The upcoming wall of software complexity in computational sciences

Journées Nationales du GDR GPL – June 2022 – Vannes, France

Vincent Reverdy, CNRS IN2P3/INS2I, Laboratoire d'Annecy de Physique des Particules
From nice drawings on a blackboard...
... to unmaintainable monsters
Expectation vs reality

**Expectation**

- EVERYTHING IS CONNECTED
- EVERYTHING IS BEAUTIFUL

**Reality**

- OH DEAR GOD NO
1. Introduction

An introductory tale

2. Problem

Framing the problem of software complexity

3. Framework

A practical guiding framework

4. Performance

Exploring performance concerns

5. Genericity

Exploring genericity and abstraction strategies

6. Expressivity

Exploring expressivity and DSLs

7. Conclusions

Facing the wall of software complexity
Let's start with a story

Once upon a time...

...in a galaxy far far away...
Let's start with a story

...on a small piece of rock...

...wandering aimlessly in a vast Universe...
Let's start with a story

...a team of astrophysicists was wondering about the nature of life, the Universe, and everything.
Let’s start with a story

Why do galaxies form a cosmic web?

Let’s run simulations to better understand where it comes from!
A crash course in astrophysics simulations

They said: "Let's take an enormous box..."
A crash course in astrophysics simulations

...with periodic boundary conditions...

(3D torus)
A crash course in astrophysics simulations

...and let's fill that enormous box with particles weighing the mass of millions of suns...

(note: yes that's kind of huge)
A crash course in astrophysics simulations

Now, divide the box in cells using a regular grid and apply the following recipe:
A crash course in astrophysics simulations

1. Introduction

- 1) For each cell $c$ containing particles with position $\vec{x}_i^c$ and velocity $\vec{v}_i^c$

- 2) Interpolate density $\rho$ in cell $c$ depending on surrounding particles

- 3) From $\rho$ compute the gravitational potential $\Phi$

- 4) From $\Phi$ interpolate back the acceleration $\vec{a}$ at position $\vec{x}_i^c$

- 5) From $\vec{a}$ compute the new speed $\vec{v}_j$ of each particle

- 6) From $\vec{v}_j$ compute the new position $\vec{x}_j$ of each particle

7. Conclusions
A crash course in astrophysics simulations

Using this recipe with millions of particles we can simulate galaxy formation!
From galaxies to expanding the Universe

Simulating galaxies is nice...

...but simulating the expansion of the Universe requires to take the approach to a whole new level...
First they took a supercomputer.
From galaxies to expanding the Universe

Second they made the box expand as the Universe does according to General Relativity (considering a homogeneous FLRW metric)
Third, they filled the box with billions of particles with the same statistical distribution as the matter in the primordial Universe.
Fourth, they updated their algorithm using an Adaptive Mesh Refinement (AMR) strategy to increase resolution in regions of interest.
And after all this work this is what they obtained:
From galaxies to expanding the Universe

...and they lived happily ever after...

...except for one tiny annoying detail...
A tiny annoying detail about General Relativity

\[ G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} \]

Wait, what about General Relativity?

space-time geometry

energy-matter contents
A tiny annoying detail about General Relativity

In cosmological simulations the space-time geometry evolution is precomputed...

...that means no dynamic backreaction of the contents on the geometry
A tiny annoying detail about General Relativity

It's classical physics in a pre-computed expanding background

Is that even correct?
A tiny annoying detail about General Relativity

Why no fully relativistic simulations?

Because...

1. There is not enough computing power
2. Even if there is, it’s not possible algorithmically
3. Ok, maybe... but in any case it’s not interesting
Because no-one really knows how to write such a code...

Numerical cosmology

Large scale
Billions of particles
Newtonian gravity
Adaptive Mesh Refinement
Multigrid methods
Space-filling curves
Millions of computing hours

Numerical relativity

"Small" scale
Few bodies
General relativity
Fixed-grids
Spectral methods
Non-trivial initial conditions

Two domains with uncomposable complex codes!
The untold truth

...and no-one really realizes it...

Is it a research problem?

YES

Is it a physics problem?

YES

PHYSICS SOLUTIONS

NO

Is it a computational problem?

YES

COMPUTATIONAL SOLUTIONS

NO

It's a technical problem

TECHNICAL SOLUTIONS (NOT OUR BUSINESS) (MAGIC HAPPENS HERE)
Programs = Code = Technical artifacts

For the most part,

in computational sciences,

the structural complexity of programs

is an unthought

There is no solution to be found
to a problem that does not exist
1. Introduction
   An introductory tale

2. Problem
   Framing the problem of software complexity

3. Framework
   A practical guiding framework

4. Performance
   Exploring performance concerns

5. Genericity
   Exploring genericity and abstraction strategies

6. Expressivity
   Exploring expressivity and DSLs

7. Conclusions
   Facing the wall of software complexity
Most physics codes are built from the same categories of components.
Combining individual components

1. Introduction
2. Problem
3. Framework
4. Performance
5. Genericity
6. Expressivity
7. Conclusions

**Physics**
- Gravitation
- Electromagnetism
- Optics
- Hydrodynamics
- Spheric
- 2D grid
- 3D box
- Torus

**Topology & Geometry**
- Gravitation
- Electromagnetism
- Optics
- Hydrodynamics
- Spheric
- 2D grid
- 3D box
- Torus

**Hardware architectures**
- X64
- GPU
- FPGA
- TPU

**Algorithms & Numerical methods**
- Neural networks
- Finite volumes
- Finite differences
- Spectral
- Distribution
- MPI

**Data & Data structures**
- Graph
- Tree
- Map
- Array

**Parallelism & Concurrency**
- Scheduler
- SIMD
- Multithreading

**Combining individual components**

1. A relativity code in spherical coordinates
2. A hydro code in a 3D cubic box
3. Mixing components of both
4. Adding more software components
Combining individual components

1. Introduction
2. Problem
3. Framework
4. Performance
5. Genericity
6. Expressivity
7. Conclusions
A combinatorial explosion of complexity
Hitting the wall of complexity

The wall of complexity

\[ \prod_{i=0}^{n} |C_i| \]

Amount of scientific code components

The wall of complexity

Exascale?

Tipping point

Written codes

Uncharted territories

Researchers production

Missed opportunities of scientific discovery

Time

\[ \prod_{i=0}^{n} |C_i| \]
Monodisciplinary approaches are not enough

The astrophysics approach
- Computer science = given
- Simplifying physics/models
- More physics for same level of complexity

The computer science approach
- Astrophysics = given
- Better algorithms/data structures/performances
- More computing for same level of complexity
Attacking combinatorial explosion as a proper scientific problem

- Treating the root cause instead of the symptom:
  - Technical symptom $\Rightarrow$ code
  - Root cause $\Rightarrow$ compositionality / combinatorial explosion

- Interdisciplinary by nature:
  - Emergent complexity
  - Composition of physics and computer science components
2. Problem

Framing the problem

\[ \prod_{i=0}^{n} |C_i| \Rightarrow \sum_{i=0}^{n} |C_i| \] (in first approximation)

Structural complexity of programs

- Analogy with algorithmic complexity but on the structure of programs itself

A possible angle of attack

- Software architecture
- Programming languages
- Compilers
- ...
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Subtitle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>An introductory tale</td>
<td></td>
</tr>
<tr>
<td>2. Problem</td>
<td>Framing the problem of software complexity</td>
<td></td>
</tr>
<tr>
<td>3. Framework</td>
<td>A practical guiding framework</td>
<td></td>
</tr>
<tr>
<td>4. Performance</td>
<td>Exploring performance concerns</td>
<td></td>
</tr>
<tr>
<td>5. Genericity</td>
<td>Exploring genericity and abstraction strategies</td>
<td></td>
</tr>
<tr>
<td>6. Expressivity</td>
<td>Exploring expressivity and DSLs</td>
<td></td>
</tr>
<tr>
<td>7. Conclusions</td>
<td>Facing the wall of software complexity</td>
<td></td>
</tr>
</tbody>
</table>
### Practical design principles

#### Handling software complexity
- Generally guided by practical development principles
- Not coming from theoretical proofs

<table>
<thead>
<tr>
<th>Design patterns</th>
<th>Coding principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational patterns</td>
<td>Liskov substitution principle</td>
</tr>
<tr>
<td>Structural patterns</td>
<td>Law of Demeter</td>
</tr>
<tr>
<td>Behavioral patterns</td>
<td>Composition over inheritance</td>
</tr>
<tr>
<td>Concurrency patterns</td>
<td>Rule of three</td>
</tr>
<tr>
<td>Functional patterns</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tools</th>
<th>Development strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit tests</td>
<td>Lean development</td>
</tr>
<tr>
<td>Autocompletion</td>
<td>DevOps</td>
</tr>
<tr>
<td>Static analysis</td>
<td>Agile</td>
</tr>
<tr>
<td></td>
<td>SCRUM</td>
</tr>
</tbody>
</table>
A guiding metric of good vs bad for abstraction: the GPE triad

Genericity

Performance

Expressivity
Approaching abstraction through genericity, performance, and expressivity
A converging iterative process

Expectation

Reality

Aim

Genericity

Performance

Expressivity
Multidimensional arrays as a canonical example: why still no nd-arrays in C++23?

In Python
- Numpy arrays

In C++: performance concerns
- At least BLAS performance
- Exploiting SIMD
- Memory footprint
- Parallelizability

In C++: generality concerns
- Iterator and data types
- Access patterns
- Symmetries
- Allocator

In C++: expressivity concerns
- Terse syntax for most cases
- Full syntax for expert users
- Options passing
Exploring performance concerns

1. Introduction
   - An introductory tale

2. Problem
   - Framing the problem of software complexity

3. Framework
   - A practical guiding framework

4. Performance
   - Exploring performance concerns

5. Genericity
   - Exploring genericity and abstraction strategies

6. Expressivity
   - Exploring expressivity and DSLs

7. Conclusions
   - Facing the wall of software complexity
Ensuring best possible performances

Several possible approaches

- New languages
- New compilers
- Preprocessors and code generators
- Metaprogramming and active libraries

Active libraries

- Feedback of the user code on the library
- C++ template metaprogramming
- Generative programming
- Compile-time execution

Embedded Domain Specific Languages

- DSL within a host language (C++)
4. Performance

Benchmark of standard algorithms on `vector<bool>` vs their `bit_iterator` specialization (logarithmic scale)

Average time for 100 benchmarks with a vector size of 100,000,000 bits

<table>
<thead>
<tr>
<th>Name</th>
<th>vector&lt;bool&gt;</th>
<th>bit_iterator&lt;<code>uint64_t</code>&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_of</td>
<td>114x</td>
<td>3359x</td>
</tr>
<tr>
<td>count</td>
<td>86x</td>
<td>300x</td>
</tr>
<tr>
<td>search</td>
<td>1906x</td>
<td>116x</td>
</tr>
<tr>
<td>copy</td>
<td>522x</td>
<td>113x</td>
</tr>
<tr>
<td>fill</td>
<td>153x</td>
<td></td>
</tr>
<tr>
<td>swap_ranges</td>
<td>461x</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>334x</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>31x</td>
<td></td>
</tr>
<tr>
<td>rotate</td>
<td>389x</td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inplace_merge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_permutation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>accumulate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel Core i7-2630QM @ 2.00GHz
High-performance computational sciences when software complexity is the bottleneck

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
</table>
| - Combinatorial explosion of complexity  
- Low-level optimization opportunities | - Pure performance still grows exponentially  
- Explosion of optimization opportunities |

### New bottlenecks

- Development time  
- Human resources

### Not bottlenecks anymore

- Hardware capabilities  
- Pure performance

### Consequence

- Software always lags far behind hardware

### In first-order approximation

- Computational power can be considered as infinite at time of development
1. Introduction: An introductory tale
2. Problem: Framing the problem of software complexity
3. Framework: A practical guiding framework
4. Performance: Exploring performance concerns
5. Genericity: Exploring genericity and abstraction strategies
6. Expressivity: Exploring expressivity and DSLs
7. Conclusions: Facing the wall of software complexity
The root of all evil

Components ≠ Vector

\[ \begin{pmatrix} 3.5 \\ -1.2 \\ 0.9 \end{pmatrix} \]

In numerical physics

\[ \vec{v} \]

In maths

\( \vec{v} \): independence from change of basis
Amplification of conceptual approximations

Conceptual approximations get amplified through higher layers of abstractions

- "Almost right" can quickly transform into "Totally wrong"
Top-down vs bottom-up approaches

- **Concepts**
  - Big-picture
  - General
  - Top
  - Down
  - Details

- **Abstractions**
  - Up
  - Bottom
  - Applications
  - Counter-examples
  - Use cases

Better for software implementation

Better for software architecture
Bottom-up approaches tend to work better to find the right abstractions

**Bottom-up approach**
- Accumulate concrete examples first
- Let abstractions emerge from details

**Programming languages vs human languages**
- Human concepts ≠ Computer concepts
- Human languages are fuzzy by nature
- Programming languages need rigorous definitions
Constraining abstractions from use cases: mapping the design space

Looking for all possible constraints
- More use cases → More constraints on abstractions
- Starting with everything one may want
- Looking for the weirdest applications
- Finding boundaries

Remove constraints one by one
- Some use cases add more constraints than others
- Start by removing corner cases that add strong constraints

Software architecture is not about what one can have it's about **deciding** what one **cannot** have
Concept-based programming

- Allow to define mathematical classes of types

Object Oriented Programming
- Monolithic type hierarchies
- Context-independent hierarchies
- Top-down approach

Concept-based Programming
- Named sets of constraints
- Context-dependent constraints
- Bottom-up approach

\[ x: T \quad x \rightarrow \sqrt{x} \]  
\[ T \text{ should be a number} \]

\[ v: T, i: U \quad (v, i) \rightarrow v[i] \]  
\[ T \text{ should be a container} \]
\[ U \text{ should be an integer} \]
Introduction

An introductory tale

Problem

Framing the problem of software complexity

Framework

A practical guiding framework

Performance

Exploring performance concerns

Genericity

Exploring genericity and abstraction strategies

Expressivity

Exploring expressivity and DSLs

Conclusions

Facing the wall of software complexity
Defining expressivity

### Practical definition

- Ease-of-use
- Clear
- Concise
- Precise
- Accurate

### Formal definition

Symbolic calculus in C++

Symbolic calculus with matrices

```cpp
int main(int argc, char* argv[]) {
    // Defining 2D dynamic array type
    using matrix = decltype(ndarray<double, shape()()>);

    // Defining symbols
    symbolic a;
    symbolic X;
    symbolic Y;
    symbolic Z;

    // Loading data
    matrix x = read_data("xdata.csv");
    matrix y = read_data("ydata.csv");
    matrix z = read_data("zdata.csv");

    // Symbolic formula
    formula f = a * X * transpose(Y) * Z;

    // Computation
    return f(a = 0.5, X = x, Y = y, Z = z);
}
```
### Designing Domain Specific Languages

#### Most important principle
- Start with what users should be able to write

#### Interdisciplinarity
- Start from application domain
- Reverse engineer grammar rules from application domain

#### AST manipulation
- **DSL**: Domain-Specific Languages: Create new languages with new compilers
- **EDSL**: Embedded Domain-Specific Languages: Use metaprogramming for AST manipulation
Passing as much information as possible to compilers

<table>
<thead>
<tr>
<th>Current state of affairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Compilers generally have no idea what the end user has in mind</td>
</tr>
<tr>
<td>- Information is lost in between the user and the compiler</td>
</tr>
<tr>
<td>- Compilers try to guess the information that has been lost</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- High-level information is useful information to be exploited for code transformation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keeping the structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Reflecting the structure of application domain abstractions in the structure of programs</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Summary: 1) There is a combinatorial explosion of complexity is scientific codes

\[ \prod_{i=0}^{n} |c_i| \]

Missed scientific opportunities
Under-exploited supercomputers & datasets

Wall of complexity

Uncharted territories

Written codes

Researchers production

Tension

Tipping point

Time

Amount of scientific code components

Physics

Geometry

Topologies

Architectures

Parallelism

Algorithms

Numerical methods

Data structures

Precision

Data analysis
Summary: 2) The problem is often a blind spot of computational sciences
Summary: 3) Working on programming languages can allow to reduce this complexity

\[ \prod_{i=0}^{n} |C_i| \Rightarrow \sum_{i=0}^{n} |C_i| \quad \text{(in first approximation)} \]
Summary: 4) Evaluating solutions in terms of GPE can serve as a guide
## Conclusions

### Performance
- In first-order approximation, computational power can be considered as infinite at time of development.

### Genericity
- Conceptual approximations get amplified through higher layers of abstractions.
- Concept-based design using bottom-up approach can help.

### Expressivity
- Starting with what users should be able to write.
- Pass as much high-level information as possible to compilers.
- Reflecting the structure of the application domain into the structure of programs.

### The wall of software complexity
- Many application domains are facing or will soon face a problem of structural code complexity.
- It’s anything but a technical problem and will require computer science approaches.
- Research in programming languages and compilers can help.
Thank you for your attention

Any question?