Target one thread per booked core
  - If scalability is poor, run sequentially (or rather find the best operating point)
  - Parallelizing small portions of a job is wasting resources (see Amdal's law)

Strong Scaling

- Strong Scaling = **Speedup vs # threads**, for a given total amount of data
- (Weak Scaling = Speedup vs # threads, for a given amount of data per thread)
- Efficiency = Speedup / # threads
- 1 - Efficiency = wasted resources!

# threads	Time	Speedup	Efficiency
1	3.93	1	100%
2	2.00	1.96	**98%**
4	1.09	3.61	**90%**
8	0.61	6.41	**80%**

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/



Some Optimization Opportunities

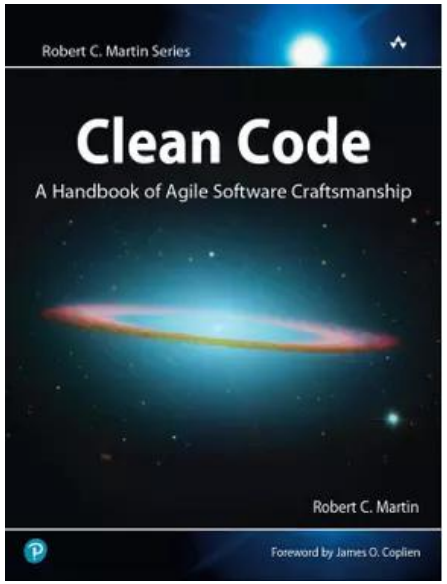
- High-level optimization saves **orders of magnitude**
  - Minimize **operations**: algorithm
  - Minimize (shared) **file system** usage: pass-by-memory, sparsity
- Mid-level optimization saves **factors**
  - Minimize **instructions** per operation: design, implementation, language, precision, vectorization
  - **Cache** computation: compile-time computation, just-in-time compilation
  - Minimize parallelization **overheads**: top-level parallelization, no parallelization!
- Low-level optimization saves **percents to factors**
  - Minimize **cache misses** per instruction: data-oriented design
  - Minimize **branch mispredicts**: data-oriented design

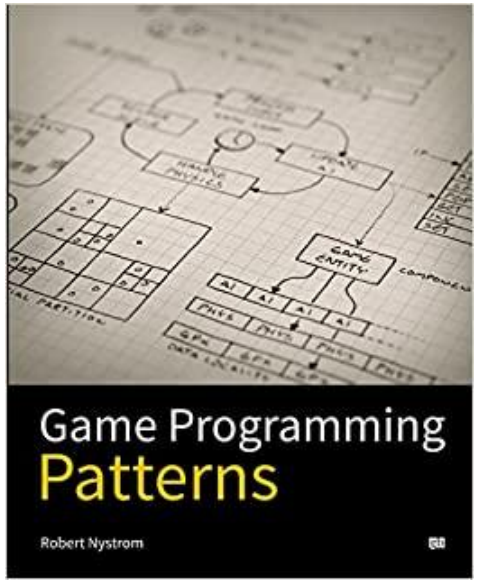
Some Pessimization Habits

- Pessimization = Something **less efficient while not simpler** than another option
  - Allocate many **small objects**, e.g. a list of N-D arrays instead of one (N+1)-D array
  - **Bunny hop** in memory, e.g. misorder nested loops
  - Iterate over **many containers** at once instead of creating one container of structs
  - Use **non-contiguous containers** like linked lists or dictionnaries
  - Fill classes with **temporary members**, i.e. mix hot and cold variables
  - Follow multiple **indirections** in loops

Forget Everything I Just Said...

- The Wise One (Hugues) once said: “85% of the lifetime of a code is spent in **reading** it, not writing it or executing it!”
- Top priority is on **clean code**, not fastest code... (which are not adversarial, though)





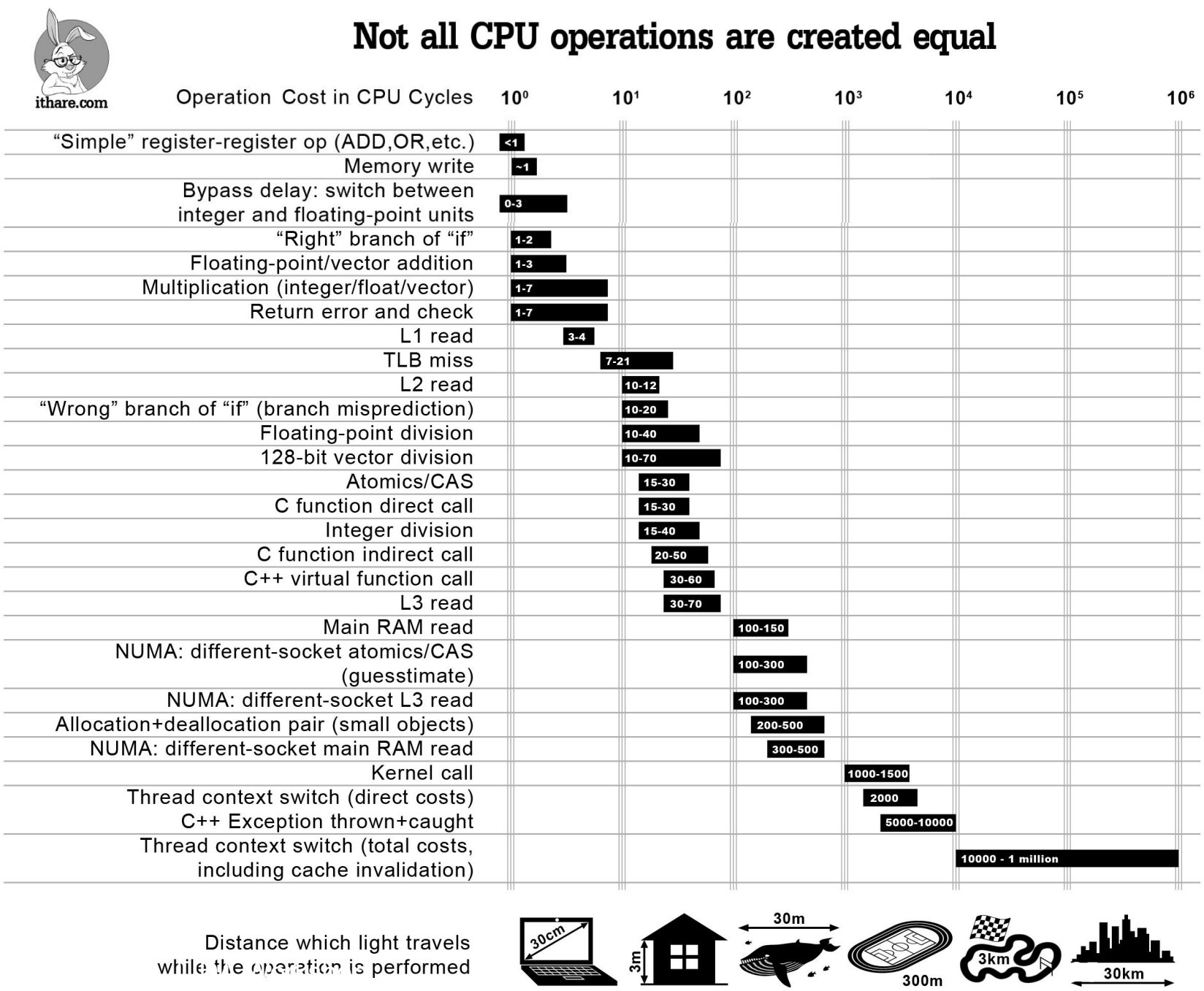
https://gamesfromwithin.com/data-oriented-design

4

Supplementary Material

Clock Cycles

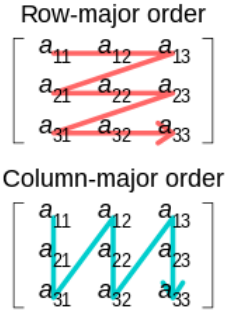
- Branch mispredict
  - 10-20 cycles
- Division
  - 10-40 cycles
- Virtual function call
  - 30-60 cycles
- RAM access
  - 100-150 cycles
- Allocation + deallocation
  - 200-500 cycles



http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Numpy Array Ordering

Matrices -- and more generally ND arrays -- are stored linearly in memory. By default, numpy uses row-major ordering (`C_CONTIGUOUS` in numpy's wording), which means neihboring elements of a row are contiguous in memory, while neighboring elements of a column are separated in memory by the width of the array.



Let us create some random matrix and check the memory layout:

```
import numpy as np

side = 2048
shape = (side, side)

a = np.random.random(shape)
a.flags
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

The Numpy Transpose Fraud/Optimization

With numpy, `transpose()`-ing an array realy means changing the indexing scheme, not touching the data in memory. See how fast the "transform" is (effectively a mere copy) and how we go from row-major ordering to column-major ordering (`F_CONTIGUOUS`).

```
%timeit a.transpose()

at = a.transpose()
at.flags
```

```
98.2 ns ± 2.31 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

We can check that the underlying memory of both arrays have the same contents with `ravel('K')`, which returns a 1D array without reordering elements in memory:

```
np.all(a.ravel('K') == at.ravel('K'))
```

```
True
```

It is possible to force row-major ordering with `ascontiguousarray()`:

```
%timeit np.ascontiguousarray(at)

at_c = np.ascontiguousarray(at)
at_c.flags
```

```
68.6 ms ± 3.75 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

Quiet Cache Misses

In the previous section, we have instantiated three matrices, with the following properties:

- `a` is a random square matrix stored with **row-major** ordering;
- `at` is the transpose of `a` stored with **column-major** ordering;
- `at_c` is `at` stored with **row-major** ordering.

Let's sum `a` and each of its transposes:

```
%timeit a + at
%timeit a + at_c

np.all(a + at == a + at_c)
```

```
68.4 ms ± 4.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
7.43 ms ± 152 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
True
```

Obviously, results are the same. On paper, the exact same mathematical operations are performed, i.e. exactly `a.size` additions. Yet, computation times differ by an order of magnitude. What happened?

The only difference is how efficiently **cache memory** is used, relying on CPU internal functionning.

In the first case (`a + at`), the CPU is effectively busy at 100%, but it spends most of its time filling and emptying its memory instead of carrying actual computations. Unfortunately, it is very difficult to detect the so-called cache misses. Profilers like prints, psutils or cProfile won't help. Instead, intrusive tools like cachegrind or proprietary profilers are needed.

Moreover, libraries like numpy abstract the underlying memory layout away, for convenience, which makes it quite difficult to track memory usage in Python and take care of it.

Still, understanding how code is handled by CPUs, applying a few good practices or benchmarking a few implementation options helps designing cache-efficient programs. This is the purpose of **Data Oriented Design**...

Organize Data for Your Computation

For completeness, here is what happens if we now compute the product of the two matrices:

```
%timeit np.matmul(a, at)
%timeit np.matmul(a, at_c)
```

```
157 ms ± 20.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
219 ms ± 16.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Again, actual mathematical operations are the same, but elements are more efficiently visited in the first case because the computation is performed row-wise on `a` and column-wise on `at`. This demonstrates that there is no computation-agnostic best solution: the only way to organize data well is to do so accordingly to their usage.