Practical HPC-Software Engineering for Research

Jonas Thies (Jonas.Thies@DLR.de)
Institute of Simulation and Software Technology
High Performance Computing

Aspects of modern software development

- Distributed development processes via git, subversion,…
- Community software (github, bitbucket,…)
- Open source licensing (BSD, MIT, (L)GPL,…)
- Software architecture
- Build systems (CMake, Autotools,…)
- Meta build systems (Spack, EasyBuild, Conda)
- Test frameworks (GoogleTest, PyTest, jUnit,…)
- Continuous integration testing (Jenkins, gitlab-ci,…)
- Integrated development environments (IDEs, e.g. Eclipse, QtCreator,

Do I need all that???

Probably not. But some of it may be **very useful**

At DLR we categorize software in order to come up with a reasonable subset for each Individual software effort:

Class 0: short scripts, mostly private use, purpose: try something out, generate plots for a paper etc.

Class 1: prototypical software that should be used and extended by others

Class 2: Software intended for long-term use also outside the own group

Class 3: critical software or software with product character

From the DLR Software Engineering Guidelines

Checklists for different maturity levels

Change Management

Recommendation	Comment	Status
EÄM.2: The most important information describing how to contribute to development are stored in a central location. *(from application class 1)*	Build steps are missing	todo
EÄM.5: Known bugs, important unresolved tasks and ideas are at least noted in bullet point form and stored centrally. *(from application class 1)*		ok
EÄM.7: A repository is set up in a version control system. The repository is adequately structured and ideally contains all artifacts for building a usable software version and for testing it. *(from application class 1)*		ok
EÄM.8: Every change of the repository ideally serves a specific purpose, contains an understandable description and leaves the software in a consistent, working state. *(from application class 1)*		ok

Reasoning and further advice

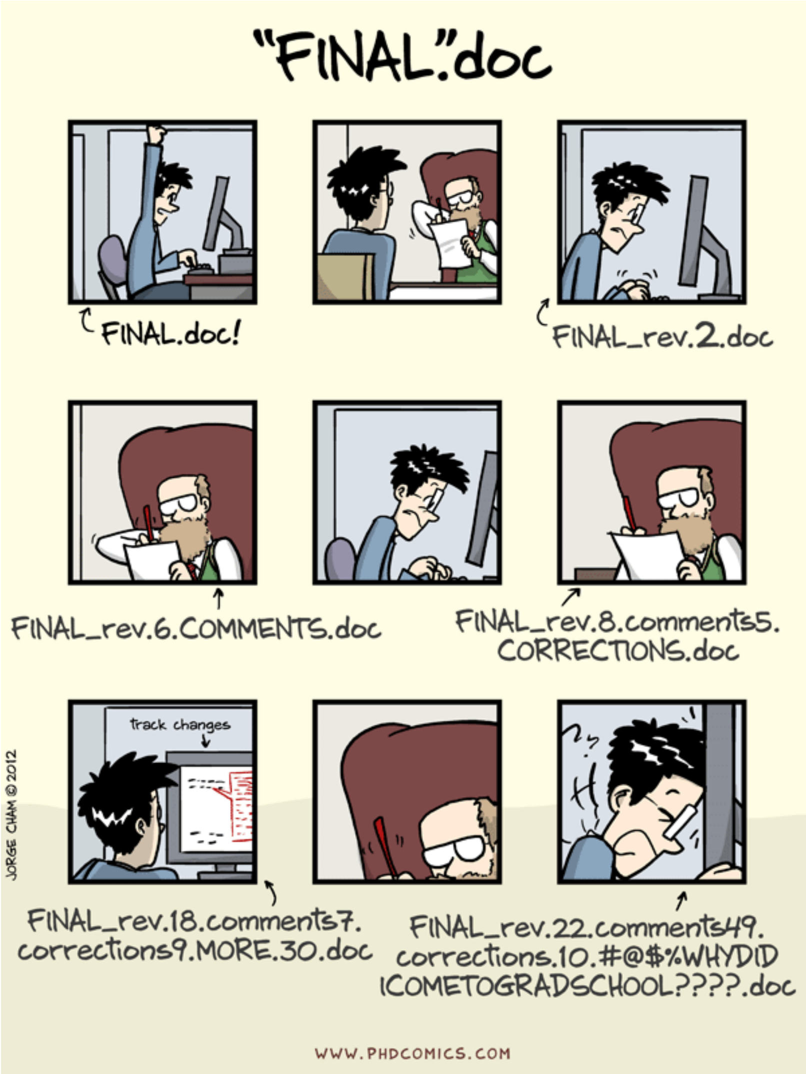
The repository is the central entry point for development. All main artifacts are stored in a safe way and are available at a single location. Each change is comprehensible and can be traced back to the originator. In addition, the version control system ensures the consistency of all changes.

The repository directory structure should be aligned with established conventions. References are usually the version control system, the build tool (see the Automation and Dependency Management section) or the community of the used programming language or framework. Two examples:

TAO:

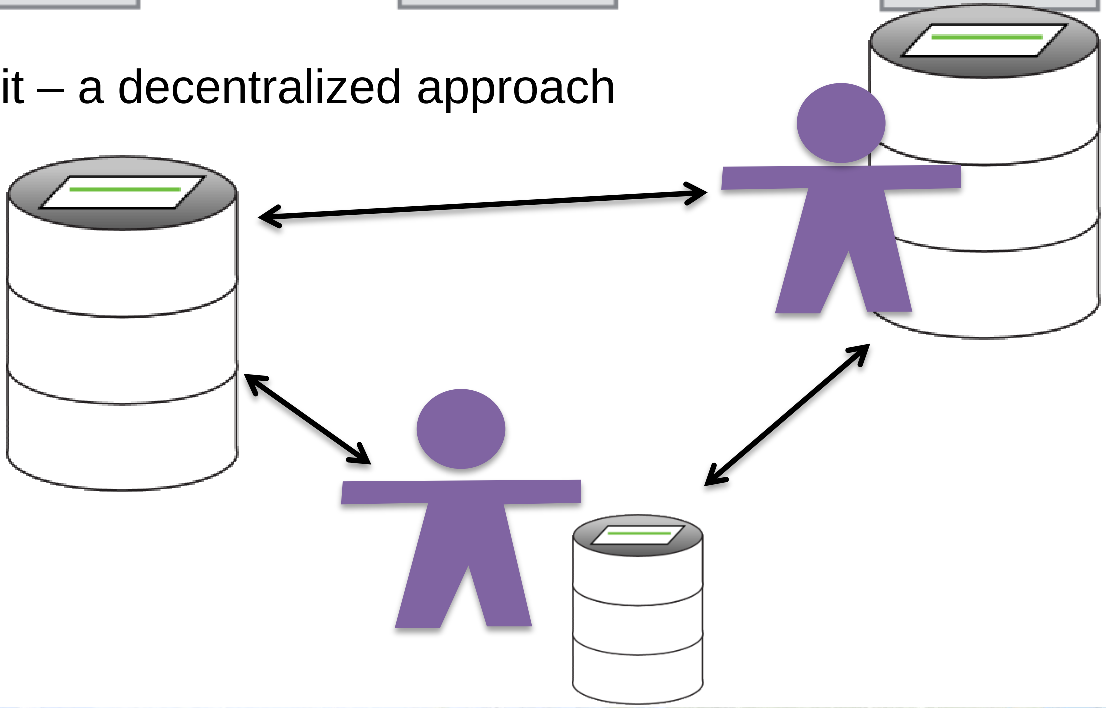
- If it's not in the repository, it doesn't exist.
- If there is no unit test, the feature will eventually break.

Version control – **why and how**



- Simple case: version = sequence changes:

- Git – a decentralized approach

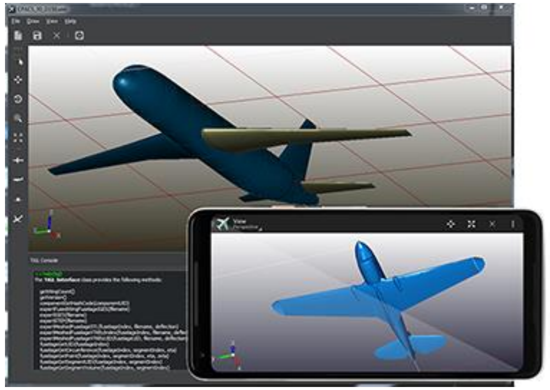


Git – **basic look & feel**

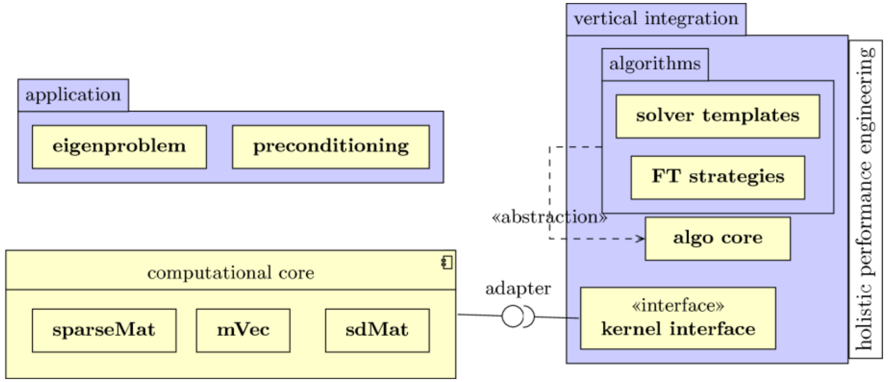
```
jonas@ubuntu: ~/test/git_demo
File Edit View Search Terminal Tabs Help
jonas@ubuntu: ~/essex/phist    jonas@ubuntu: ~/test/git_demo

commit 6e6257a9658b1a6f476464131d54ee464cc65d4d (HEAD -> master)
Author: Jonas Thies <Jonas.Thies@DLR.de>
Date:   Fri Jul 5 13:54:06 2019 +0200

    new function for comparing model states,
    fix a broken test and add a makefile

commit db831d6b0448e4e11c6af65fefbd0533a46d029f
Author: Jonas Thies <Jonas.Thies@DLR.de>
Date:   Fri Jul 5 09:22:17 2019 +0200

    Re #33: added a status flag to the model struct
    so that the print function can recognize wether the model is initialized.
    @charly, please check if this satisfies your requirements on the model status.

commit b64fd808d90f2621a8bc8f09ee4ca715b49fbb16
Author: Jonas Thies <Jonas.Thies@DLR.de>
Date:   Fri Jul 5 09:21:20 2019 +0200

    latest version

commit 5d0da7421b4477a8bba6359d68a8c9d90a05675f
:
```

DLR

Testing your code

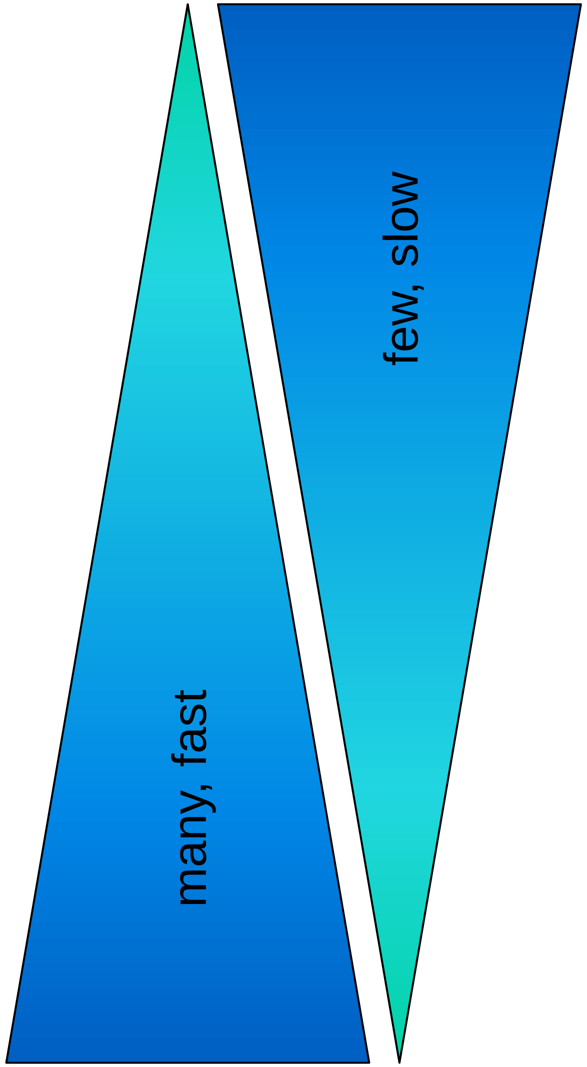


- **System tests:** for a given set of input data, the overall software produces the expected output data



Software architecture of the phist software (https://bitbucket.org/essex/phist)

- **Integration tests:** make sure that parts of the software work together as expected
- **Unit tests:** test for correct behavior of classes and subroutines with synthetic input data



GoogleTest – **basic look & feel**

sc-030114l.intra.dlr.de - PuTTY

```
thie_jo@sc-030114l:~/essex/phist/build_builtin> ./phist-1.7.7-kernels-test-Release --gtest_filter=DMvecTest*100_5*scale
[sc-030114l.intra.dlr.de:32274] mca_base_component_repository_open: unable to open mca_mtl_psm: libpsm_infinipath.so.1: cannot open shared object
 file: No such file or directory (ignored)
Pinning the OpenMP threads to the cores based on simple heuristics (may fail).
Using 1 ranks per node with 24 threads each
PE0: Result of pinning is coreId(threadId):     6 (6)    18 (18) 4 (4)    19 (19) 13 (13) 14 (14) 1 (1)    22 (22) 11 (11) 12 (12) 0 (0)    10 (10) 5
 (5)    23 (23) 16 (16) 3 (3)    15 (15) 7 (7)    2 (2)    8 (8)    20 (20) 21 (21) 9 (9)    17 (17)
Note: Google Test filter = DMvecTest*100_5*scale
[==========] Running 6 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 2 tests from DMvecTest_100_5
[ RUN      ] DMvecTest_100_5.scale
[       OK ] DMvecTest_100_5.scale (0 ms)
[ RUN      ] DMvecTest_100_5.vscale
[       OK ] DMvecTest_100_5.vscale (0 ms)
[----------] 2 tests from DMvecTest_100_5 (1 ms total)

[----------] 2 tests from DMvecTestWithAlignedViews_100_5
[ RUN      ] DMvecTestWithAlignedViews_100_5.scale
[       OK ] DMvecTestWithAlignedViews_100_5.scale (1 ms)
[ RUN      ] DMvecTestWithAlignedViews_100_5.vscale
[       OK ] DMvecTestWithAlignedViews_100_5.vscale (1 ms)
[----------] 2 tests from DMvecTestWithAlignedViews_100_5 (2 ms total)

[----------] 2 tests from DMvecTestWithUnalignedViews_100_5
[ RUN      ] DMvecTestWithUnalignedViews_100_5.scale
[       OK ] DMvecTestWithUnalignedViews_100_5.scale (1 ms)
[ RUN      ] DMvecTestWithUnalignedViews_100_5.vscale
[       OK ] DMvecTestWithUnalignedViews_100_5.vscale (0 ms)
[----------] 2 tests from DMvecTestWithUnalignedViews_100_5 (1 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 3 test cases ran. (5 ms total)
[  PASSED  ] 6 tests.
thie_jo@sc-030114l:~/essex/phist/build_builtin>
```

Summary: how much software engineering should a PhD student do?

It depends. Generally, one should focus on the contents (the what), not the software development process (the how). Certain tools are, however, crucial for the efficient development of your software.

Happy to answer any remaining questions now or later:
Jonas.Thies@DLR.de

- Use version control for practically everything
  - my recommendation: git
  - group and annotate commits in a useful way
- Write unit tests – they are as important as the program code itself, and guide you towards a modular code structure
- Automate repetitive processes like configuring, compiling and testing the code
- Often, code hosting, issue tracking, continuous integration testing and a wiki for documentation are offered as a packaged solution (gitlab, github, bitbucket,…)

Advanced Topics

Part 2: Testing parallel code

- Levels of parallelism
- New “parallel” bugs
- Tools for specific bugs
- Unit tests
- Conclusion

Part 3: Performance engineering

- CPU architecture
- Performance modeling
- Performance “bugs”
- Finding bottlenecks
- Conclusion

Levels of parallelism: **SIMD**

- SIMD: "*single instruction, multiple data*"
- Also called SIMT ("*single instruction, multiple threads*") on GPUs

- Example: AVX-floating point unit of the CPU:
  (FMA operation calculates 4 double-precision fused multiply-add commands in one step)

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} \leftarrow \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

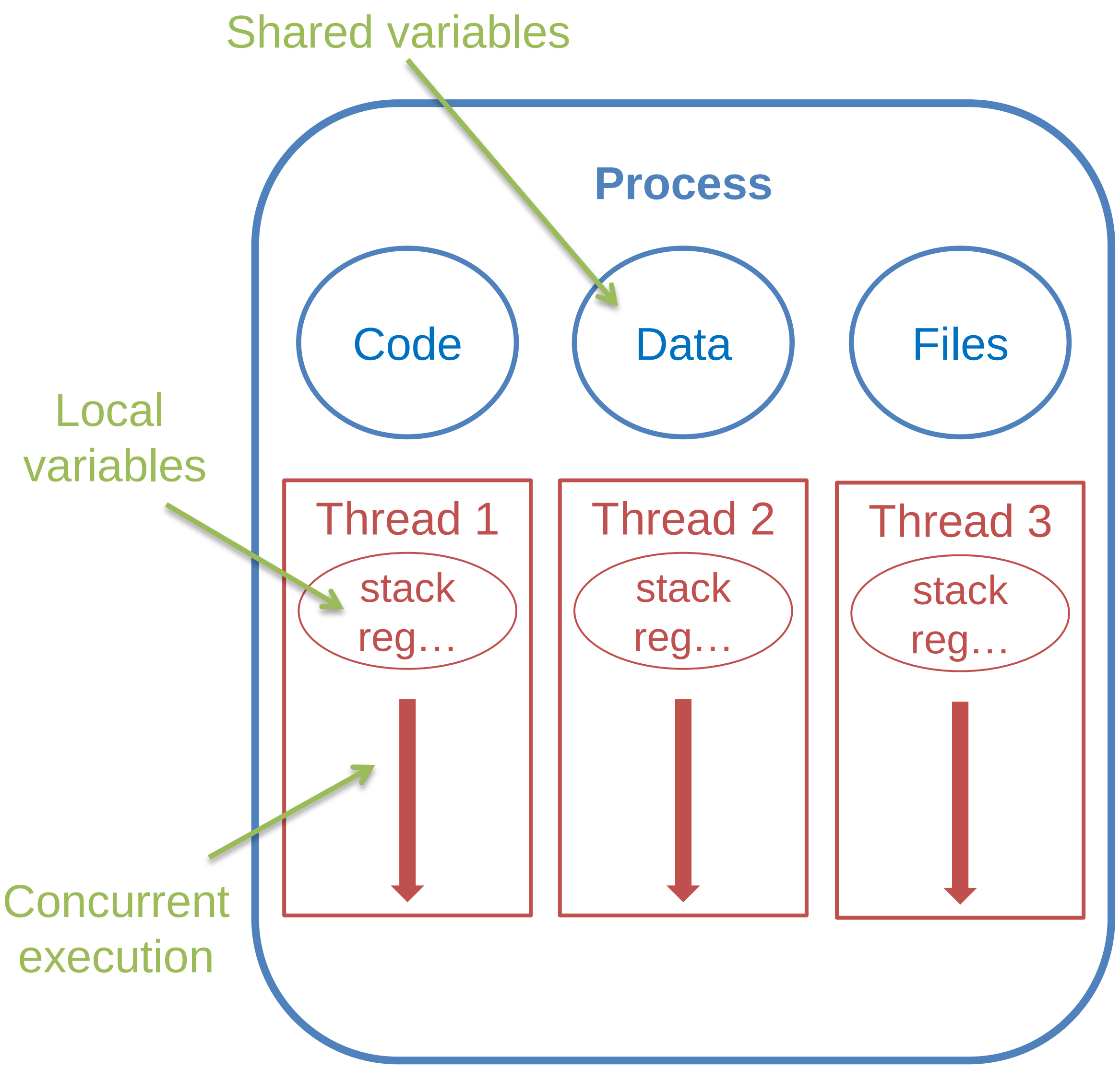
- Requirement: **Alignment** of data (pointer addresses must be a multiple of 32 bytes)
  - Handled by the compiler
  - Debugging only needed for hand-written SIMD code

  ⇒ not further discussed here

- Helpful tool: Intel SDE (https://software.intel.com/en-us/articles/intel-software-development-emulator)

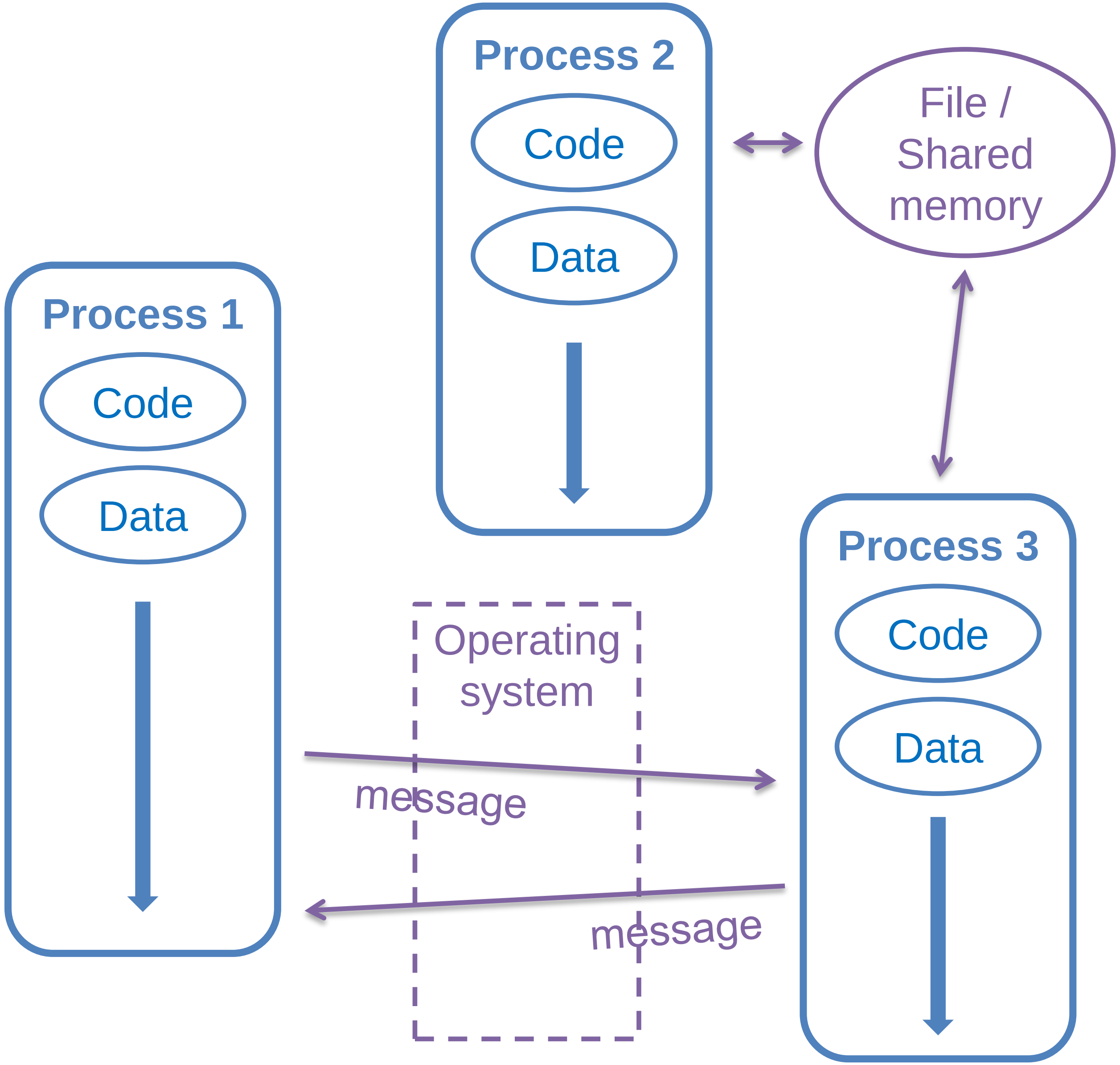
Levels of parallelism: **multi-threading**

- Threads: “*lightweight processes*”
  - Own execution stack
  - Shared data & resources (like files)
- Requires **synchronization**
  - to access shared data & exchange results
  - to access unique resources
- Programming models:
  - Work sharing
  - Task-based
  - Master-worker / Thread-pool, …
- Programming “languages”:
  - Languages: C++11, Java, ~~Python~~
  - Directives: **OpenMP** with C/C++/Fortran
  - Libraries: Qt (high-level), pthreads (low-level), …



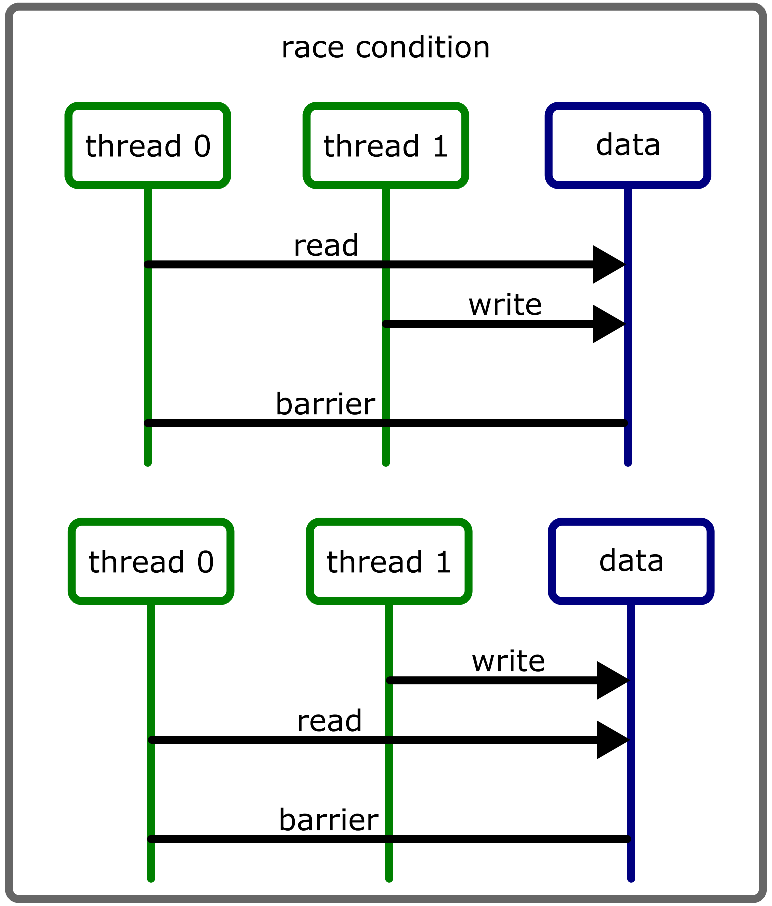
Levels of parallelism: **multi-processing**

- Processes: “*individual execution contexts*”
    - Own execution stack & data
    - Shared OS environment
- Requires inter-process **communication**
    - Shared data (files, memory)
    - Message passing
- Programming models:
    - Server-client
    - SPMD (“*single program multiple data*”)
    - PGAS (“*partitioned global address space*”)
- Programming “languages”:
    - SPMD: **MPI** + C/C++/Fortran
    - PGAS: GASPI, C++Dash, Fortran’08
    - …



New “parallel” bugs: **race conditions**

- Concurrent access to the same data element:
    - Read + write
    - Write + write
- Common pitfall for multi-threading
- **Non-deterministic** ⇒ difficult to reproduce & examine
- Another example TOCTTOU (“*time of check to time of use*”)
    - Also possible over network (client-server scenario)

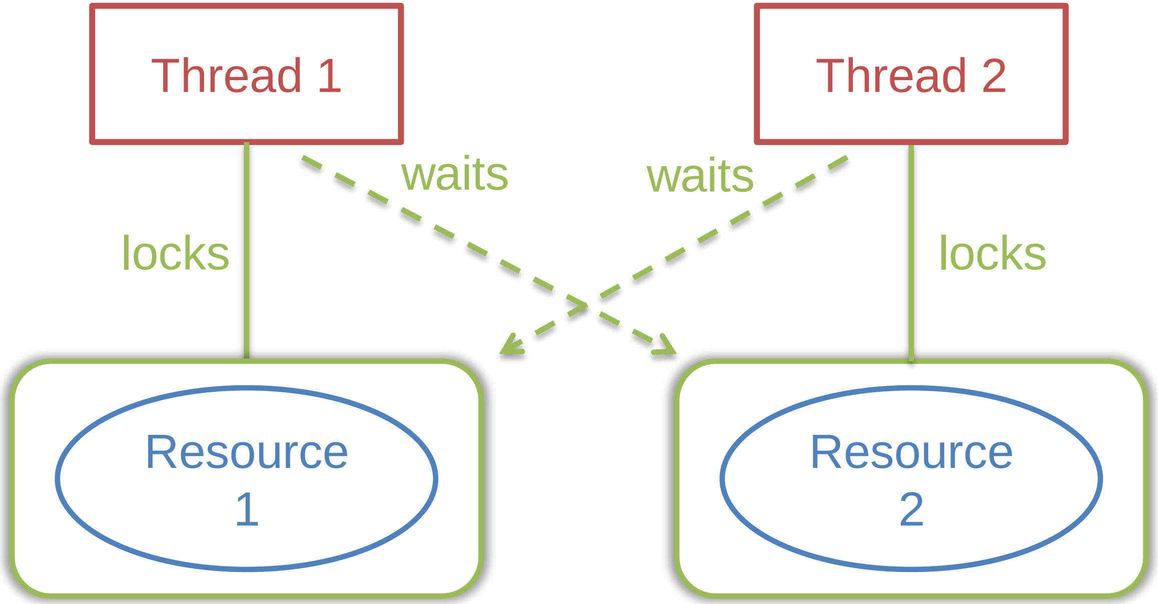


New “parallel” bugs: **deadlocks**

- *Circular blocking waiting*:
  - 2 or more threads / processes
  - waiting while blocking other resources

- Rare, but no easy recovery / avoidance

- **Non-deterministic** ⇒ difficult to reproduce & examine



Tools for specific bugs: **compiler instrumentation**

- **Sanitizer** options for modern GCC and Clang
  - For C/C++/Fortran on Linux
  - Quite fast
  - Need to recompile everything
  - Readable output with debug symbols
  - Open Source: https://github.com/google/sanitizers/wiki

- *Thread* sanitizer:
  - Detects **race conditions** and **deadlocks** for multi-threaded programs
  - Activated with `-fsanitize=thread`
  - Possibly reports false positives

Not specific to parallel programs:

- *Address* sanitizer:
  - Detects **invalid memory access**
  - Detects memory (de)allocation errors
  - Activated with `–fsanitize=address`
  - Crucial for low-level or parallel code

- *Undefined behavior* (UB) sanitizer:
  - Finds unexpected bugs
  - UB: special cases with no guaranteed behavior
  - Activated with `-fsanitize=undefined`
  - Useful from time-to-time…

Tools for specific bugs: **valgrind**

- **Debugging tool**
  - For Linux
  - Extremely slow
  - Works with (almost) all executables
  - Readable output with debug symbols
  - Open Source: http://valgrind.org/

- *Helgrind* (or *DRD*) tool:
  - Detects **race conditions** and **deadlocks** for multi-threaded programs
  - Run with `valgrind –tool=helgrind <exe>`
  - Possibly reports false positives

Not specific to parallel programs:

- *Memcheck* tool:
  - Detects **invalid memory accesses**
  - Detects memory (de)allocation errors
  - Detects uninitialized data
  - Run with `valgrind --tool=memcheck <exe>`
  - MPI-support to detect MPI buffer errors (needs special compiler flags + LD_PRELOAD)
  - Sometimes reports false positives
  - Crucial when address sanitizer is no option

- Performance tools (*cachegrind*, etc.):
  - Not so useful as the hardware is emulated…

Tools for specific bugs: **must**

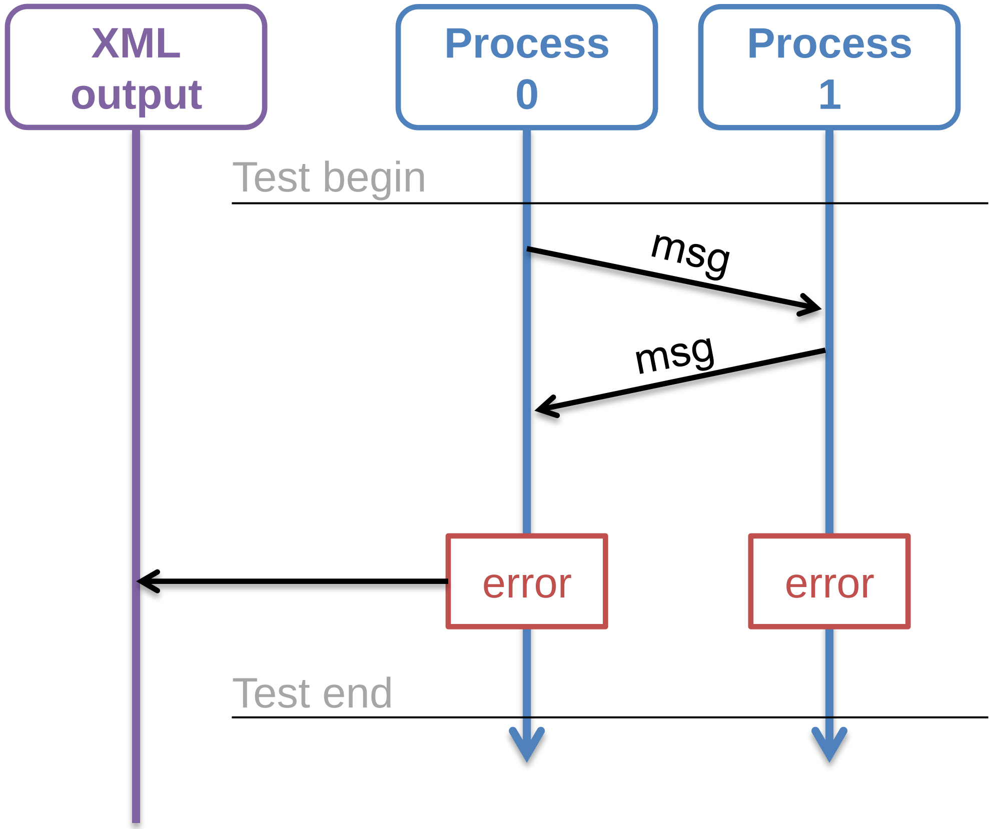
- **MPI communication checker**
  - Detects MPI usage errors
  - Detects deadlocks with MPI
  - Will detect data races with one-sided communication in MPI
  - Run program with `mustrun -np <n> <exe>`
    (instead of `mpirun -np <n> <exe>`)
  - Open Source: https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

Unit tests: **problems from the wild (1)**

- Setup:
  - parallel unit tests with
  - 2 processes
  - Output on process 0

- Same error on all processes

⇒ **Error reported correctly**

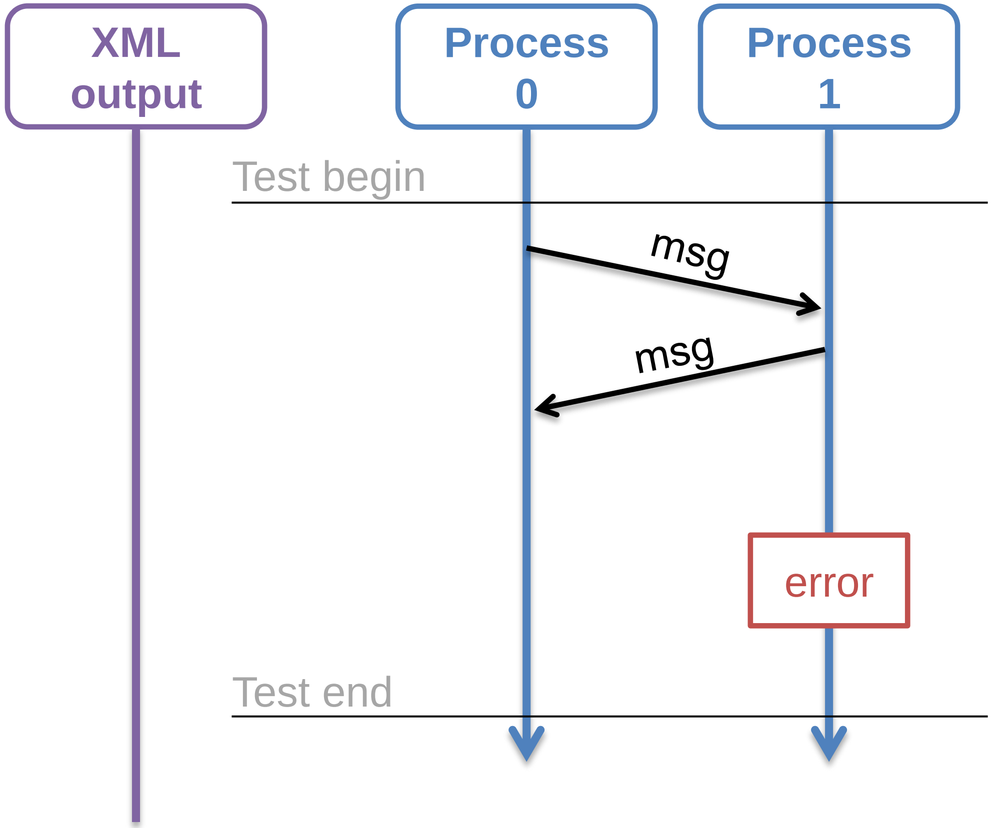


Unit tests: **problems from the wild (2)**

- Setup:
  - parallel unit tests with
  - 2 processes
  - Output on process 0

- Error only on process 1

⇒ **Error not reported!**

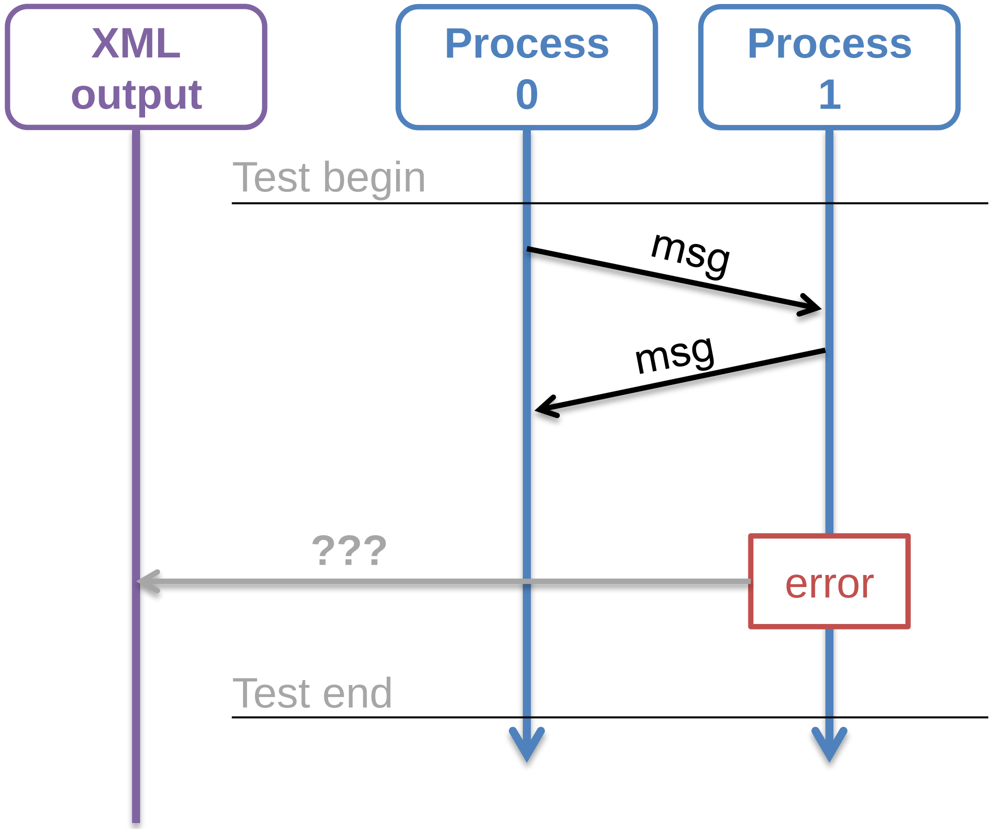


Unit tests: **problems from the wild (3)**

- Setup:
  - parallel unit tests with
  - 2 processes
  - Output on all processes

- Error only on process 1

⇒ **Multiple processes write into the same file!**

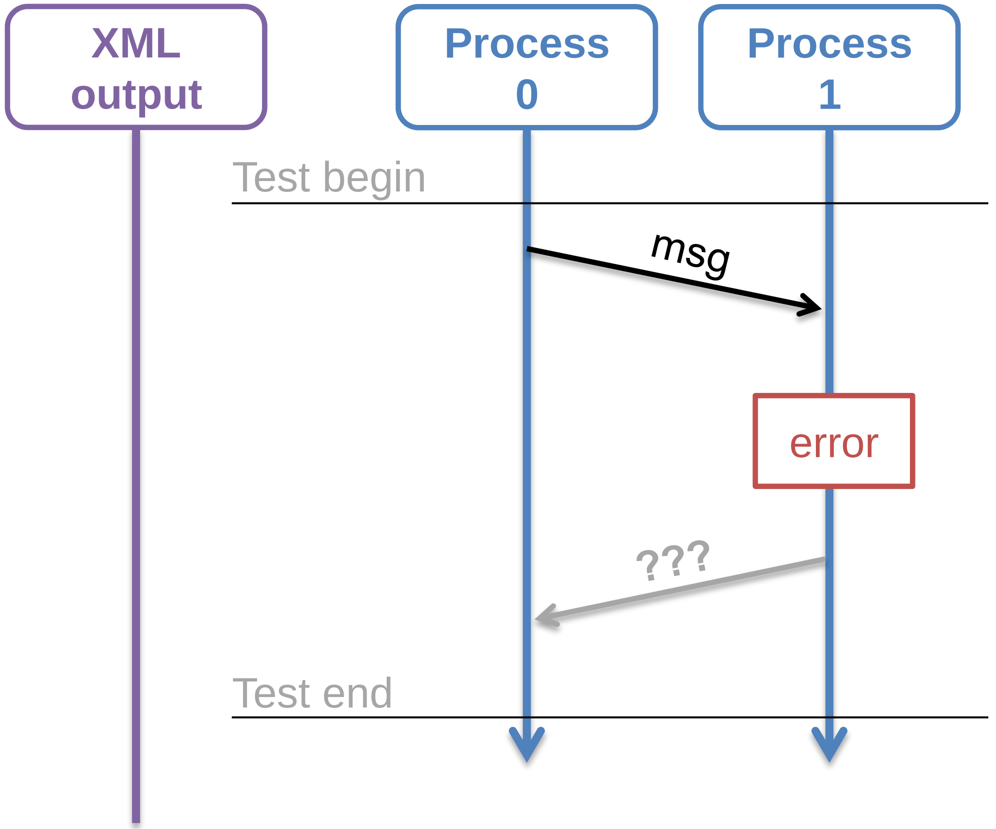


Unit tests: **problems from the wild (4)**

- Setup:
  - parallel unit tests with
  - 2 processes
  - Output on process 0

- Error only on process 1, process 0 waiting

⇒ No output & program does not terminate!

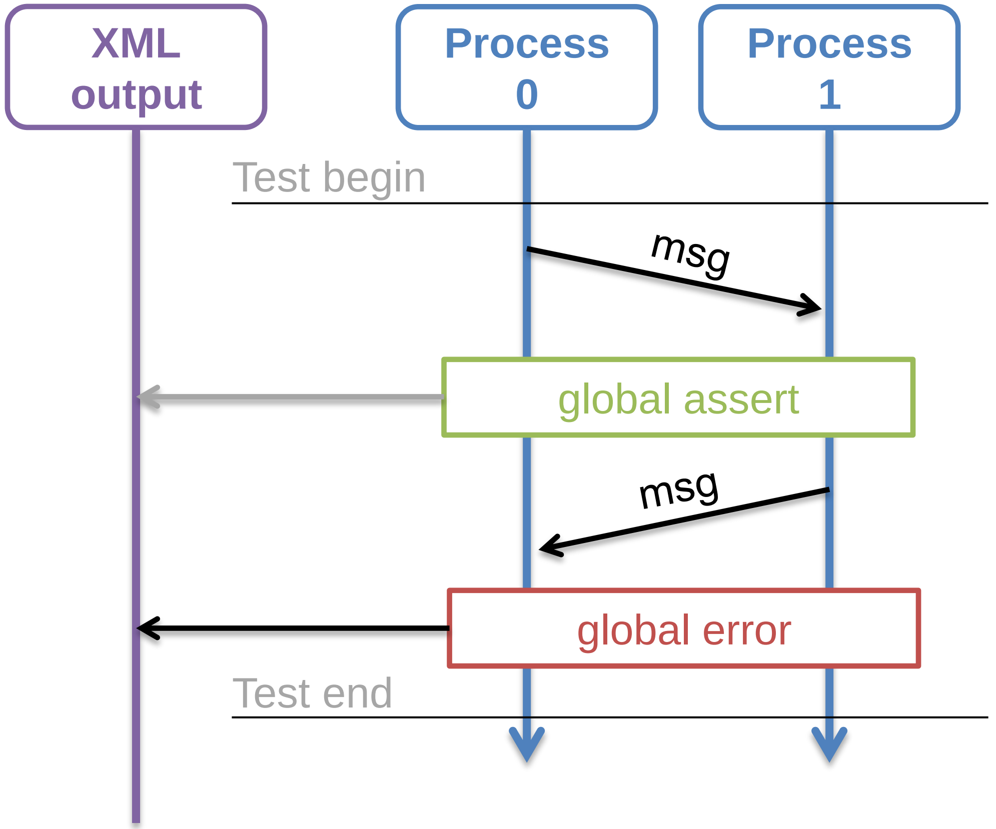


Unit tests: **our solution**

- Setup:
  - parallel unit tests with
  - 2 processes
  - **Global assertions and output**

- Error only on process 1

⇒ **Error reported correctly, program terminates!**



Unit tests: **frameworks**

- For C/C++: **googletest+MPI**
  - Thread-safe, but no multi-threading functions
  - Version with MPI support, e.g. included in P https://bitbucket.org/essex/phist/
  - Open Source (no MPI): https://github.com/google/googletest

- For C/C++: Trilinos package Teuchos
  - Tools package of Trilinos
  - Large library for scientific computing
  - Open Source: https://trilinos.org

- For Fortran: **pFUnit**
  - Supports OpenMP and MPI
  - Developed by the NASA
  - Open Source: http://pfunit.sourceforge.net/

- For Java: Junit
- For Python: PyTest

- Others???

Unit tests: **test setup**

- To detect (all important) bugs:
  - Run tests with different tools
  - Vary number of threads / processes

  ⇒ Drawback: exploding number of combinations

- Limited time / resources:
  - Automation with CI (e.g. Jenkins)
  - Start with simple tests (1 process/thread)
  - Combine tests for “orthogonal” problems

ESSEX Jenkins build jobs

Parallelism

CPU jobs

vary #procs/threads:
1x12, 2x6, 3x4, 12x1

GPU+CPU jobs
variants:
1(GPU),
1(GPU)+1x6(CPU),
1(GPU)+2x4(CPU),
1(GPU)+11x1(CPU)

Tools / flags

Debug mode

Normal tests with
`-fsanitize=address`
(or others)

Release mode

Normal tests +
Larger accuracy tests

+ different backends /
compilers /
optional libraries

Summary: testing parallel code

- Parallel code is complex & **non-deterministic**:
  - Multiple levels of parallelism
  - Different programming models

  ⇒ New **parallel bugs** (data races, deadlocks)

- **Parallel unit tests**:
  - Serial frameworks may lead to more problems.

  ⇒ Tests should support the desired parallelism.

  - Test setup (combine tools and #threads/procs)

- **Tool support** is crucial:
  - Problems not easy to reproduce (in debugger)
  - Tools can help to detect bugs

  ⇒ Choose correct tool(s) for your use case.

- Not covered:
  - more subtle errors like starvation
  - differing results through non-ordered operations

Advanced Topics

Part 2: Testing parallel code

- Levels of parallelism
- New “parallel” bugs
- Tools for specific bugs
- Unit tests
- Conclusion

Part 3: Performance engineering

- CPU architecture
- Performance modeling
- Performance “bugs”
- Finding bottlenecks
- Conclusion

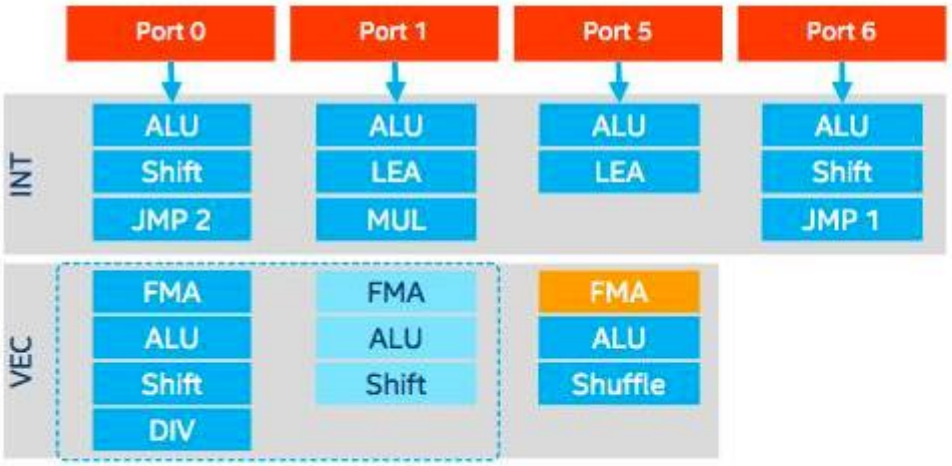
CPU architecture: computing units

- Intel “Skylake” Gold **core**:
  - 2 FMA (fused multiply-add) units
  - **SIMD** width: 512 bit (e.g. AVX512):
    fits 16 single or 8 double precision numbers

  $\Rightarrow 8 \cdot 2 \cdot 2 =$ **32 Flops / cycle** (DP)

  - **Latency: 4 cycles** (FMA/add/sub/mul)
  - Other operations (div, sqrt) are much slower

$\Rightarrow$ Need lots of **independent “multiply-additions”**
(e.g. 128 to fill the pipeline of 1 core)

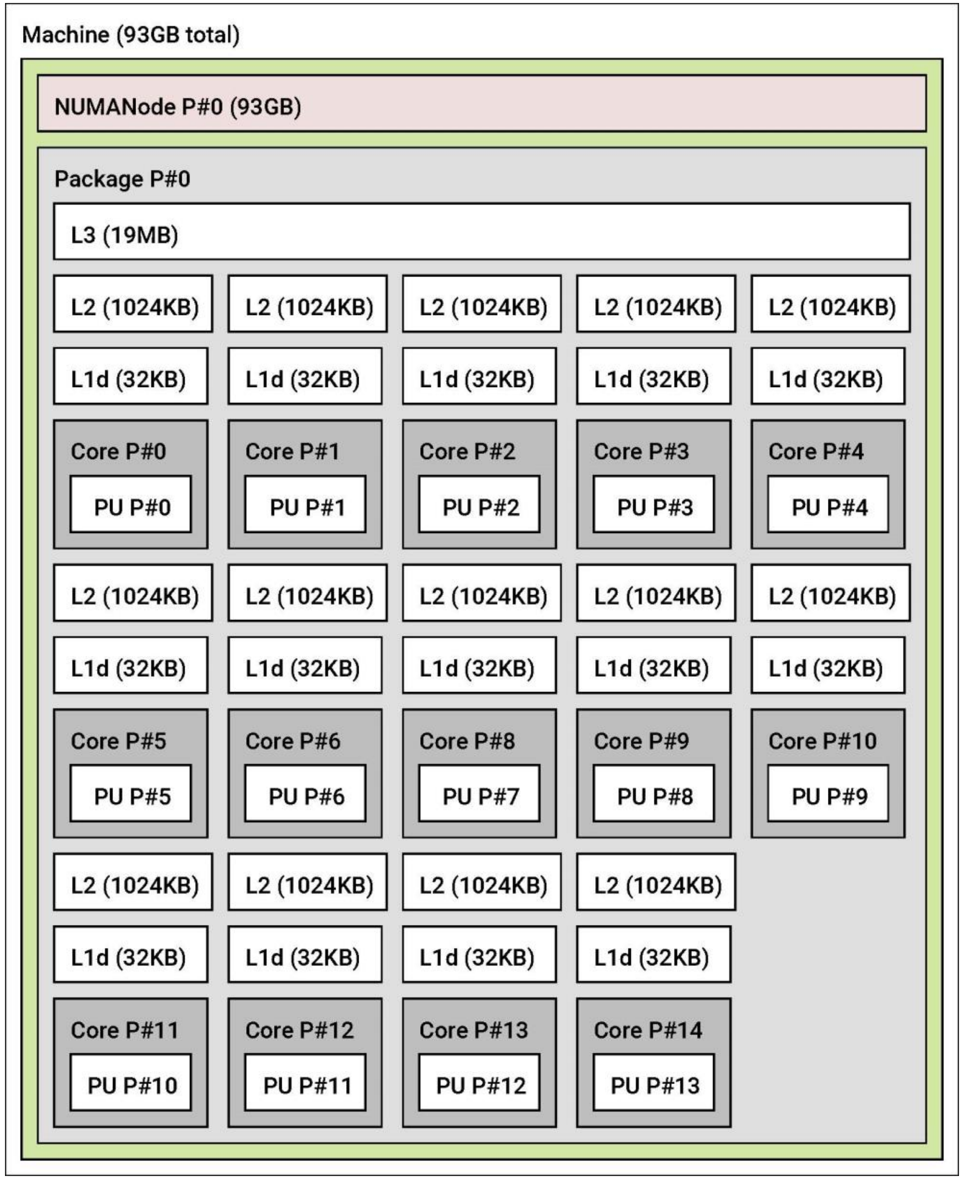


Excerpt from the Intel “Skylake” Gold architecture
source: Intel

CPU architecture: memory hierarchy

- Intel “Skylake” Gold **socket**:
  - 14 cores per socket
  - **3 cache levels** with:
    L1 cache (32kB, 4 cycles latency)
    L2 cache (1MB, 14 cycles latency)
    L3 shared cache (19MB, >50 cycles latency)
  - “Slow” main memory
    (94GB per socket, >400 cycles latency)
  - Caches organized in **lines of 64 bytes** and optimized for “streaming accesses”

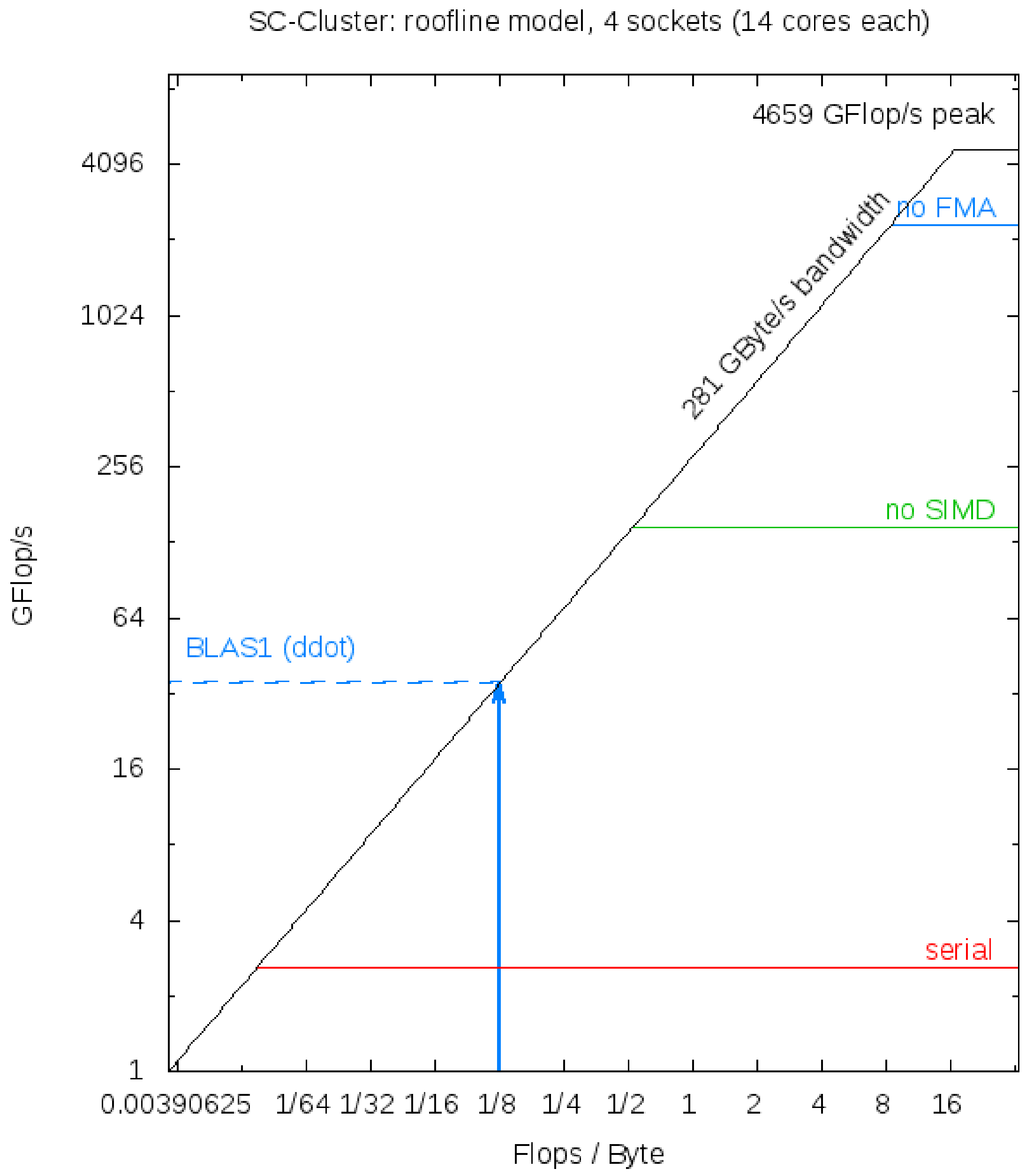
⇒ Need lots of **contiguous accesses** to a **small data set**



Performance modeling: **roofline**

- The **roofline model**
  - applicable **peak performance**: $P_{max}\ \left[\frac{Flop}{s}\right]$
    (of the required operations)
  - **computational intensity**: $I\ \left[\frac{Flop}{byte}\right]$
    ("work" per byte transferred of the algorithm)
  - applicable **peak bandwidth**: $b_s\ \left[\frac{byte}{s}\right]$
    (of the slowest data path utilized)
  - Expected performance: $P = \min(P_{max}, I \cdot b_s)$

⇒ A lot of problems are **memory-bound**
(nice hack: we can do more operations for free)



Performance modeling: **workflow**

1. **Analyze algorithm:**
   - Calculate computational intensity
   - Estimate working set size (does it fit into L3?)
2. **Benchmark**
   - Select relevant operations (FMA or pure add?)
   - Calculate peak performance (CPU family specific)
   - Measure peak bandwidth (system specific)

General remarks:

- works well for “simple” computational kernels
- assumes the problem is big/parallel enough
- Predictions are almost 100% accurate for large contiguous main memory accesses
- Non-contiguous accesses have overhead (e.g. consider cache lines and cache misses)
- It’s hard to tune code to obtain $\geq 10\%$ peak...

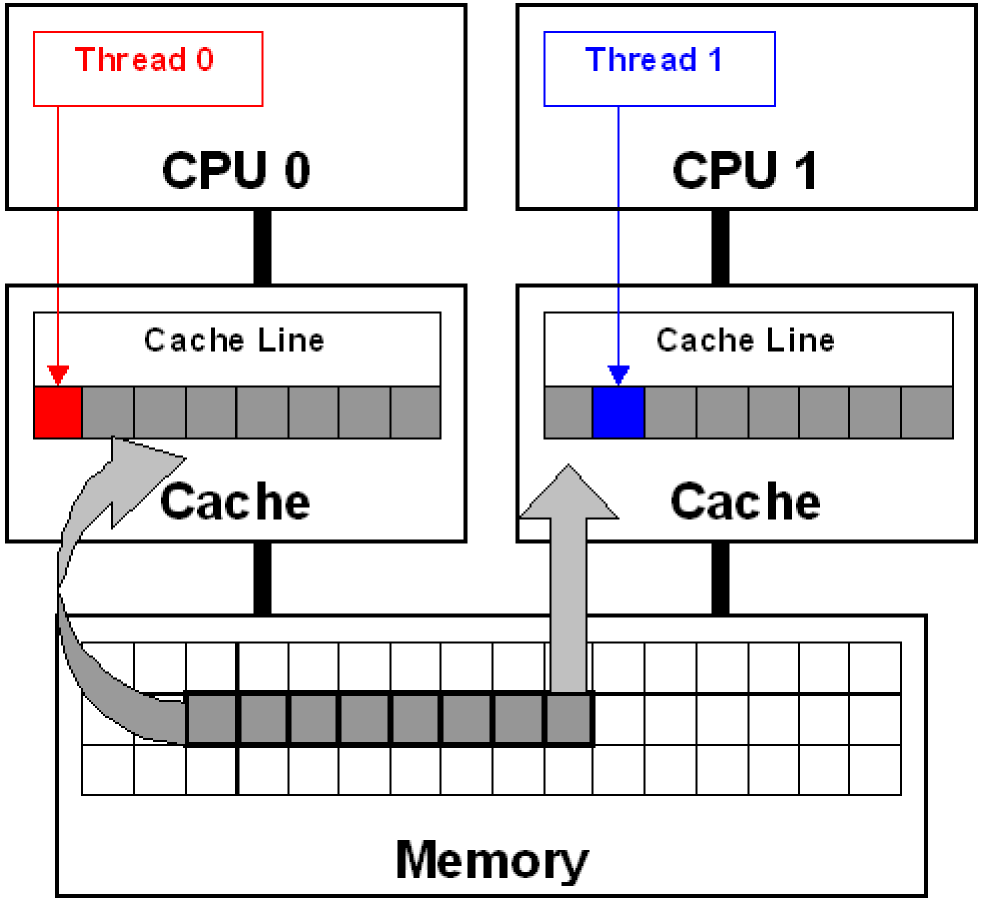
⇒ **Goal: Hit the right bottleneck!**
(and publish that your code is as fast as it gets)

Performance “bugs”: false sharing

- Scenario:
  - Cache line modified by threads on multiple cores
    (e.g. different elements in a small chunk of 64b)
  - System must guarantee cache coherence
  - Code completely correct – no data race, etc.

  ⇒ Behavior:
  Cache line written to main memory and reloaded

- Mitigation:
  - Work on **local data** where possible
  - Avoid `array[nThreads]`, add **padding** to 64b
    (e.g. in C: `double array[8][nThreads];`)

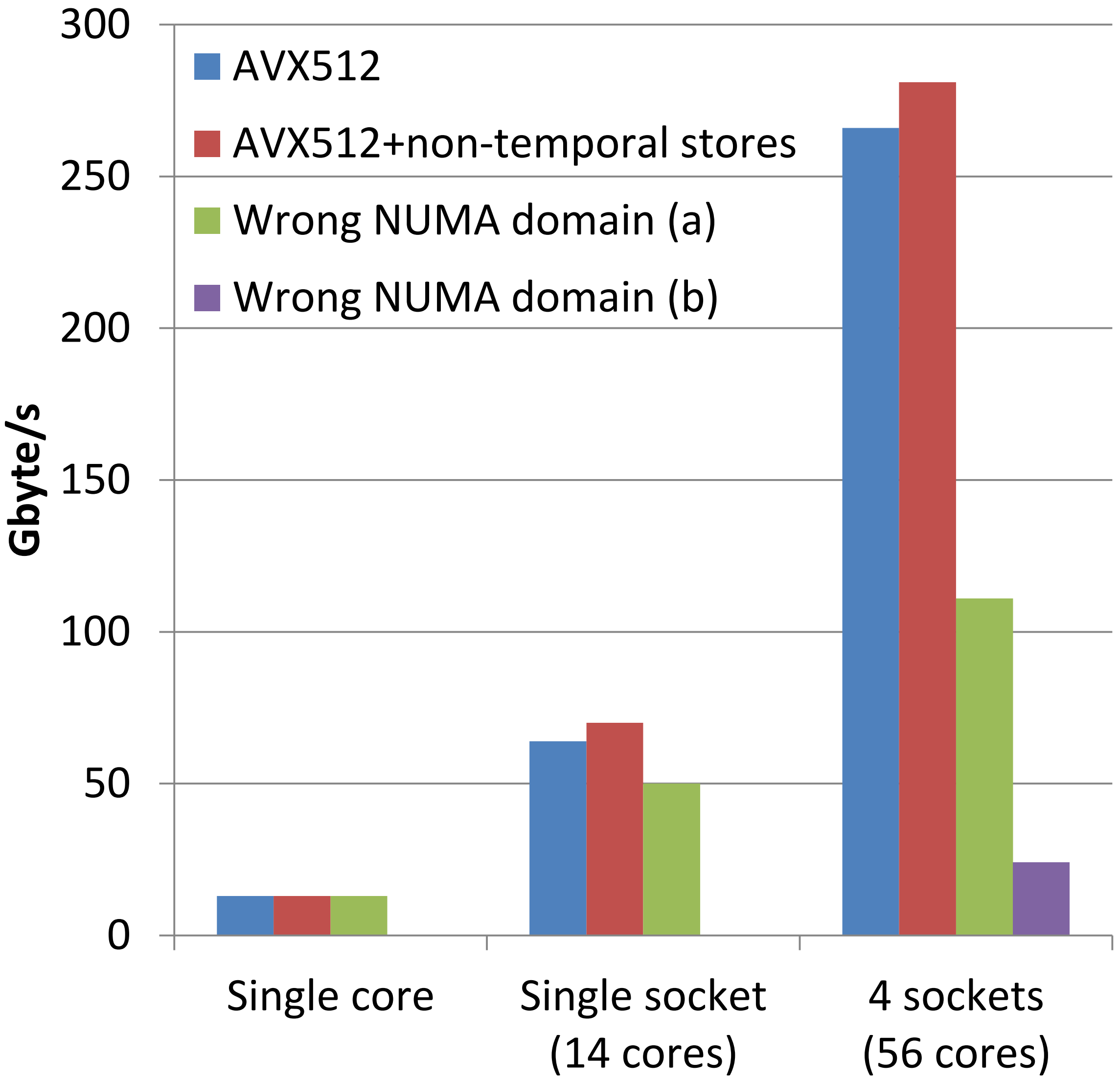


Source: Intel

Performance “bugs”: NUMA effects

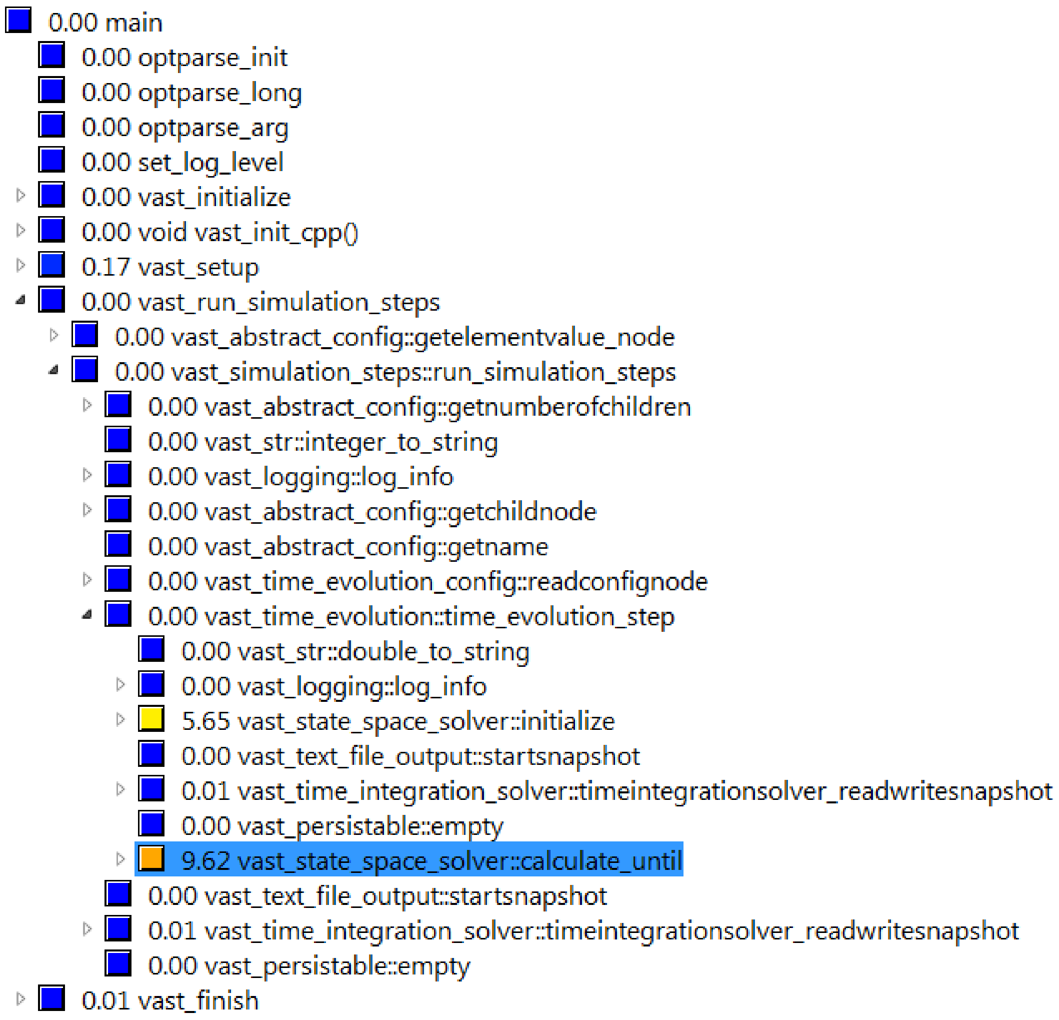
- NUMA (non-unified memory access):
  - Faster/slower access to different memory parts
  - Systems with multiple CPU sockets
    (each socket has its own memory banks)
  - Some AMD CPUs
    (NUMA in a single socket)

- Mitigation:
  1. **Pinning**: bind processes and threads to cores
  2. **First-touch policy**: memory belongs to the NUMA domain that uses it first. (not trivial!)



Finding bottlenecks: **measuring with Score-P (1)**

- Tool to measure performance:
  - Compiler wrapper for C/C++/Fortran
  - Nice and easy-to-use
  - Supports multi-threading & -processing (OpenMP and MPI)
  - Useful for a fast & rough overview
  - Open Source: http://www.vi-hps.org/projects/score-p/
- Basis for more advanced tools: Scalasca, Vampir …



Finding bottlenecks: measuring with Score-P (2)

- Workflow:
  - Instrument compiler with ScoreP wrapper
    (e.g. `CXX=scorep-g++ cmake <path>`)
  - Run test case
  - Investigate measurement overhead
    (using `scorep-score`)
  - Filter out small functions
    (`SCOREP_FILTERING_FILE`, simple text format)
  - Rerun test case…

  ⇒ **Ensure same runtime as without ScoreP**

- **Hardware counters**:
  - CPU measures itself!
  - Available in ScoreP through PAPI
    Open Source: http://icl.utk.edu/papi/
  - Real run-time data per function about
    Operations, cache accesses, …
  - Interesting points:
    - Vectorized (SIMD) vs. non-SIMD FP ops
    - Achieved memory bandwidth
    - Cache misses
  - However: not all CPUs provide correct results
    (tool will usually not provide counters then)

Summary: performance engineering

- Know your architecture:
  - **SIMD** operations
  - Memory / **cache hierarchy**

  ⇒ Ideally: lots of similar operations on small data

- Setup a model:
  - Simple model of algorithm + hardware
  - Compare actual & predicted runtime

  ⇒ Goal: **hit the right bottleneck**
  Better understanding

- Avoid pitfalls like false sharing, NUMA, …
- Use tools for timings and hardware counters
- Read a book:
  Hager & Wellein: "Introduction to High Performance Computing for Scientists and Engineers", 2018
- Practical observations:
  - Optimized vs. normal code: factor >100
  - Problems: vectorization, temporary objects, …

- If it’s not in the repo, it doesn’t exist
- If it’s not tested, it will break
- Parallel programs are not deterministic and need a specialized test framework
- Scalability alone is not a good measure of parallel code performance - **%roofline** is

Happy to answer any remaining questions now or later:
Jonas.Thies@DLR.de