Modern computer science and IGA

A. Ratnani
IPP, Garching, Germany
July 7, 2017

Acknowledgements: Colleagues and collaborators

B. Dingfelder[2], E. Franck[1], M. Gaja[2], H. Guillard[3],
K. Kormann[2], J. Lakhlili[2], A. Loyer[3], C. Manni[4], M. Mazza[2],
B. Nkonga[5], S. Serra-Capizzano[6], E. Sonnendrücker[2],
H. Speleers[4], T-M. Tran[7], X. Wang[2]

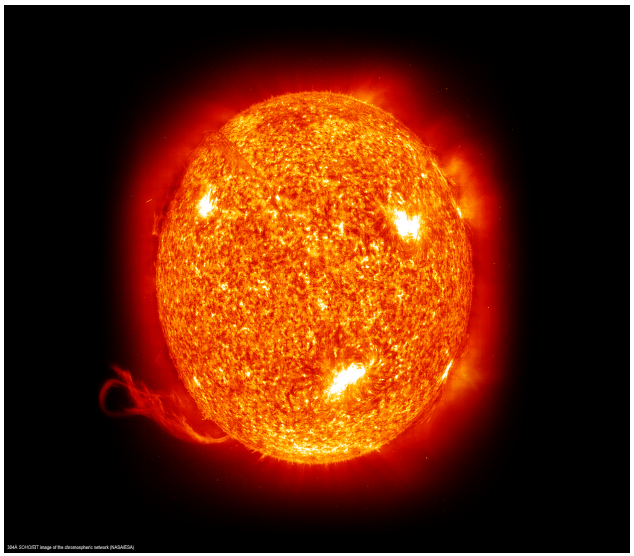
[1] Inria Nancy Grand Est and IRMA Strasbourg, France
[2] Max-Planck-Institut für Plasmaphysik, Garching, Germany
[3] Inria Sophia-Antipolis, France
[4] University of Rome Tor Vergata, Rome, Italy
[5] University of Nice, France
[6] University of Insubria, Como, Italy
[7] EPFL Lausanne, Switzerland
[8] Max-Planck-Institut für Plasmaphysik, Greifswald, Germany

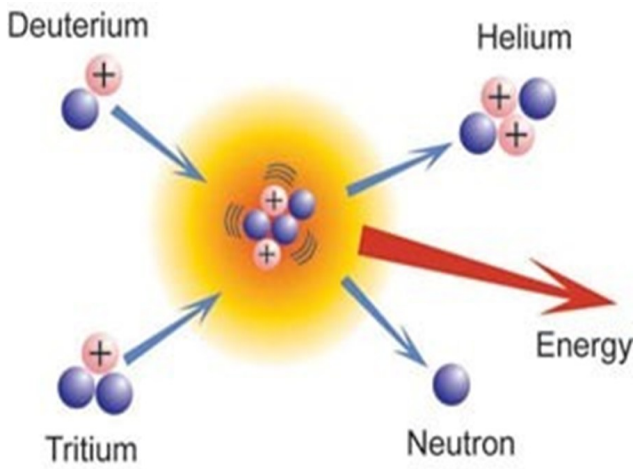
Outline

- Context and introduction to Plasma Physics
- Modern architectures of supercomputers
- Applications
  - Compatible B-Splines Finite Elements
  - Multigrid (GLT)
- Modern computer science (for mathematicians)
  - Language Theory and Compilers
  - Language Theory and GLT

Context and introduction to Plasma Physics

Fusion DT: At sufficiently high energies, deuterium and tritium can fuse to Helium. A neutron and 17.6 MeV of free energy are released. At those energies, atoms are ionized forming a **plasma**.

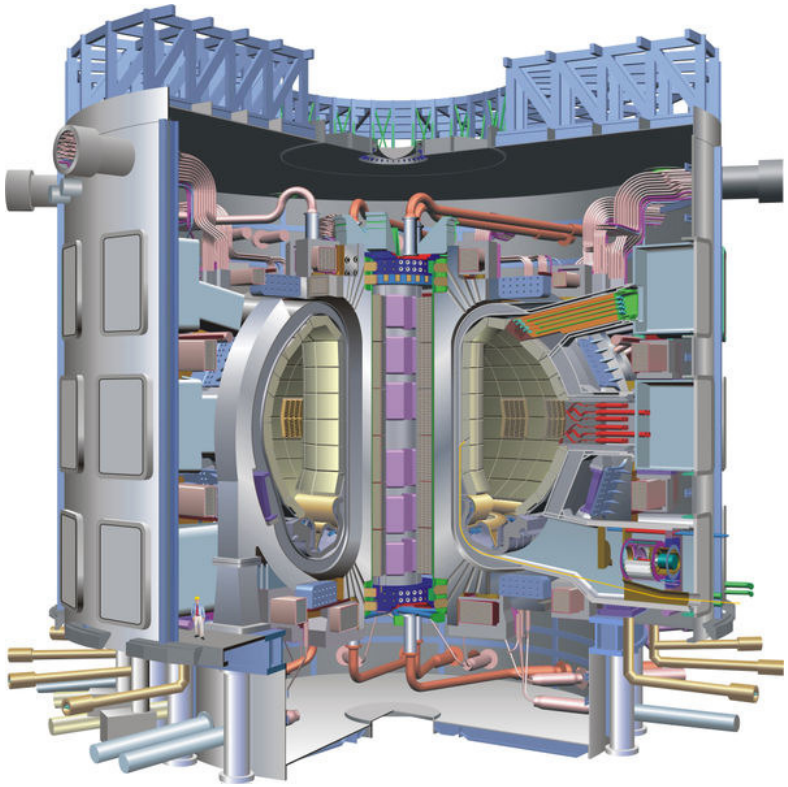


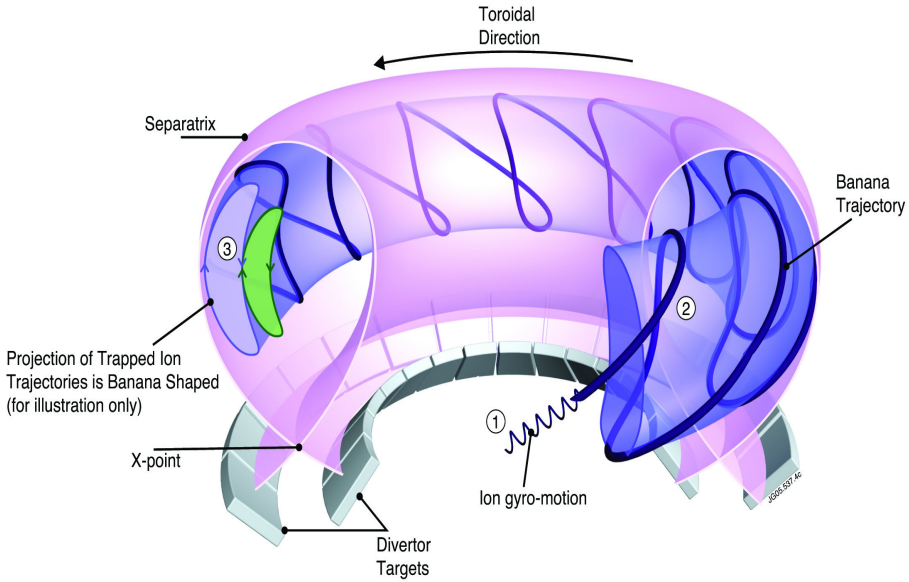


(left) The sun is a fusion reactor, (right) Fusion nuclear reaction

Magnetic Confinement Fusion Devices

Tokamaks

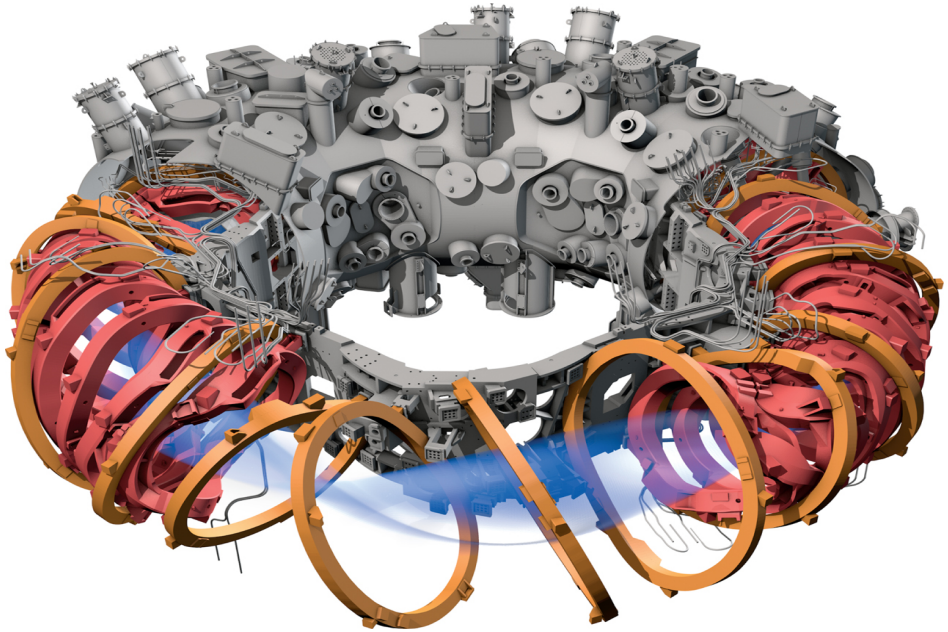


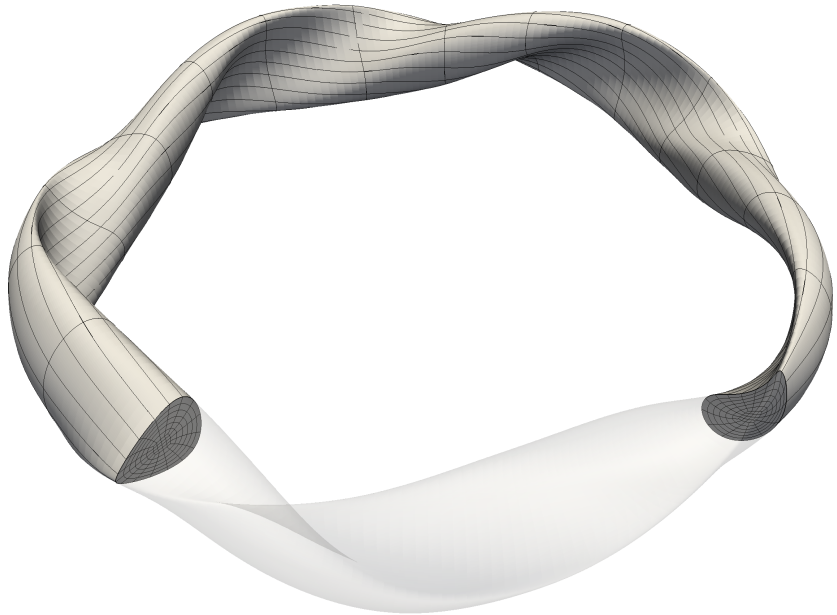


(left) ITER machine (Cadarache, France)
(right) example of particles trajectory inside the Tokamak.

Magnetic Confinement Fusion Devices

Stellarators





(left) W7X (Greifswald, Germany)
(right) mesh example (Courtesy from F. Hindenlang)

- Plasmas are a collection of charged particles.
- Fourth state of matter along with solid, liquid and gas.
- Dominant force applied on particles is electromagnetic force.
- Microscopic (n-body) model consists in equations of motion of single particles in electromagnetic field.

$$\frac{d\mathbf{x}_k}{dt} = \mathbf{v}_k, \quad \frac{d\mathbf{v}_k}{dt} = \frac{q}{m}\left(\mathbf{E}(t,\mathbf{x}_k) + \mathbf{v}_k \times \mathbf{B}(t,\mathbf{x}_k)\right) \tag{1}$$

- Too many particles for numerical simulations.
- Statistical physics derives more tractable **kinetic equations** (Boltzmann or Vlasov) from these.

Let s denotes particles species (electron, ion, impurities, ...)
The Plasma is governed by the coupling of Vlasov-Maxwell system

$$\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \frac{\partial f_s}{\partial \mathbf{x}} + \frac{q_s}{m_s} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0 \qquad (2)$$

$$\rho = \sum_s q_s \int f_s(t, \mathbf{x}, \mathbf{v}) d\mathbf{v}, \quad \mathbf{J} = \sum_s q_s \int f_s(t, \mathbf{x}, \mathbf{v}) \mathbf{v} d\mathbf{v} \qquad (3)$$

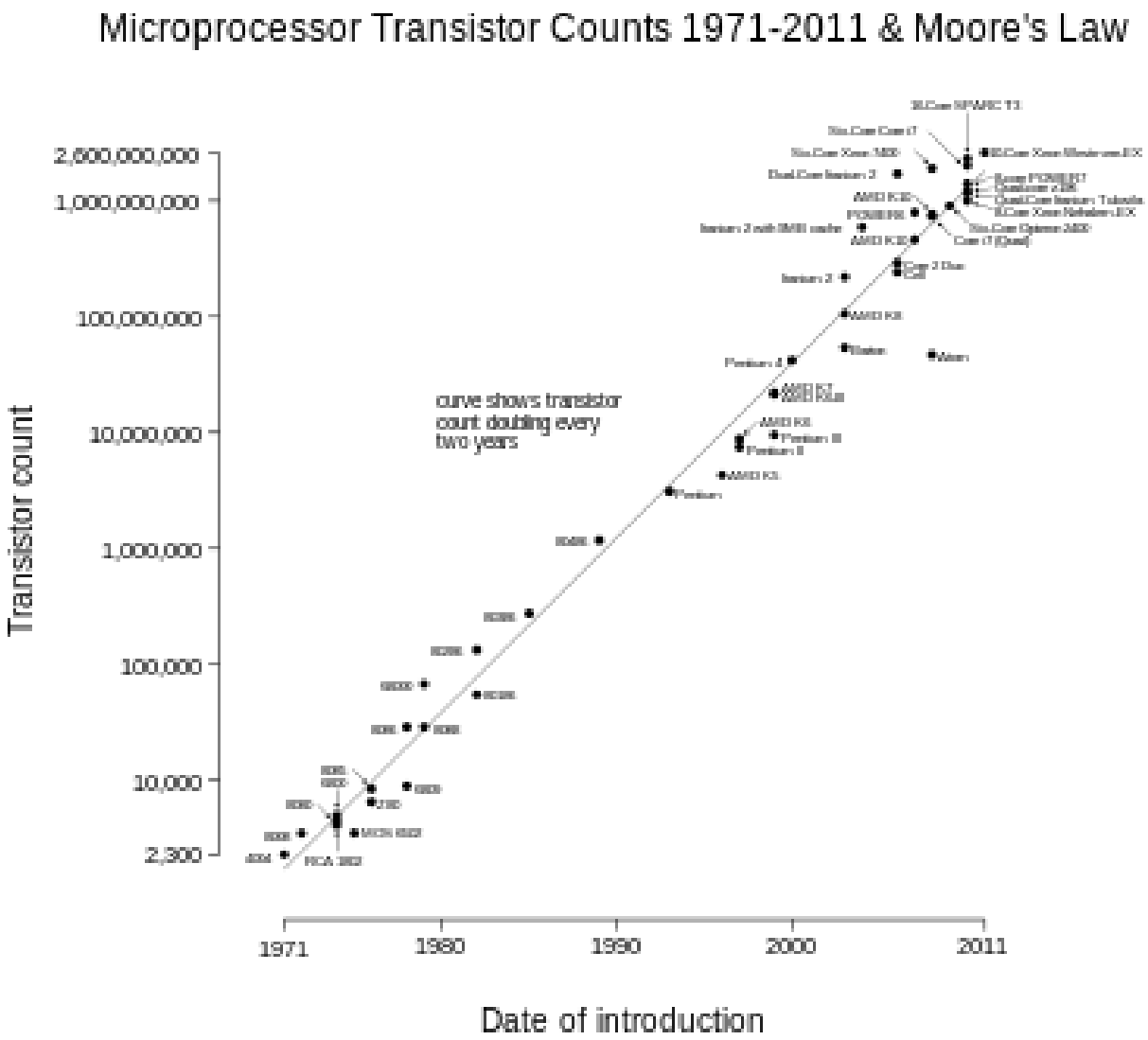
$$\frac{\partial \mathbf{E}}{\partial t} + \mathrm{curl}\mathbf{B} = \mu_0 \mathbf{J}, \quad \frac{\partial \mathbf{B}}{\partial t} + \mathrm{curl}\mathbf{E} = 0 \qquad (4)$$

$$\mathrm{div}\mathbf{E} = \frac{\rho}{\epsilon_0}, \quad \mathrm{div}\mathbf{B} = 0 \qquad (5)$$

- Taking the moments leads to a **fluid description** of the Plasma.

Modern architectures of supercomputers

Moore's Law


Microprocessor Transistor Counts 1971-2011 & Moore's Law
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
curve shows transistor count doubling every two years
1971
1980
1990
2000
2011
Date of introduction

Common Processors

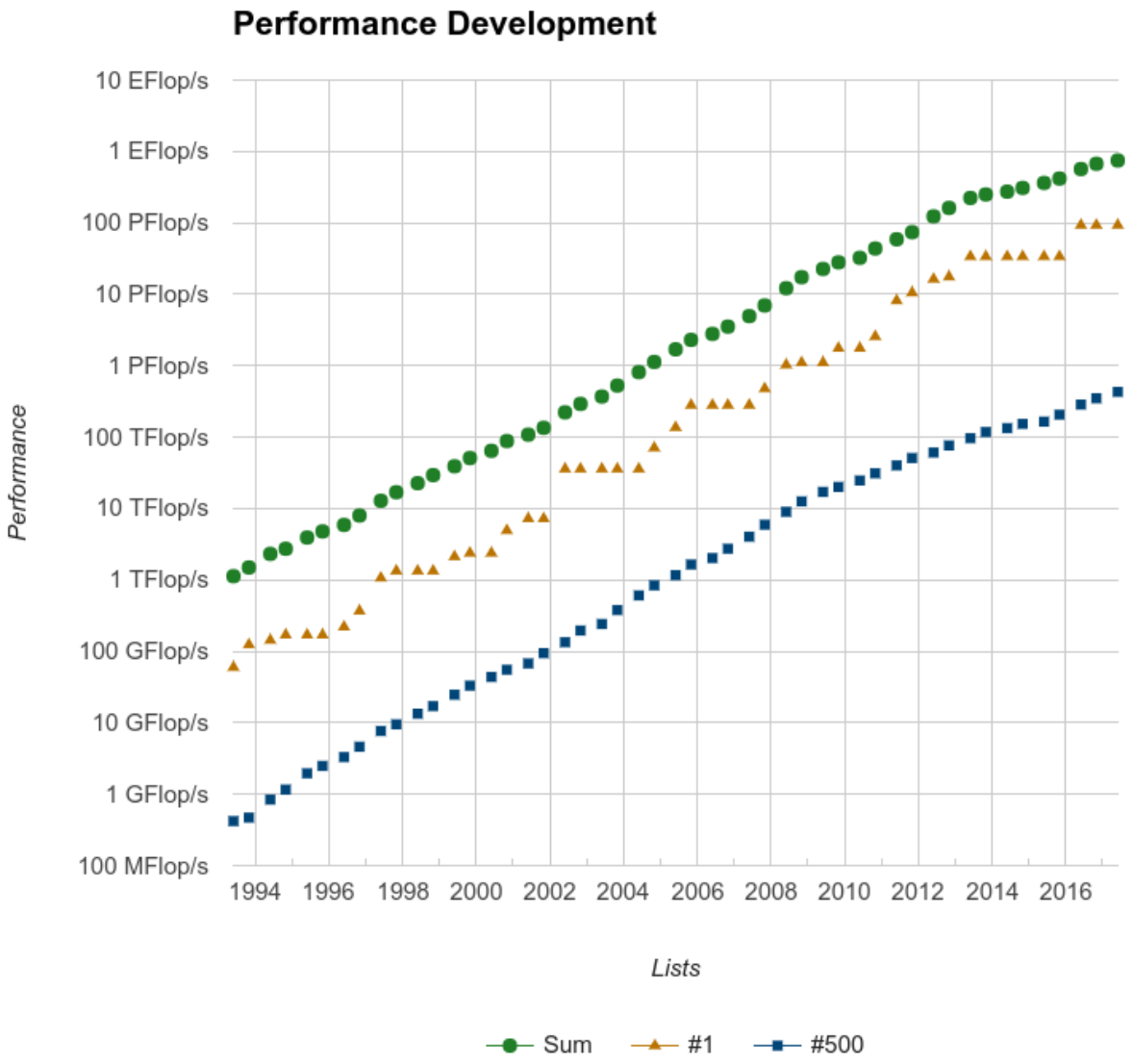
Processor	Launched	Number of Cores	Freq.
E3-1500 v6 (Mobile, 7th generation)	2017-Q1	2-4	2.1-3.0 Ghz
Xeon D-1500 (formerly Broadwell)	2016-Q1	8-10	1.6-2.4 Ghz
E3-1200 v5 (Desktop, 6th generation)	2015-Q4	2-8	1.5-2.2 Ghz
Xeon E5-2600 v3 (formerly Haxwell)	2015-Q3	4	2-2.8 Ghz

Table : Some Intel processors

Processor	L2 cache	Number of Cores	Freq.
Opteron 6386 SE	16 MB	16	2.8-3.5 Ghz
Athlon X4 880K (Desktop)	4 MB	4	4.0-4.2 Ghz
Athlon X2 450K (Desktop)	1 MB	2	3.5-3.9 Ghz

Table : Some AMD processors

Top 500


Performance Development
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
Performance
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
Lists
Sum
#1
#500

Top 500 Family system share evolution

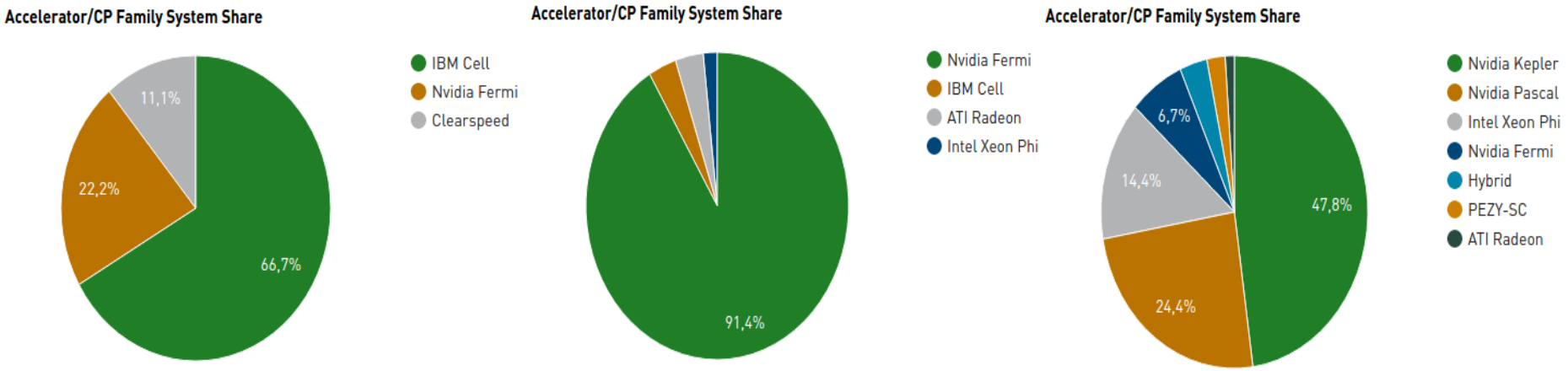


Figure : From left to right: 2010, 2012, 2017

Walls of modern supercomputers

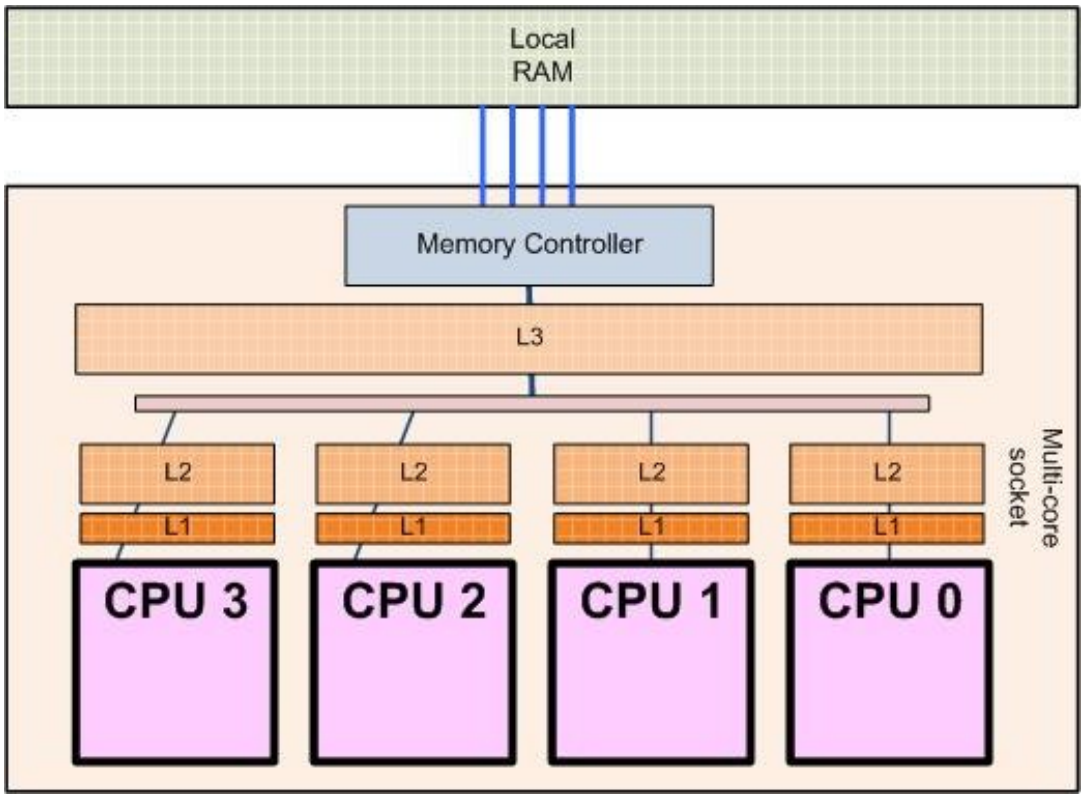
- Programming wall
  - ➠ writing parallel codes is time-consuming
- Memory wall
  - Latency of the memory is decreasing very slowly
  - Concurence: number of cores per memory module is increasing
  - Technological advances on memory are very slow
- Portability wall
  - ➠ CPUs, GPUs, Mics
  - ➠ Portability is mandatory
- Power wall
  - Dissipated power $\sim \omega^3$ (freq.)
  - Dissipated power per cm^2 is limited by the cooling system
  - Power costs are expensive (critical for Exascales)

Walls of modern supercomputers

- The computing power of supercomputers is doubling every year (faster than Moores Law, but electrical consumption is also increasing).
- The number of cores is increasing rapidly (massively parallel (IBM Blue Gene Q) and many-cores architectures (Intel Xeon Phi)).
- Emergence of heterogeneous accelerated architectures (standard processors coupled with GPU).
- Machine architecture is becoming more complex and the number of layers increasing (processors/cores, memory access, network and I/O).
- Memory per core has been stagnating and is beginning to decrease.
- Performance per core is stagnating and is much lower on some machines than on a simple laptop (IBM Blue Gene).
- Throughput towards the disk and memory is increasing more slowly than the computing power.

Memory wall

- On-chip (in opposition to off-chip) caches are faster, but very small size
- Large caches have delays
  - Hardware to check longer addresses in cache
  - Associativity (needed for more general set of data in cache)
- Even worse between Cores or Nodes (and depends on the topology)



Amdahl's Law

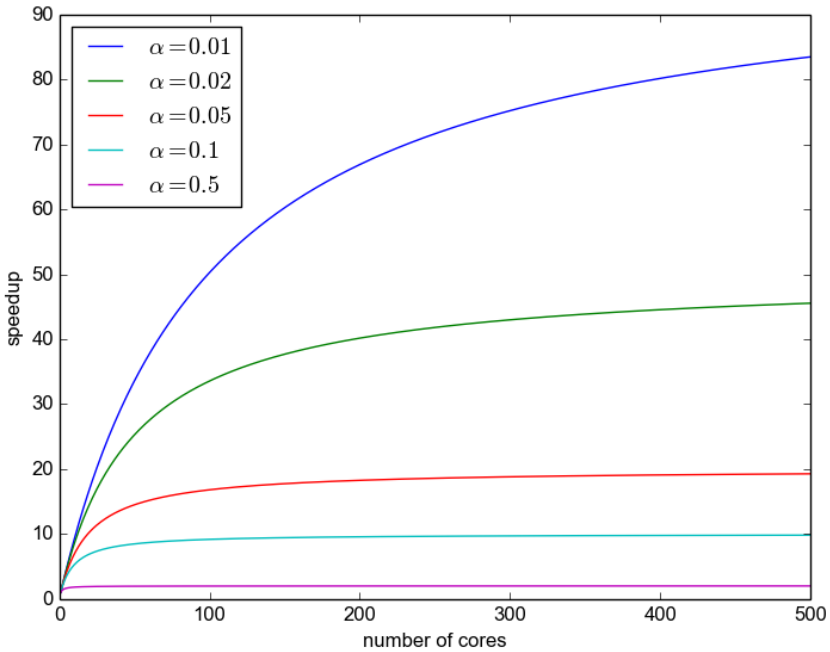
- Amdahl's Law predicts the theoretical maximum speedup obtained by parallelizing a code ideally, for a given problem with a fixed size:

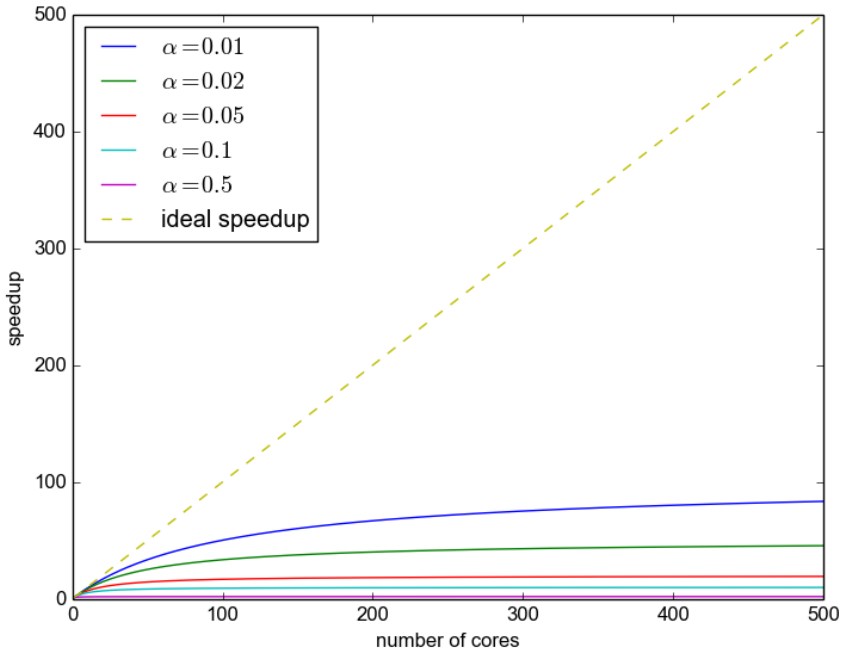
$$Sp(P) = \frac{T_s}{T_{\|}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

with Sp the speedup, T is the execution time of the sequential code (monoprocessor), $T_{\|}(P)$ the execution time of the ideally parallelized code on P cores and α the non-parallelizable part of the application.

- Regardless of the number of cores, the speedup is always less than the inverse of the percentage represented by the purely sequential fraction.

Amdahl's Law


$\alpha = 0.01$
$\alpha = 0.02$
$\alpha = 0.05$
$\alpha = 0.1$
$\alpha = 0.5$
speedup
number of cores
0
10
20
30
40
50
60
70
80
90
0
100
200
300
400
500


$\alpha = 0.01$
$\alpha = 0.02$
$\alpha = 0.05$
$\alpha = 0.1$
$\alpha = 0.5$
ideal speedup
speedup
number of cores
0
100
200
300
400
500
0
100
200
300
400
500

Gustafson-Barsis Law

- The Gustafson-Barsis Law predicts the theoretical maximum speedup obtained by parallelizing a code ideally for a problem of constant size per core, and *supposing that the execution time of the sequential fraction does not increase with the overall problem* size:

$$Sp(P) = \alpha + P(1 - \alpha)$$

with Sp the speedup, P number of cores and α the non-parallelizable part of the application.

- This law is more optimistic than Amdahls because it shows that the theoretical speedup increases with the size of the problem being studied.

Consequences

Consequences for the applications

- It is necessary to exploit a large number of relatively slow cores.
- Tendancy for individual core memory to decrease: Necessity to not waste memory.
- Higher level of parallelism continually needed for the efficient usage of modern architectures (regarding both computing power and memory size).
- The I/O also becoming an increasingly current problem.

Consequences for the developers

- The time has ended when you only needed to wait a while to obtain better performance (i.e. stagnation of computing power per core).
- Increased necessity to understand the hardware architecture.
- More and more difficult to develop codes on your own (need for experts in HPC as well as multi-disciplinary teams).

Programming methods and languages

- MPI
- OpenMP
- OpenACC
- Task based parallelism (for example starpu)

Alternative solutions

- TBB (restricted to Inel compilers and C++)
- DSL (Chapel, Terra, Liszt ...)

Applications

Sparse Matrices

A Matrix is a Linear Operator, it provides the dot subroutine. However, a matrix has a specific internal data structure (depending on its format) that is used to fill-in the matrix or appropriate for some linear solvers.

Examples

- matrix_coo
- matrix_bnd
- matrix_csr
- matrix_csc
- matrix_cds

- Pros
  - \+ facilitates fast conversion among sparse formats
  - \+ permits duplicate entries (see example)
  - \+ very fast conversion to and from CSR/CSC formats
  - \+ very fast conversion of kronecker linear operators
- Cons
  - \- arithmetic operations
  - \- slicing

Sparse Matrices

A Matrix is a Linear Operator, it provides the dot subroutine.
However, a matrix has a specific internal data structure (depending on its format) that is used to fill-in the matrix or appropriate for some linear solvers.

Examples

- matrix_coo
- matrix_bnd
- matrix_csr
- matrix_csc
- matrix_cds

- Pros
  - \+ memory consumption
  - \+ band solvers
- Cons
  - \- works only for 1d

Sparse Matrices

A Matrix is a Linear Operator, it provides the dot subroutine. However, a matrix has a specific internal data structure (depending on its format) that is used to fill-in the matrix or appropriate for some linear solvers.

Examples

- matrix_coo
- matrix_bnd
- matrix_csr
- matrix_csc
- matrix_cds

- Pros
  - \+ efficient arithmetic operations CSR + CSR, CSR * CSR, etc.
  - \+ efficient row slicing
  - \+ fast matrix vector products
- Cons
  - \- slow column slicing operations (consider CSC)
  - \- changes to the sparsity structure are expensive (consider LIL or DOK)

Sparse Matrices

A Matrix is a Linear Operator, it provides the dot subroutine. However, a matrix has a specific internal data structure (depending on its format) that is used to fill-in the matrix or appropriate for some linear solvers.

Examples

- matrix_coo
- matrix_bnd
- matrix_csr
- matrix_csc
- matrix_cds

- Pros
  - \+ efficient column slicing, column-oriented operations
  - \+ suited for some direct solvers
- Cons
  - \- slow row slicing, expensive changes to the sparsity structure

Sparse Matrices

A Matrix is a Linear Operator, it provides the dot subroutine.
However, a matrix has a specific internal data structure (depending on its format) that is used to fill-in the matrix or appropriate for some linear solvers.

Examples

- matrix_coo
- matrix_bnd
- matrix_csr
- matrix_csc
- matrix_cds

- Pros
  - + efficient banded matrices with bandwidth that is fairly constant from row to row
  - + can be extended to multi-dimensional case
  - + leads to the stencil format when the bandwidth is constant
  - + well balancing for distributed memory
- Cons
  - - (block) structured grid

Compatible B-Splines Finite Elements

Domain Decomposition

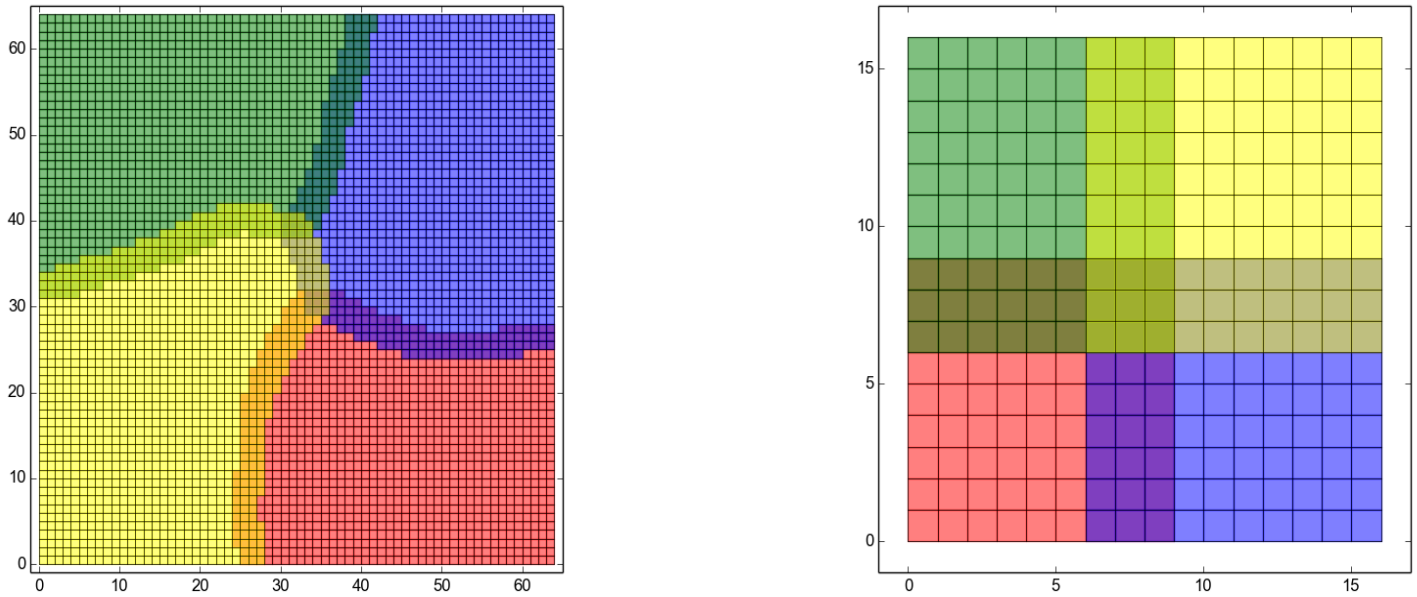
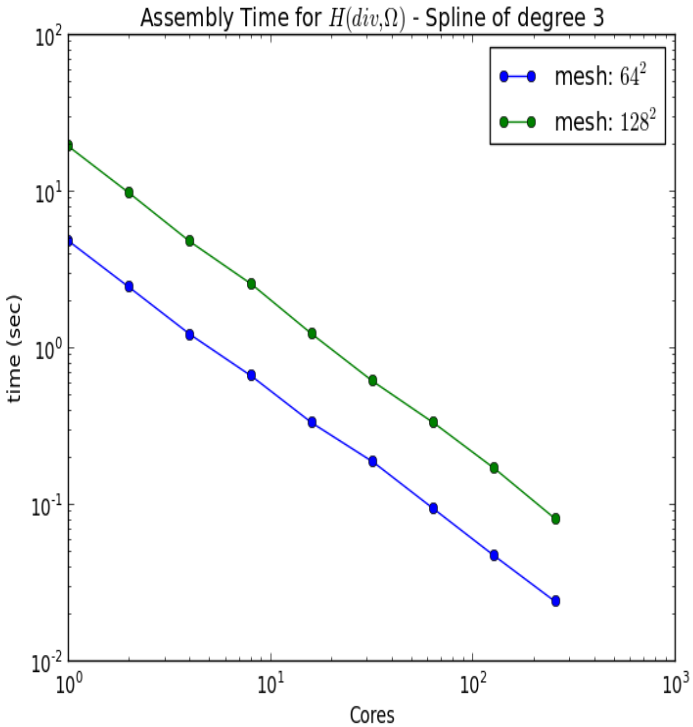


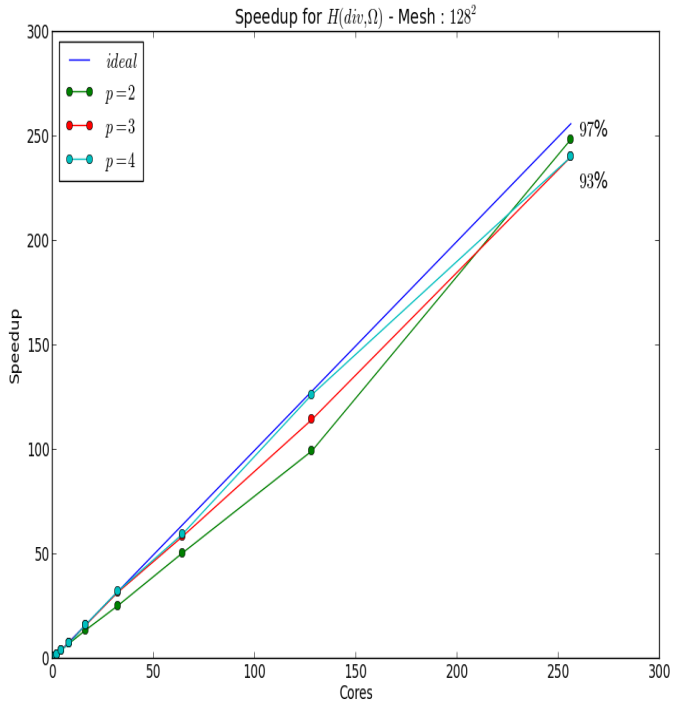
Figure : Metis (left) and tensor (right) partitioning.

Which one is the best? why?

Compatible B-Splines Finite Elements

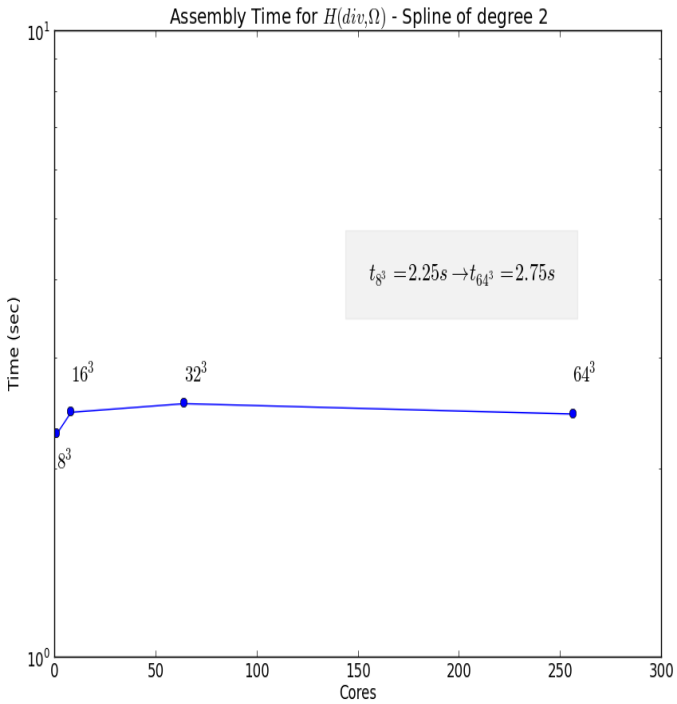
The $2D$ case





Compatible B-Splines Finite Elements

The $3D$ case



Statistics: Quadratic Splines on a grid 32^3:

- $23'101'440$ non zeros for $H(curl)$
- $98'304$ dofs for $H(curl)$
- $13'860'864$ non zeros for $H(div)$
- $98'304$ dofs for $H(div)$

POMS: Parallel and robust Multigrid for B-Splines

Motivations and goals

- a Parallel MG+GLT (blackbox) solver using *distributed memory* `MPI` and *shared memory* `OpenMP` and `OpenACC`.
- main ingredient for other solvers
  - Curl-Curl, Div-Div, Alfven operator, ...
  - Auxiliary spaces preconditioning
- Hybrid version MPI+OpenMP in progress (first runs on Marconi) still in progress
- **POMS** is supported by Eurofusion and HLST.

Marconi: HPC infrastructure

- Marconi-A1
  - **CPU** 2x Intel Xeon E5-2697 v4 @2.3GHz, 18 cores
  - # **Cores** 54432
  - # **Nodes** 1512
  - **Memory per node** 128GB
- Marconi-A2
  - **CPU** 1x Intel KNL Xeon Phi7250 @1.4Ghz, 68 cores
  - # **Cores** 244800
  - # **Nodes** 3600
  - **Memory per node** 96GB
- Marconi is classified in Top500 list:
  - Marconi-A1 rank 46 in June 2016
  - Marconi-A2 rank 12 in November 2016.

1. *Pure MPI* runs with the number of MPI tasks up to 36 (for Broadwell) or 68 for KNL.
2. *Pure OpenMP* with the number of threads up to 36 (for Broadwell) or 68 for KNL.
3. *Hybrid MPI+OpenMP* runs with a fixed 4 MPI tasks.

POMS: Parallel and robust Multigrid for B-Splines

Numerical results: Preliminary results in 1D

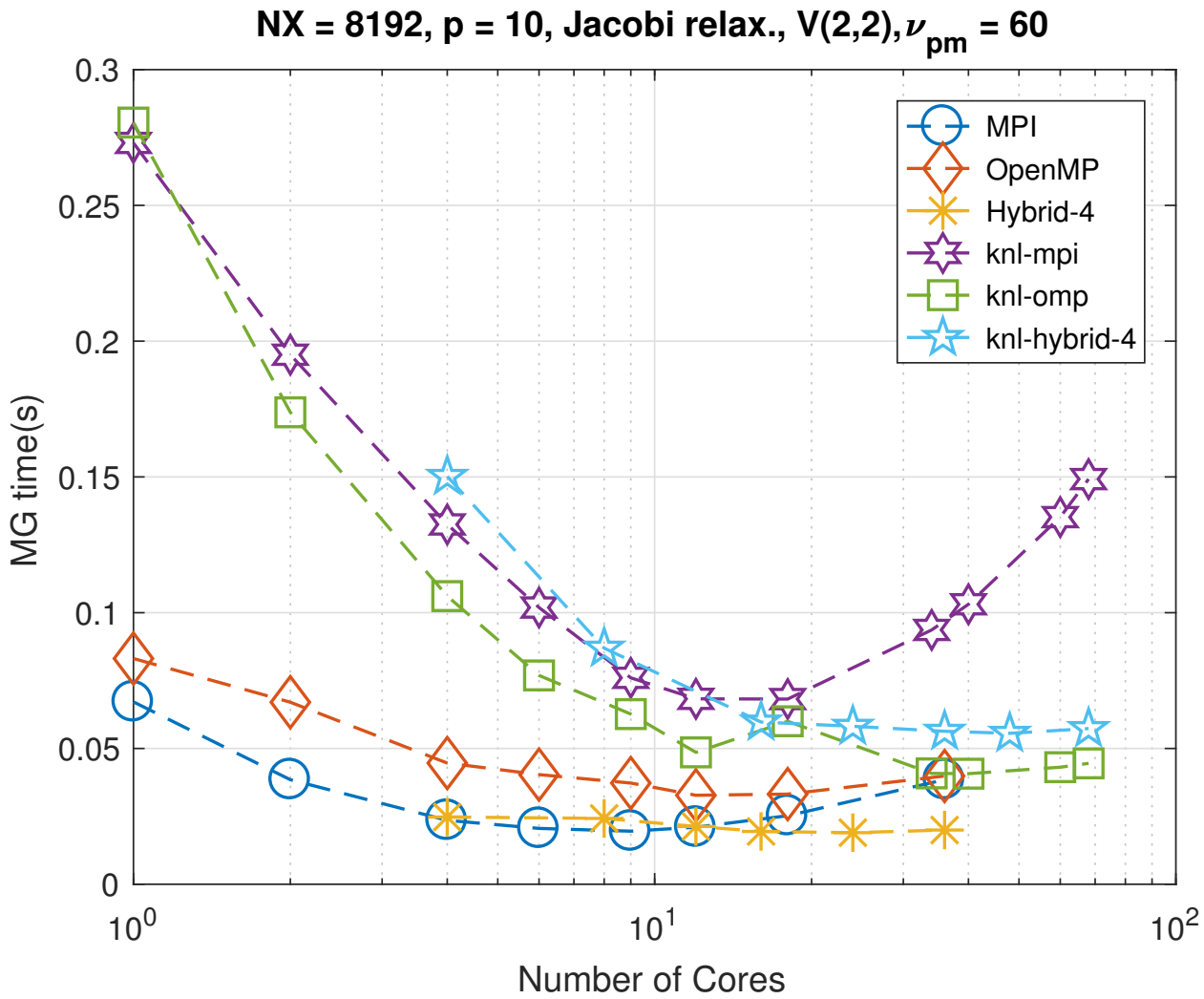


Figure : Strong scaling runs for pure `MPI`, pure `OpenMP` and hybrid `MPI+OpenMP` (with fixed 4 `MPI`) on a single Marconi node, using Broadwell

POMS: Parallel and robust Multigrid for B-Splines

Numerical results: Preliminary results in 1D

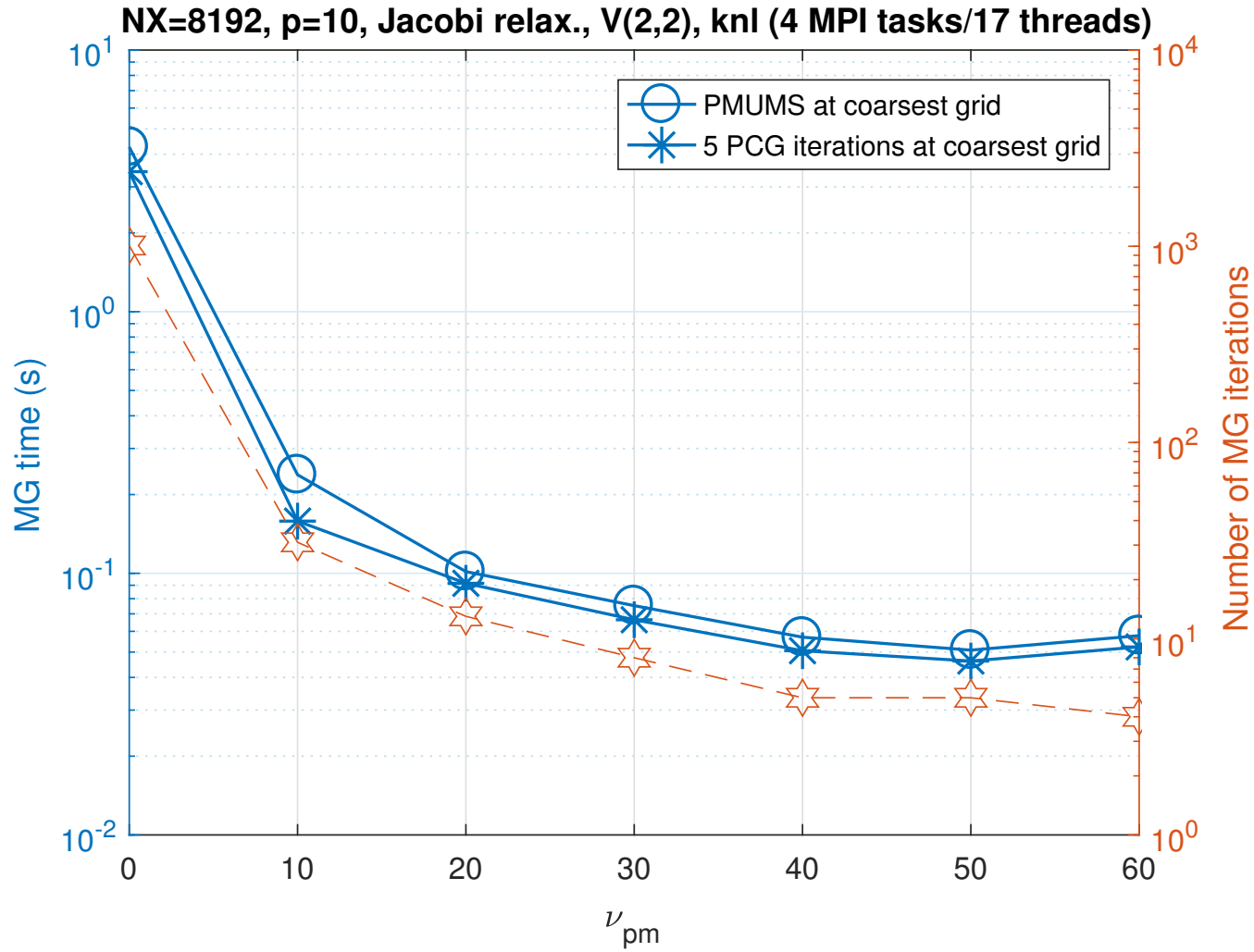


Figure : Effects of the GLT post-smoother number of iteration ν_{pm} on the MG times and number of iterations. This run is performed with 4 MPI tas

Modern computer science (for mathematicians)
or how to **make coding GREAT again**?

Linear Operators are objects that provide only a **generic** dot method.
$L : x \rightarrow y = L(x)$
A linear operator is defined by its **action** on arrays. No assumption is maid on the internal data structure.

Examples

- linear_operator_kron
- linear_operator_block
- linear_operator_expansion
- linear_operator_custom
- linear_operator_derivative

$$L = A_1 \otimes A_2, \quad A_1 \otimes A_2 \otimes A_3$$

Linear Operators are objects that provide only a **generic** dot method.
$L : x \rightarrow y = L(x)$
A linear operator is defined by its **action** on arrays. No assumption is maid on the internal data structure.

Examples

- linear_operator_kron
- linear_operator_block
- linear_operator_expansion
- linear_operator_custom
- linear_operator_derivative

$$L = \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{pmatrix}$$

Linear Operators are objects that provide only a **generic** dot method.
$L : x \rightarrow y = L(x)$
A linear operator is defined by its **action** on arrays. No assumption is maid on the internal data structure.

Examples

- linear_operator_kron
- linear_operator_block
- linear_operator_expansion
- linear_operator_custom
- linear_operator_derivative

$$L = \sum_k \alpha_k A_k$$

Linear Operators are objects that provide only a **generic** dot method.
$L : x \rightarrow y = L(x)$
A linear operator is defined by its **action** on arrays. No assumption is maid on the internal data structure.

Examples

- linear_operator_kron
- linear_operator_block
- linear_operator_expansion
- linear_operator_custom
- linear_operator_derivative

The user needs only to provide a procedure pointer

Linear Operators are objects that provide only a **generic** dot method.
$L : x \rightarrow y = L(x)$
A linear operator is defined by its **action** on arrays. No assumption is maid on the internal data structure.

Examples

- linear_operator_kron
- linear_operator_block
- linear_operator_expansion
- linear_operator_custom
- linear_operator_derivative

$$\mathbf{Jv} = \frac{F(\mathbf{u} + \epsilon\mathbf{v}) - F(\mathbf{u})}{\epsilon}$$

What is CLAPP/Django?

Within the **CLAPP** *environment developped at the NMPP, the* **Django** *(framework) allows for solving system of pdes using the Finite Elements method*

- Environment for Component-Based Parallel Programming
- Organized in small libraries
- Merging efficiency with generality
- Written in Fortran 2003 + MPI
- Full control of the OOP cost
  ➠can be made negligeable through an inlining procedure
- Robustness: more than 400 tests + Continuous Integration + extensive documentation
- Reliability, well-defined and natural interfaces, variability
- Open systems: the components are heterogenous and will target different architectures
- Package manager to build, install, update all components

Welcome to the CLAPP/Django world

Solid foundations based on Category Theory (CT)

- Fortran objects are of two kinds:
  - Atoms (**Objects**)
  - Associations (**Arrows**)
- Allows passing lower level specifications through high level API
- Arrows are designed carefuly by taking into account 3 attributes (or costs)
  - Communication between processors
  - Memory consumption
  - Arithmetic operations
- A key point of (CT) is diagram commutativity (parallel paths)

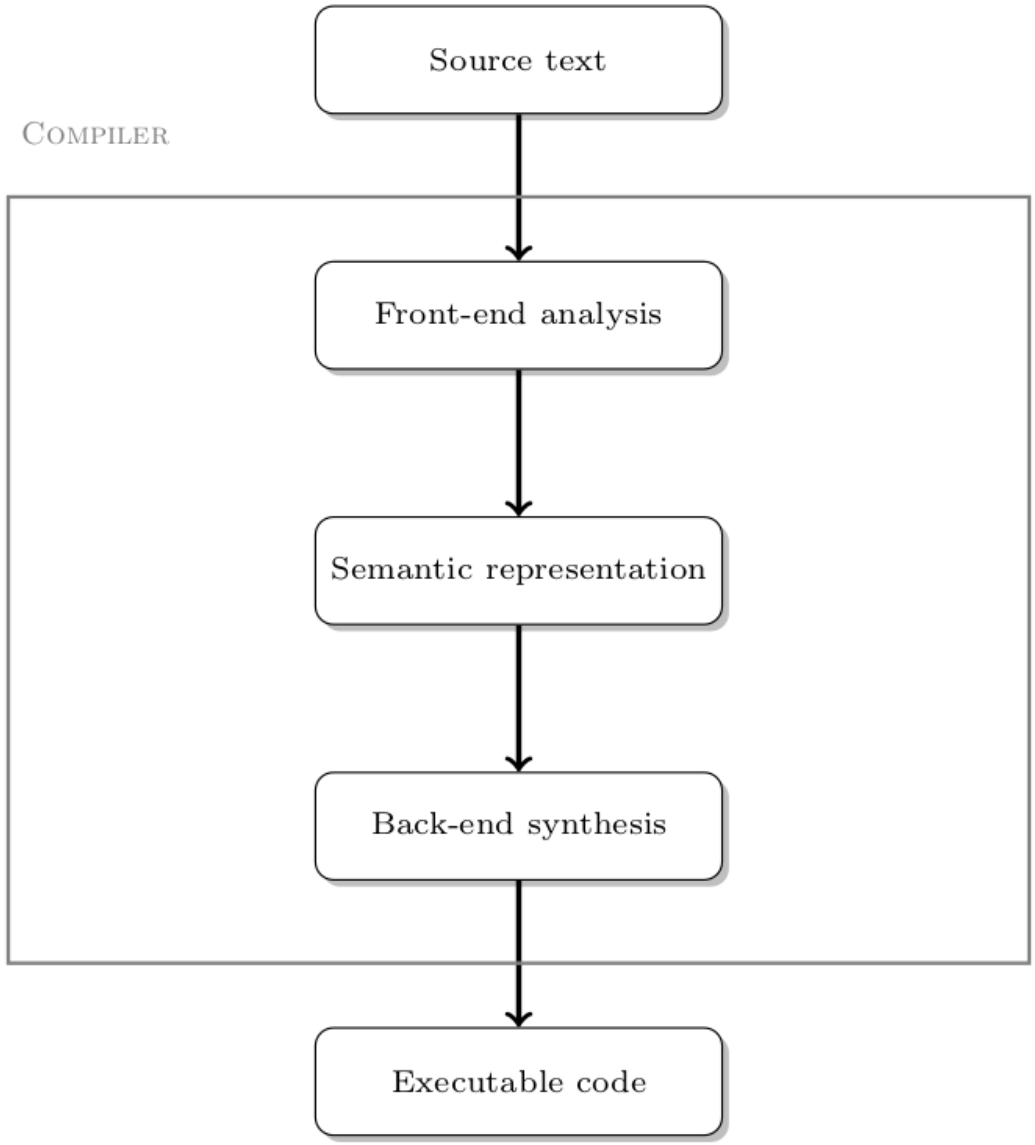


Figure 1: Conceptual structure of a compiler

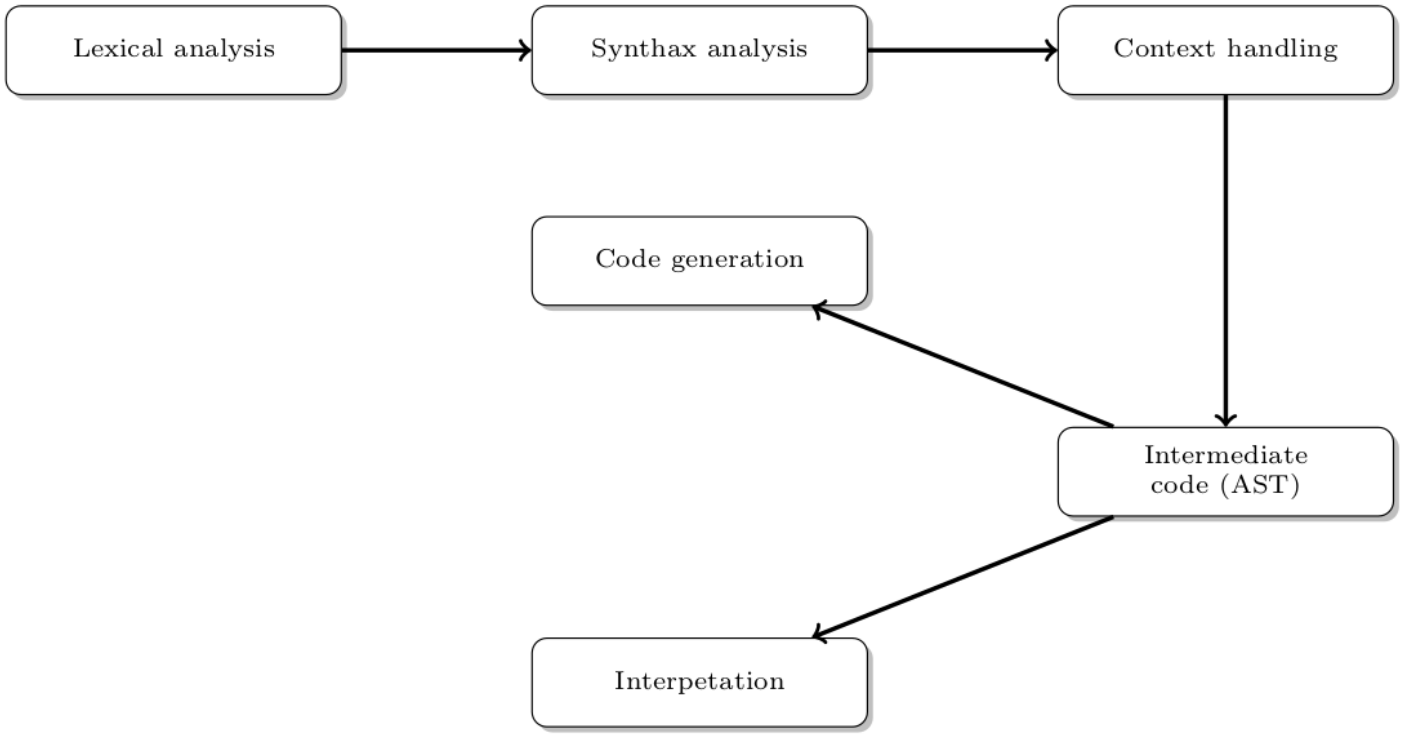


Figure 2: Conceptual structure of a compiler

Definitions

Definition (Alphabet)

An **alphabet** Σ is a finite nonempty set of symbols. Symbols are assumed to be indivisible.

Definition (String)

A **string** over an alphabet Σ is a finite sequence of symbols of Σ.

Definition (Concatenation)

Let $x = a_1 a_2 \ldots a_n$ and $y = b_1 b_2 \ldots b_m$ be two strings. The **concatenation** of x and y, denoted by xy, is the string $xy := a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$.

Definition (Language)

For any alphabet Σ, a **language** over Σ is a set of strings over Σ. The members of a language are also called the **words** of the language.

Definition

The set of all strings over an alphabet Σ is denoted by Σ^*, and the set of all nonempty strings over Σ is denoted by Σ^+. The empty set of strings is denoted by $\emptyset$.

- ϵ for the empty string, which contains no symbols at all
- $\forall x \in \Sigma^*$, we have $x = \epsilon x = x\epsilon$.

Definition

A **grammar** $\mathcal{G}$ is a quadruple $(\Sigma, V, S, \mathcal{P})$, where:

1. Σ is a finite nonempty set called the **terminal alphabet**. The elements of Σ are called the terminals.
2. V is a finite nonempty set disjoint from Σ. The elements of V are called the **nonterminals** or **variables**.
3. $S \in V$ is a distinguished nonterminal called the **start symbol**.
4. $\mathcal{P}$ is a finite set of **productions** (or **rules**) of the form

$$\alpha \to \beta$$

where $\alpha \in (\Sigma \cup V)^* \, V \, (\Sigma \cup V)^*$ and $\beta \in (\Sigma \cup V)^*$, i.e. α is a string of terminals and nonterminals containing at least one nonterminal and β is a string of terminals and nonterminals.

Grammar for arithmetic expressions

Backus-Naur form

$$
\begin{array}{rcll}
n & \in & \mathbb{R} & \textit{(numbers)} \\
\textit{expr} & ::= & \textit{term}(+\textit{expr} \mid \epsilon) & \textit{(expressions)} \\
\textit{term} & ::= & \textit{factor}(*\textit{term} \mid \epsilon) & \textit{(terms)} \\
\textit{factor} & ::= & (\textit{expr}) \mid n & \textit{(associativity)}
\end{array}
$$

This grammar can be extended easily to handle weak formulations.

- *Hint: Add new non-terminal variables*

Formal Language Theory

- the set of alphabets Σ is the set all lowercase letters (ex. a, f, ...)
- Uppercase letters will denote start symbols
- greec letters will denote any element of the alphabet and are non-terminal elements
- the set of all (admissible) verbs is denoted $\mathcal{V}$

Formal Language Theory

$$
\begin{aligned}
ABC &::= (\alpha\beta\gamma) \mid (\alpha\beta\beta\gamma) \mid (\tau\alpha\beta\beta\gamma) && \textit{(verb form I, II, V)}\\
\alpha\beta\gamma &::= \alpha a\beta\gamma && \textit{(subject form I)}\\
\alpha\beta\gamma &::= ma\alpha\beta o\gamma && \textit{(object form I)}\\
\alpha\beta\beta\gamma &::= mo\alpha a\beta\beta o\gamma && \textit{(subject form II)}\\
\alpha\beta\beta\gamma &::= mo\alpha a\beta\beta a\gamma && \textit{(object form II)}\\
\tau\alpha\beta\beta\gamma &::= mo\tau a\alpha a\beta\beta i\gamma && \textit{(subject form V)}\\
\tau\alpha\beta\beta\gamma &::= mo\tau a\alpha a\beta\beta a\gamma && \textit{(object form V)}
\end{aligned}
$$

What (sub)language is this?

Rules for GLT

Given a weak formulation $a(N_i, N_j)$, the following rules are used to compute the GLT symbol

$$
\begin{aligned}
N_i &::= N_{i_1}(N_{i_2})(N_{i_3}) && \textit{(tensor test function)} \\
N_j &::= N_{j_1}(N_{j_2})(N_{j_3}) && \textit{(tensor trial function)} \\
< expr_1 + expr_2 >_\Omega &::= < expr_1 >_\Omega + < expr_2 >_\Omega && \textit{(associaticity and integration)} \\
< f N_{i_1}^{r_1}(N_{i_2}^{r_2})(N_{i_3}^{r_3}) N_{j_1}^{s_1}(N_{j_2}^{s_2})(N_{j_3}^{s_3}) >_\Omega &::= f < N_{i_1}^{r_1} N_{j_1}^{s_1} >_{\mathcal{P}_1} (< N_{i_2}^{r_2} N_{j_2}^{s_2} >_{\mathcal{P}_2})(< N_{i_3}^{r_3} N_{j_3}^{s_3} >_{\mathcal{P}_3}) \\
r_1, r_2, r_3, s_1, s_2, s_3 &\in \mathbb{N} && \textit{(derivative orders)} \\
f &\in \{\text{constants, functions, fields}\} && \textit{(user inputs)}
\end{aligned}
$$

Formal Language for weak formulations

Examples

```
1  # 2D anisotropic diffusion problem
2  Domain(dim=2,kind=structured) :: Omega
3  Space(domain=Omega,kind=h1) :: V
4  Field(space=V) :: phi
5  Function(x,y) :: f
6  Function(x,y) :: b1
7  Function(x,y) :: b2
8
9  b(v::V) := < f * v >_Omega
10 a(v::V, u::V) := < b1 * b1 * dx(v) * dx(u)
11                  + b1 * b2 * dx(v) * dy(u)
12                  + b1 * b2 * dy(v) * dx(u)
13                  + b2 * b2 * dy(v) * dy(u) >_Omega
```

Formal Language for weak formulations

Examples

```
1  # 2D Vector Poisson
2  Domain(dim=2,kind=structured) :: Omega
3  Space(domain=Omega,kind=h1) :: V
4  Field(space=V) :: phi
5  Field(space=V) :: psi
6  Function(x,y) :: f
7  Function(x,y) :: g
8
9  b1(u::V) := < f * u >_Omega
10 b2(w::V) := < g * w >_Omega
11
12 a11(v::V, u::V) := < dx(v) * dx(u) + dy(v) * dy(u) >_Omega
13 a22(v::V, u::V) := < dx(v) * dx(u) + dy(v) * dy(u) >_Omega
14
15 b((v1,v2)::V) := b1(v1) + b2(v2)
16 a((v1,v2)::V,(u1,u2)::V) := a11(v1,u1) + a22(v2,u2)
```

Language Theory for GLT

- **Vale** is a formal language for Variational formulations
  - + available on *https://github.com/ratnania/vale*
  - + 1d, 2d, 3d, scalar and block variables, 1st and 2nd order derivatives
  - + provides a **backend** mecanism. You can link you favorite tool!
  - + Code generation of element matrix assembly for *Lua* and *Fortran*
    - ➠ OpenMP in progress
  - + uses **sympy**
  - - Compatible B-Splines
- **GeLaTo** is a symbolic library for GLT
  - + available on *https://github.com/ratnania/GeLaTo*
  - + GLT theorems are applied by manipulating the AST given by *Vale*.
  - + user-functions, constants
  - + Detailed notebooks are available
  - - mapping and 2nd order derivatives: still in progress
- Based on an Abstract Grammar using *https://github.com/igordejanovic/textX*
- Released version by October
- *Current work: Python extension to allow for parallel computation and code generation*

Conclusion and perspectives

- We are living in a new era because of the current architecture revolution of super-computers
- When designing new numerical schemes, one should take into account addition constraints
  - FLOPS and memory consumption are no longer enough
  - ➠ data movement is becoming more and more important (and expensive)
  - ➠ new metrics are being derived in order to take into account data movement, power consumption, ...
- a set of libraries has been implemented to offer an **automatic computation** of the GLT symbol as well as the numerical **discretization** of any system of pdes (restricted to H^1 discretization for the moment)
- Solid (abstract) mathematical foundations.

Simplicity does not precede complexity, but follows it.
Alan Perlis (First recipient of the Turing Award)