Using Component Model to implement a DSL: The Multi-Stencil Language Case Study

Hélène Coullon, Julien Bigot, Christian Perez

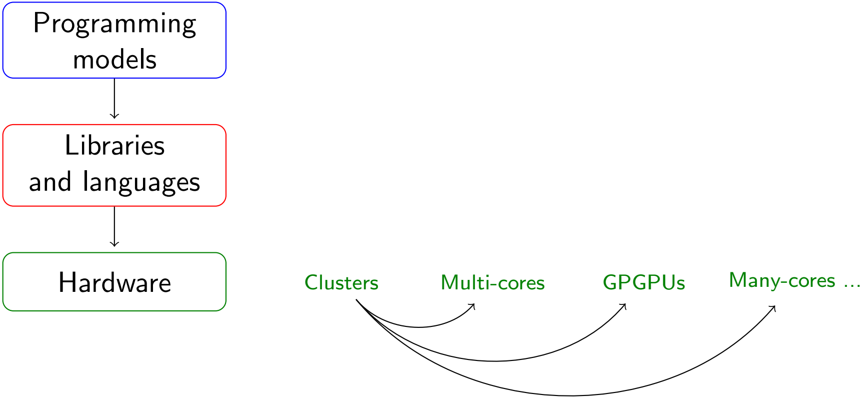
Avalon project team
Maison de la simulation

C2S@Exa – Scientific Days
Paris, November 8th, 2016

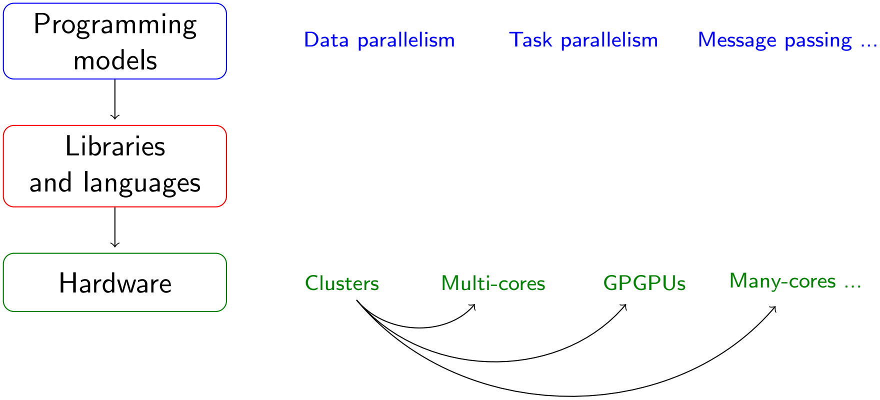
Table of Contents

Table of Contents

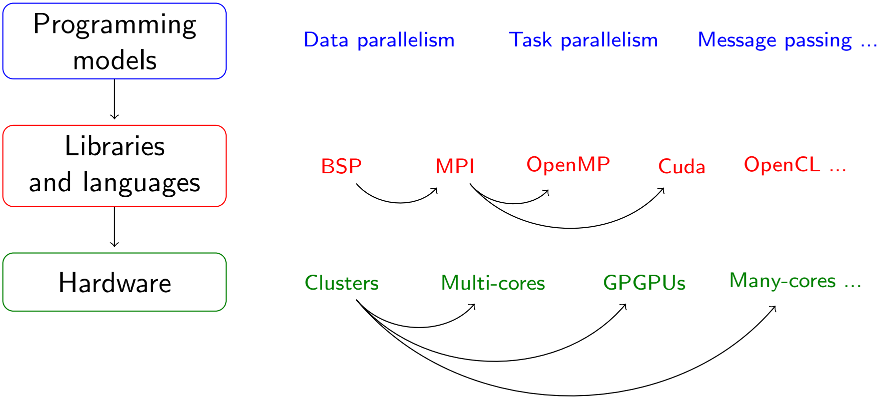
High performance computing and parallelism in 2016



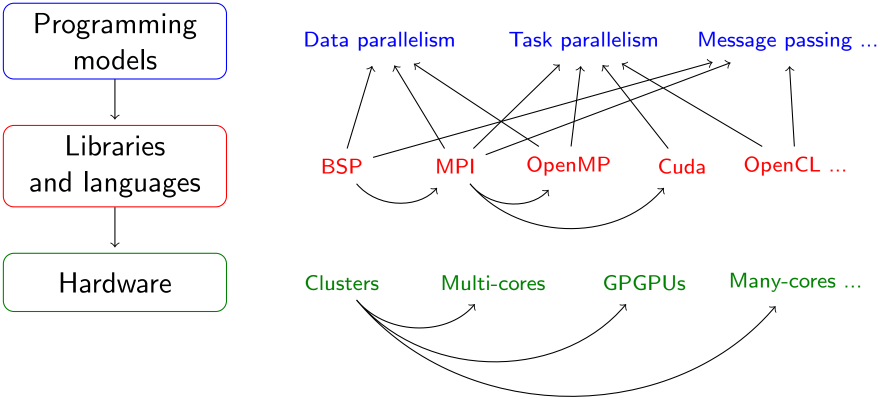
High performance computing and parallelism in 2016



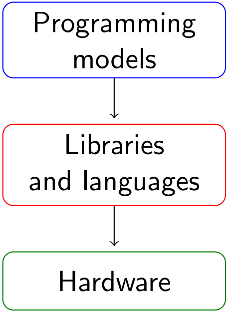
High performance computing and parallelism in 2016

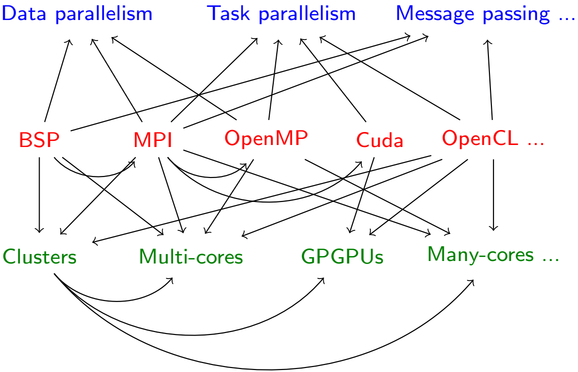


High performance computing and parallelism in 2016



High performance computing and parallelism in 2016





ETP4HPC: programming models for non experts to reach exascale!

Domain Specific Languages

Advantages

- Easy language for end users (can be application specific)
- Separation of concerns (domain/implementation)
- Implicit parallelization and optimizations

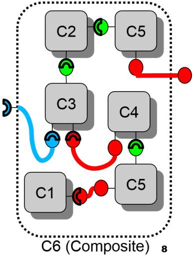
Limitations

- Difficulties deported to the DSL designer and implementer
  - Low level high performance programming
  - Maintainability and portability
- Difficult to combine DSLs (interoperability)
  - exascale applications = many specific domains and interactions

Component Models

Overview

- Technology advocating composition rather than development
  - Old idea (late 60') but major development in the 90'
  - After Object and before Model Driven Engineering (MDE)
- Component = black box with well defined interactions
  - Provides / Requires
- Application = Assembly of components



Benefits

- Application structure well defined
- Code-reuse and productivity
- Maintainability through separation of concerns

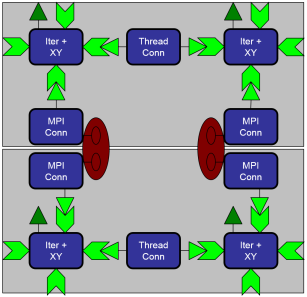
L2C: A HPC Component Model

A minimalist component model for HPC

- Component creation/deletion, configuration, and connection
- No L2C code between components runtime
  - No language interoperability provided
- Support of native interactions
  - Use/Provide: C++, CORBA, [FORTRAN (2008)]
  - MPI communicator
- `hlcm.gforge.inria.fr`

Ongoing extensions

- Concurrent reconfiguration
  - DirectL2C
- Task support
  - Cf next talk on Comet!



This talk

From a DSL: Multi Stencil Language

- Descriptive language,
- for Multi-Stencil simulations,
- without numerical code.

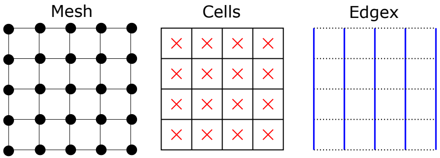
To a generated component assembly

- with automatic synchronizations for data and task parallelism,
- with empty functions to fill (components),
- with good performance.

Separation of concerns between domain/implementation/parallelization

Table of Contents

The MSL language: the mesh




```
mesh : cart
mesh entities : cell , edgex
computation domains :
  d1 in cell
  d2 in edgex
stencil shapes :
  ncc from cell to cell
  nce from cell to edgex
  nec from edgex to cell
```

- A Cartesian mesh is used
- Two kinds of mesh entities are declared onto it
- Two computations domains, one for each entity type
- Three stencil shapes (neighborhood from mesh entity to mesh entity)

The MSL language: quantity

```
quantity :
  A, cell
  B, cell
  C, edgex
  D, cell
  E, cell
  F, cell
  G, cell
  H, edgex
  I, cell
  J, cell
scalar : mu, tau
```

- A is a quantity applied onto cell
- C is a quantity applied onto edgex
- mu and tau are scalar values

The MSL language: time and computations

```
time : 500
computations :
   B[d1] = k0({tau},{A})
   C[d2] = k1({},{B[nce]})
   D[d1] = k2({},{C})
   E[d1] = k3({},{C})
   F[d1] = k4({},{D,C[nec]})
   G[d1] = k0({mu,tau},{E})
   H[d2] = k6({},{F})
   I[d1] = k7({},{G,H})
   J[d1] = k8({mu},{I[ncc]})
```

- The time loop is composed of 500 iterations.
- J is written
- onto the computation domain d1,
- by the computation k8,
- which read the scalar mu
- and the quantity I,
- which is accessed with the neighborhood ncc.

Table of Contents

Data and Task parallelism

Synchronizations

When a computation read a data, using a stencil shape, that has been written by a previous computation.

$\Gamma = [c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8]$
$\hookrightarrow [c_0, sync_1, c_1, c_2, c_3, sync_4, c_4, c_5, c_6, c_7, sync_8, c_8]$

Dependency graph

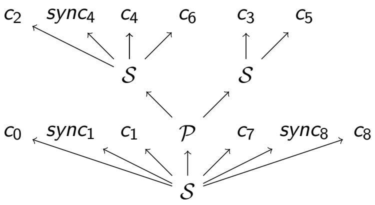
Node: a computation or a synchronization
Edge: a direct data dependency (Read / Write)

$$
\begin{tikzcd}
 & & & c_2 \arrow[r, dashed] & sync_4 \arrow[r, dashed] & c_4 \arrow[r] & c_6 \arrow[dr] & & & \\
c_0 \arrow[r, dashed] & sync_1 \arrow[r, dashed] & c_1 \arrow[ur] \arrow[dr] & & & & & c_7 \arrow[r, dashed] & sync_8 \arrow[r, dashed] & c_8 \\
 & & & c_3 \arrow[r] & c_5 \arrow[urrr] & & & & &
\end{tikzcd}
$$

Static or dynamic scheduling ?

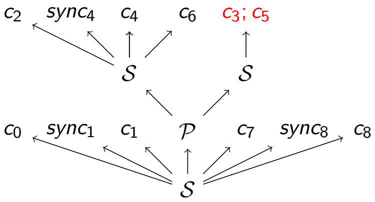
Series-Parallel Tree

Valdes & Al, The Recognition of Series Parallel Digraphs, STOC '79



Series-Parallel Tree

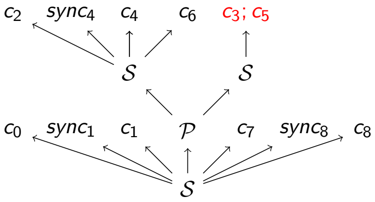
Valdes & Al, The Recognition of Series Parallel Digraphs, STOC '79



Loop fusion optimization possible

Series-Parallel Tree

Valdes & Al, The Recognition of Series Parallel Digraphs, STOC '79



Loop fusion optimization possible

Specific components

- *SEQ* to directly replace $\mathcal{S}$ nodes
- *PAR* to directly replace $\mathcal{P}$ nodes
- *SYNC* for synchronizations
- *K* for computation kernels

MSL to Component-based runtime

Ready-to-fill parallel pattern

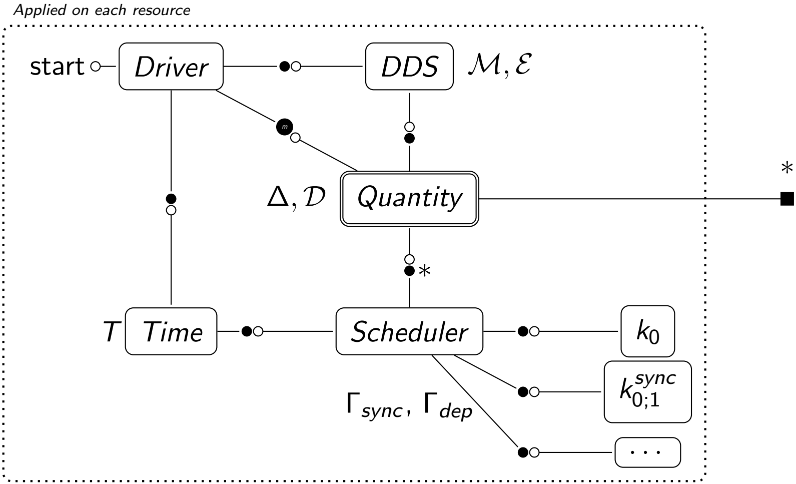
- Data parallelism
  - External distributed data structure
  - Automatic detection of synchronizations
- Task parallelism (mid-grain)
  - Static scheduling at the assembly level
  - Dynamic scheduling within a driver component
    - OpenMP task

The fine grain task parallelism is left to other languages:

- OpenMP in the kernels
- Kernels generated by stencil compilers
  - Pochoir, PATUS, Liszt etc.

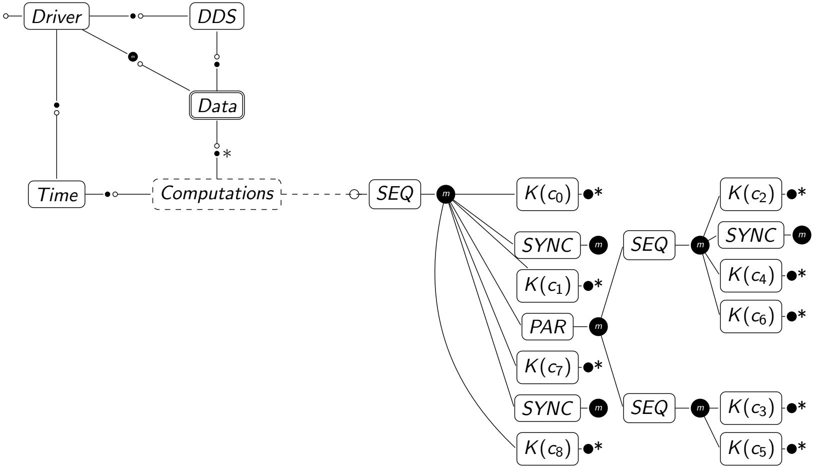
Component Assembly Runtime Template

A data parallel/task parallel component assembly.



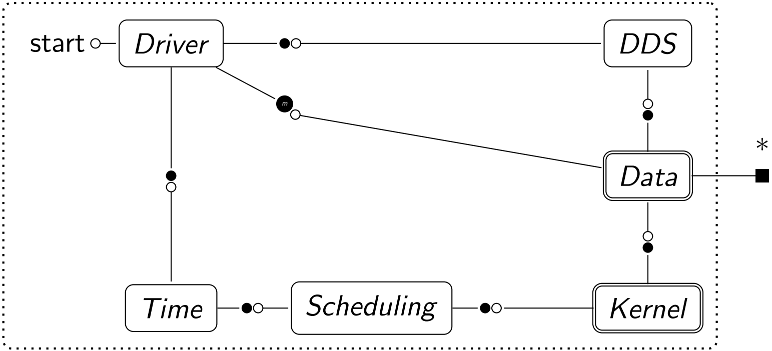
DDS: Distributed Data Structure (using SkelGIS)

Component-based runtime: Static Scheduling



Component-based runtime: Dynamic Scheduling

- *mpiOmpDyn*
  - *DDS* and *Quantity*: data parallelization.
  - *Scheduler*: DAG encoded into OpenMP tasks



Duplicated on each processor/core

Table of Contents

Evaluation Setup

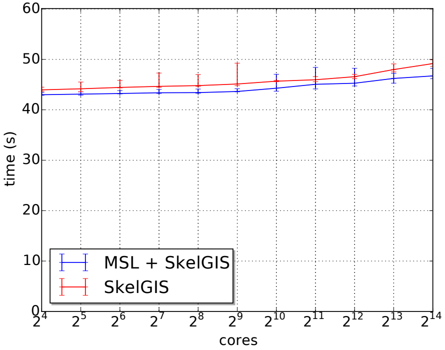
Application

- FullSWOF2D: developed at the MAPMO, University of Orléans
- Solve the shallow-water equations using a finite volume method
- Cartesian mesh
- 3 mesh entities, 7 computation domains, 48 quantities
- 98 computations (32 stencils, 66 local kernels)

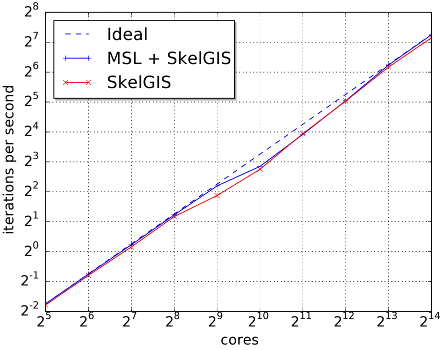
Machines

- Thin nodes TGCC Curie
- 2 CPU, 8 cores, Sandy Bridge EP (E5-2680) 2.7 GHz, 64 GB
- OpenMPI and gcc 4.9

Data Parallelization: Weak and Strong Scaling



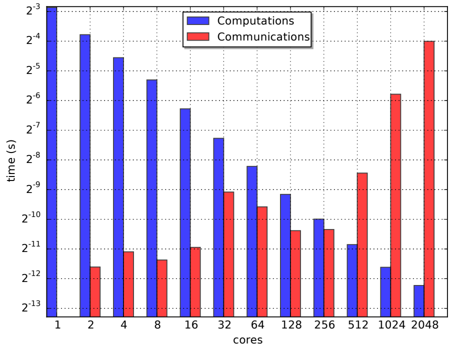
Weak Scaling
400x400 domain/core



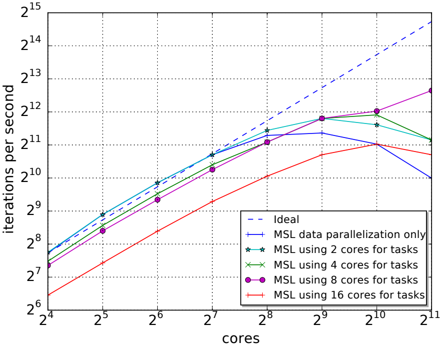
Strong Scaling
10k x 10k domain

No overhead introduced by components

Data Parallelization to Hybrid Parallelization: Strong Scaling



Data Parallelization



Data and Hybrid Parallelizations

Parallelism Level	1	2	3	4	6	10	12	16
Frequency	2	1	3	5	3	1	1	2

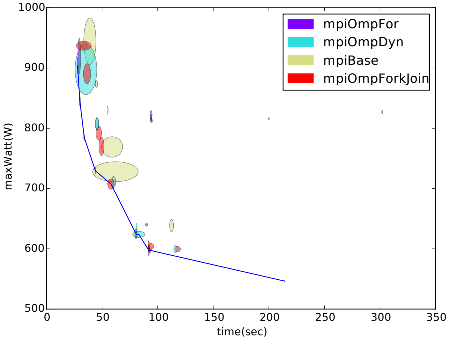
Task graph analysis

Dynamic Scheduling and Energy Evaluations

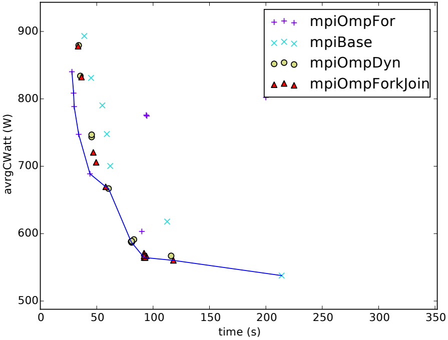
Application versions:

Variant Name	DDS / Quantity	Kernels	Scheduler
mpiBase	MPI	Sequential	Sequential
mpiOmpFor	MPI	Parallel loops	Sequential
mpiOmpForkJoin	MPI	Sequential	SPT
mpiOmpDyn	MPI	Sequential	OpenMP Tasks

Energy Usage on Grid'5000



Time vs Max Watt on 4 nodes

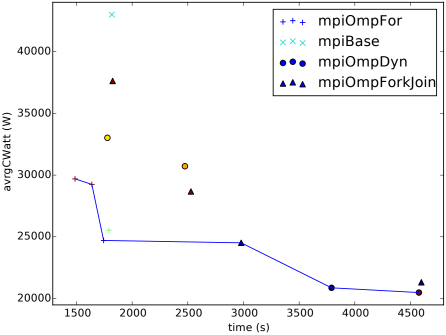


Time vs Average Watt on 4 nodes

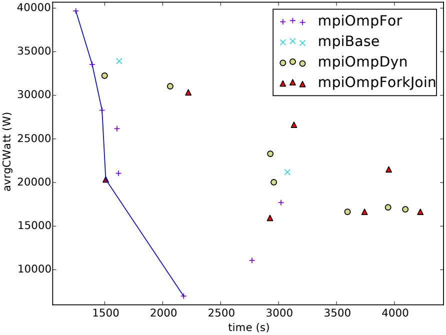
Parameters:

- Domain: 4000 x 4000
- #Iteration: 100

Energy Usage on Curie



Time vs Max Watt on 2048 cores



Time vs Average Watt on 2048 cores

Parameters:

- Domain: 20,000 x 20,000
- #Iteration: 10,000
- One measure every 5 min!

Table of Contents

Conclusion and Future Work

Conclusion

- A DSL for Multi-Stencil applications (MSL)
- The generation of a component based runtime, including scheduling
  - Data and task parallelism
- Evaluation on Grid'5000 and Curie: no overhead introduced
- Basis for ongoing energy studies

Perspectives

- Language improvement (convergence criteria, reduction etc.)
- OpenMP inside kernels/components
- CPU+GPGPUs using stencil compilers (Pochoir, PATUS etc.)
- Composition of various DSLs by agreeing on component runtimes?