

Foundations of High Performance Computing

Code optimization

Part I - There's no spoon

Luca Tornatore – I.N.A.F

Premature optimization is the root of all evil

D. Knuth

The rules of optimization club:

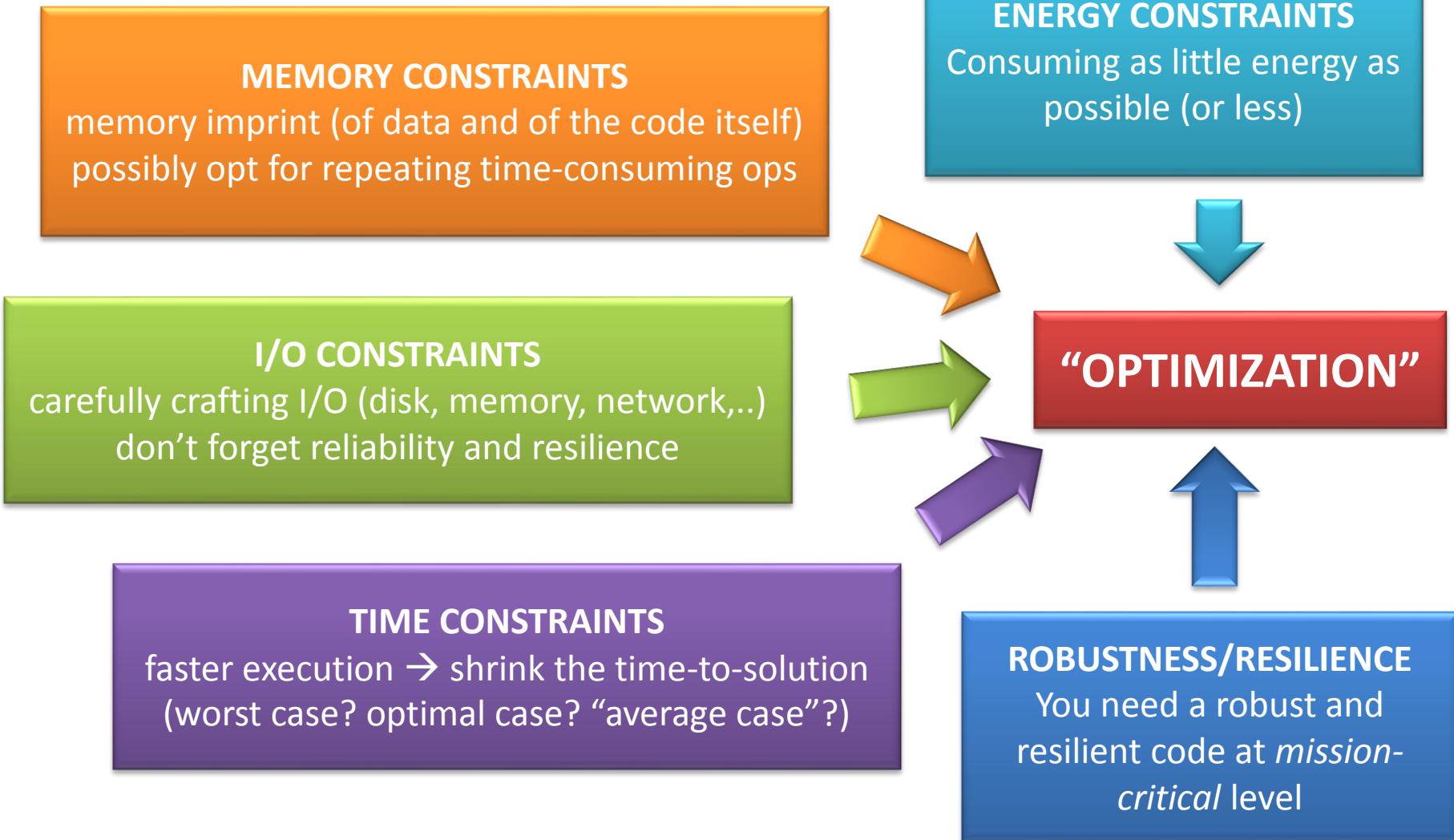
1. Don't talk about optimization
2. DON'T TALK ABOUT OPTIMIZATION yet
3. If your code yells stop, goes limp, taps out.. take a break
4. Only one code at a time
5. [.. *find something for “no shirts no shoes”*]

A misnomered concept

There is no such thing as an “optimal code” tout court

Just try to give some example of what “optimal” is..

A misnomered concept



A misnomered concept

MEMORY CONSTRAINTS

memory imprint (of data and of the code itself)
possibly opt for repeating time-consuming ops

ENERGY CONSTRAINTS

Consuming as little energy as possible (or less)

Most of the time, you will have to cope with a variable combinations of ALL the previous factors.

(satellites' software is a good example)

(worst case? optimal case? average case?)

You need a robust and resilient code at *mission-critical* level

A misnomered concept

Then, there is no such thing as an “optimal code” tout court

Probably you’d better think in terms of “*improved*” code.

- You don’t want to hurt your code that honestly does its job: a faster code that gives **incorrect ans.** is not optimal in *any* way
- You most probably will have to **re-use** that code after some time. And you will need it to be **readable**.
- **Other people** most probably will have to read, understand and re-use that code.
- A **clean, non-obscure, understandable, non-redundant** code may not be “improved” in some way, but actually improves the quality of your life (and of the others’ life).

So, why you may want to improve your code?

- 1) To get a cleaner code**
- 2) To re-design (some of) its fundamental architecture (modules)**
- 3) To improve the workflow, the memory imprint, the resource usage, the time-to-solution, ...**

So, why you may want to improve your code?

- 1) To get a cleaner code**
- 2) To re-design (some of) its fundamental architecture (modules)**
- 3) To improve the workflow, the memory imprint, the resource usage, the time-to-solution, ...**

Be DRY: Don't Repeat Yourself

Duplication in logic

→ *abstraction* is the cure

Duplication in process

→ *automation* is the cure

Do neither add **unnecessary** code nor **duplicated** code

Unnecessary code increases the amount of needed work

- to maintain the code, either debugging or updating it
- to extend its functionalities

Duplicated code increases your *bad technical debt*, that already has a large enough number of sources:

- copy-and-paste programming style
- too much *agile* approach
- hard coding, quick-and-dirty fixes, cargo-cult, sloppiness

Readability is a pillar of maintainability.

Maintainability means that software can be extended, upgraded, debugged, fixed.

- **Baseline:** *write those damn comments*
- **Advanced:** *WRITE THOSE DAMN COMMENTS !*
(possibly, use doxygen-like tools)
- **X-advanced:** avoid stupid, obvious, useless, confusing comments

→ Keep in mind: “*the code is the ultimate man page*”

- Use **meaningful and coherent names** and name modifiers inside functions and code blocks.
Don't stick to the horrible fortran-like habit of unintelligible combinations of letters and digits: var17u doesn't mean nothing after few hours you wrote it.
- Use **meaningful and coherent conventions** to name **code blocks** and to individuate the block a function belongs to.
- Clearly **divide your code in different source files**
that is unrelated to design: you may have a wonderful design awfully written in a unique 100k-lines source file.
- Adopt a **coherent and consistent formatting**
use in a logic way indentation, blank lines, comment blocks, vertical spaces.

A cleaner code | readability

- Use readable variable names
 - Use readable functions
 - Use readable Recipes' style and types
 - Clear comments
Or something else like
 - `l = l + 1`
 - `This = that + 0`
 - Adopt a consistent style, use indentation, vertical spaces.
- The gods cry when you use the Numerical Recipes' style
- QRV17_N

inside

tions
I wrote it.

locks

written in a

s,

1. Easier to understand even after months, or by someone else
2. Easier to catch side-effects of any changes
3. Easier to debug
4. Finding hot-spots is easier as well
 - In memory footprints
 - In I/O and communications
 - In instruction intensity

Doesn't look like as “optimal” ?

So, why you may want to improve your code?

- 1) To have a cleaner code**
- 2) To re-design its fundamental structure**
- 3) To improve the workflow, the memory imprint,
the resource usage, the time-to-solution**

If you want to re-design the whole code, you're probably writing a new code from scratch.

However, that is useful to stress a **fundamental point**:

optimization starts from software's design

Sounds wonderful, isn't it? (a bit hipsteric, perhaps)

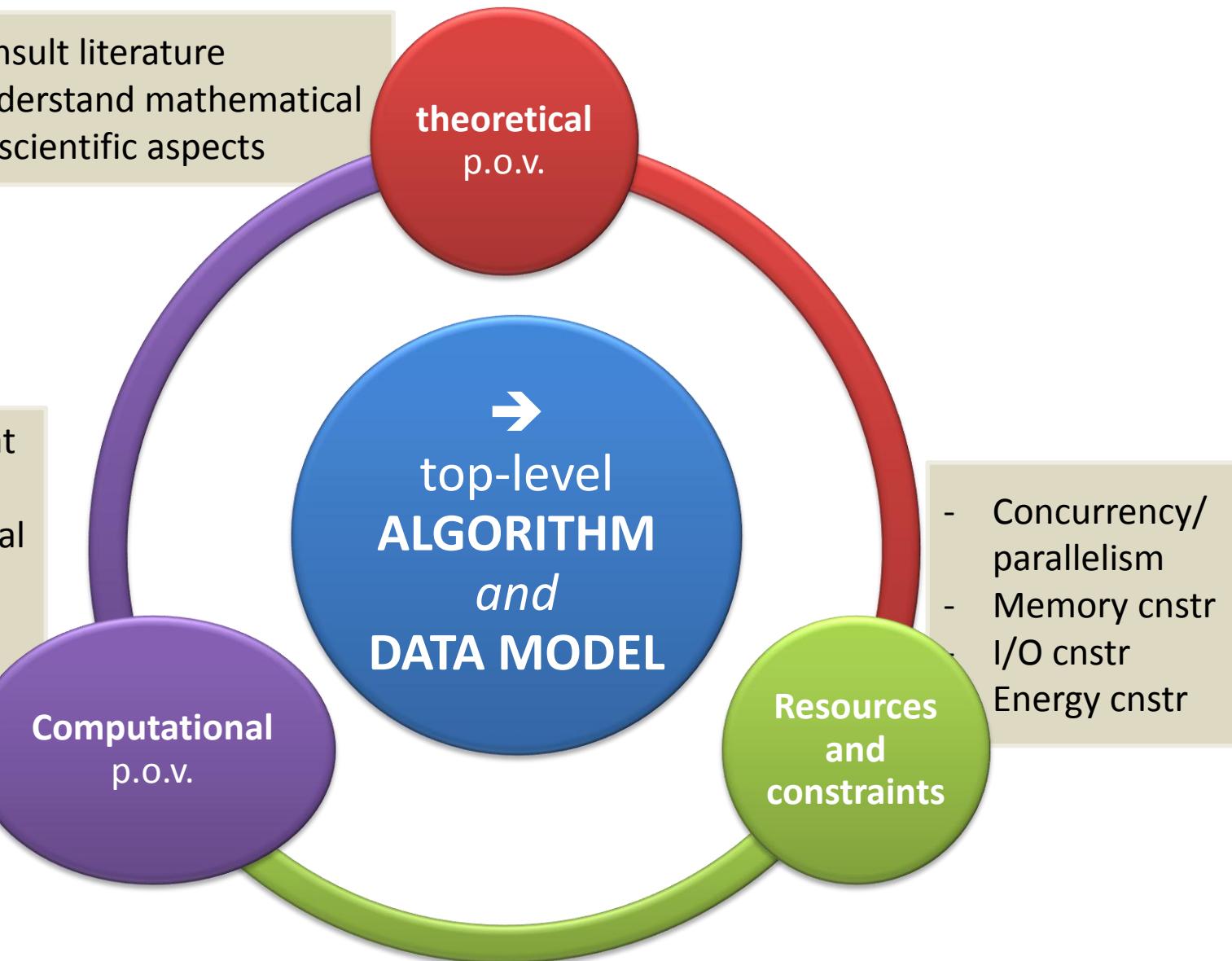
But: what is “design” ?

What is software design, *for us*

- Consult literature
- Understand mathematical and scientific aspects

Well, you're not at the blackboard:
you are on a digital discrete system..

(CFD is a good example of the concept)



What is software design, *for us*

ARCHITECTURAL
DESIGN

Highest abstract picture:
- Interacting multi-components
- Top-level DATA MODEL

HIGH-LEVEL
DESIGN

Sub-systems and
modules
Communication
strategies

DETAILED
DESIGN

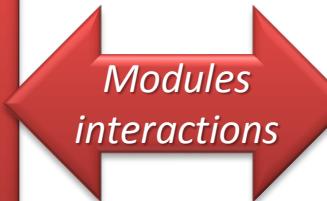
Implementation details
- many algorithms
- concurrency

Logical
structure
of
modules

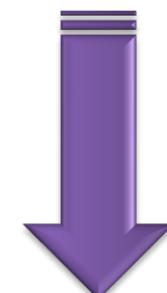
interactions

Very low-level
coding and details

ABSTRACTION



IMPLEMENTATION
CODING DETAIL



What is software design, *for us*

MODULARIZATION

- On a functional basis, divide your problem in several coherent **sub-problems of smaller complexity**.
- Individuate **relations** among those sub-problems.
- Think in terms of **modules**, i.e. independent “objects” (or functions, in older words) able to **independently carry on different tasks**.
 - abstraction is easier
 - easier to maintain
 - re-usable
 - can be made concurrent
 - resiliency is enhanced

CONCURRENCY

Different modules (tasks) can **run concurrently** either on the same resources (CPUs/cores) or on different resources (cores / FPGA / GPUS / ...).

They can be **active at different times** along the run.

They **carry dependencies and conflicts** with other modules (tasks).

So, why you may want to improve your code?

1) To have a cleaner code

2) To re-design its fundamental structure

**3) To improve the workflow, the memory imprint,
the resource usage, the time-to-solution**

*...well, that is actually covered in the future
lectures...*

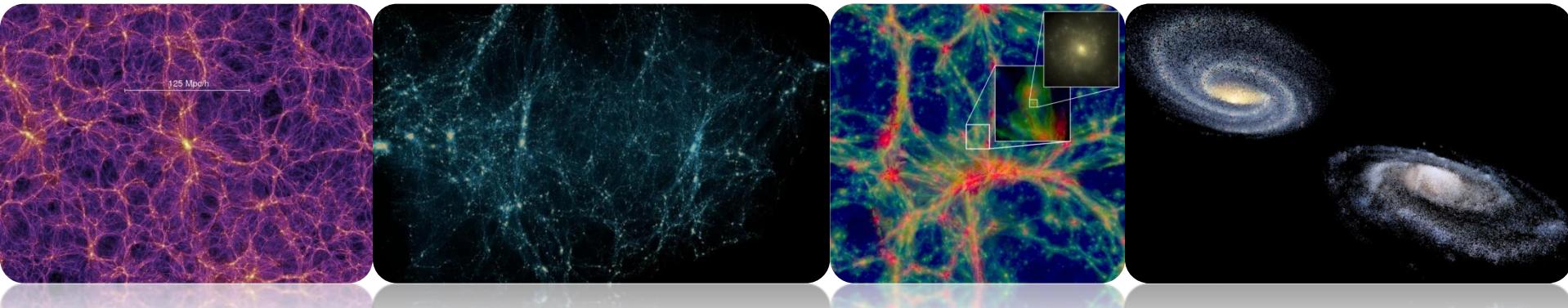
TESTING IS part of the design

- Unit test
separately stress each unit of the code
- Integration test
stress the integrated behavior of all the units together
- System test

VALIDATION and **VERIFICATION** ARE part of the design

- Validation ensures that the code does what it was meant to do
(all modules are designed accordingly to design specifications)
- Verification ensures that the codes does what it does *correctly*
(black-box testing against test-cases, ...)

A sketched example: cosmological simulations



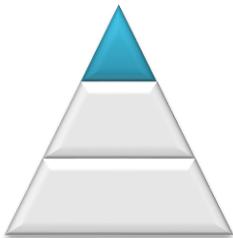
Simulation, in a cosmological framework, of the formation of

- The **cosmic structures** – LSS, galaxy clusters and galaxies.
- The **astrophysical processes** that drive their evolution and interactions.

With:

- **DM, DE and Baryons.**
- **Gravity (long-range) + a variety of interactions** at medium- and short-ranges.

A sketched example: cosmological simulations



Architecture – highest level of abstraction –

Data Model

- arrays of structures vs structures of arrays
- what data are needed (← physics modules) / how to make it easy to add data (i.e. physics modules)
- some data (position, velocity, acceleration, ID, ...) common among modules

Execution

Massively Parallel is mandatory.

What paradigm? MPI, MPI+OpenMP/pThreads, PGAS, ...

Concurrency

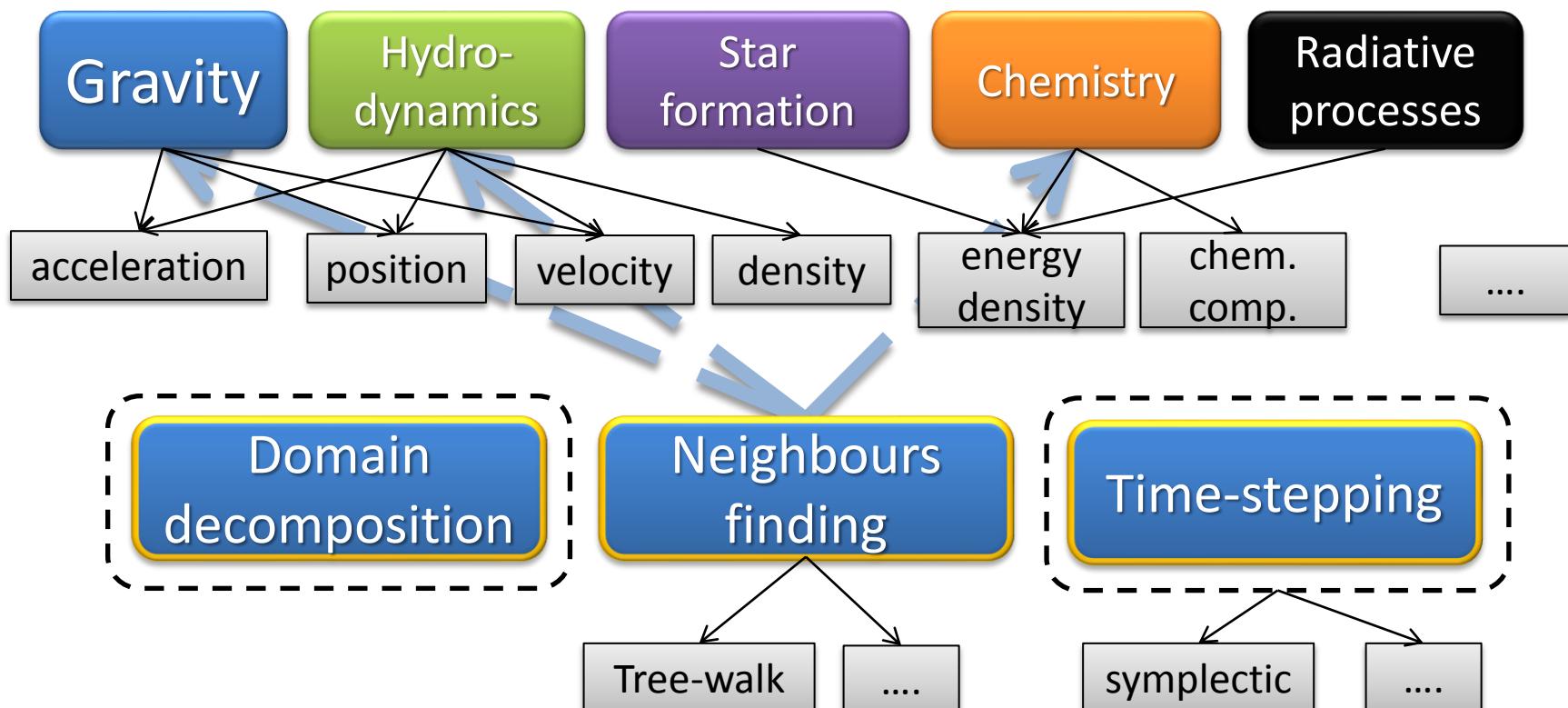
Task based, fine granularity ? ← easily to substitute physical modules

Rigidly procedural ?

A sketched example: cosmological simulations



Modules – components design (*)



(*) I should have used Data Flow Diagram, Structure Charts, UML, ...



Implementation – Details

Gravity

- Direct summation
- Pure PM
- TreePM
- Fast Multipole methods
- ...

Hydro-dynamics

- Eulerian
 - multigrid
- SPH
- Mesh-free
-

...

-

Sequential

--

--

--

--

--

Vs.

