# From DSL to HPC Component-Based Runtime: A Multi-Stencil DSL Case Study

Julien Bigot, Hélène Coullon, Christian Perez

INRIA team Avalon
Maison de la simulation (CEA)

WOLFHPC 2015 - $16^{th}$ November 2015

# Motivation

## + Domain Specific Languages

- Separation of concerns (domain/implementation)
- Easy language for the user
- Implicit optimizations
- Implicit parallelization

## - Domain Specific Languages

- Difficulties deported to the DSL designer
  - Low level high performance programming
  - Maintainability and portability
- As many DSLs as domains


Julien Bigot (CEA), Hélène Coullon (INRIA), Christian Perez (INRIA)

# Motivation

## Component models

- Divide an application into several independent black boxes
- Each component defines its interactions with outer world
- Application = Assembly of components

## + Component models

- Maintainability through separation of concerns
- Code-reuse and productivity
- Dynamic assembly of components

# Motivation

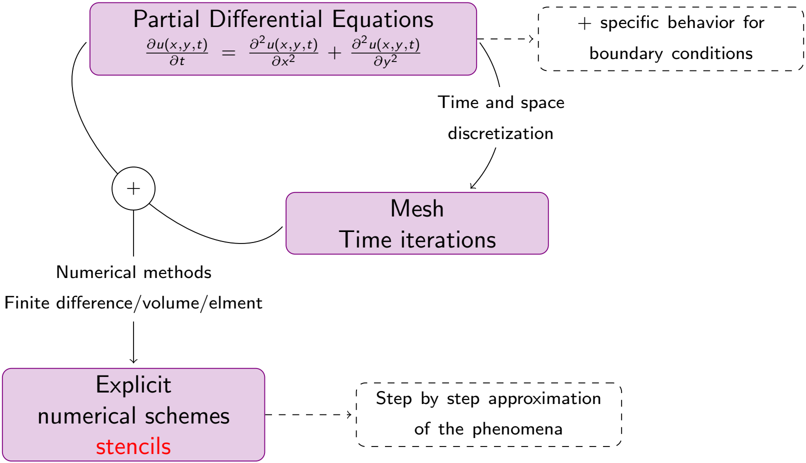
What if a DSL produces a component-based runtime ?

- Is it feasible ?
- Is it efficient ?
- Does it improve issues of DSLs ?
  - maintainability
  - portability
  - productivity

Let's take a useful example : *the Multi-Stencil Language* !

# Table of contents

# Numerical simulation = Multi-Stencil application



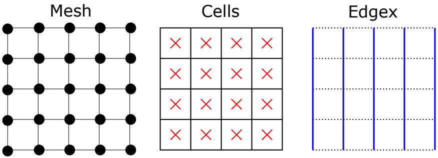
# Time and Mesh

## Time

At each time iteration of the simulation are applied the *computation kernels* of the application.

## Mesh

- A Mesh is a connected undirected graph $\mathcal{M} = (V, E)$ without bridges
- Mesh entities are a subset of $V \cup E$



# Data and Computation Kernels

## Data

Data is a set of numerical values, each one attached to a given mesh entity

## Computation kernel

- Set of data read for the computation
  - Each one associated to a stencil shape
- Data written by the computation
- A numerical expression
- A computation domain
  - Subset of mesh entities

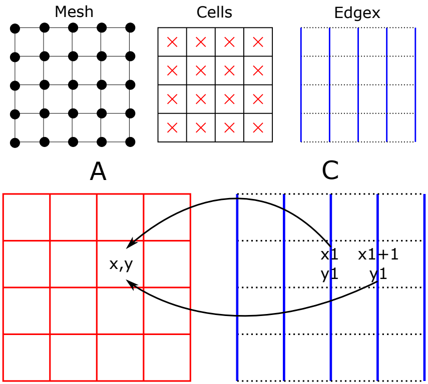
# Multi-Stencil program

$$\mathcal{MSP}(\mathcal{T}, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$

- $\mathcal{T}$ the set of time iterations to tun the simulation
- $\mathcal{M}$ the mesh of the simulation
- $\mathcal{E}$ the set of mesh entities
- $\mathcal{D}$ the set of computation domains
- $\Delta$ the set of data
- $\Gamma$ the set of computations

$=$ *the six sections of a Multi-Stencil Language program !*

# Example




$\mathcal{MSP}(\mathcal{T},\mathcal{M},\mathcal{E},\mathcal{D},\Delta,\Gamma)$

```
mesh: cart
mesh entities: cell,edgex
computation domains:
  allcell in cell
  alledgex in edgex
data:
  A,cell
  C,edgex
time:500
computations:
  A[allcell]=comp(C[n1])
```

# Multi-Stencil Language

## MSL is not

- a new stencil optimizer/compiler
- a new distributed data structure

## MSL is

- a high-level language for multi-stencil simulations
- agnostic from the type of mesh used (data structure)
- based on identifiers only

*MSL produces a ready-to-fill*
*component-based parallel pattern of the simulation !*

# Related Work

## Complementary work

- Distributed data structures : SkelGIS, Global Arrays
- Stencil DSLs (on grids) : Pochoir, PATUS
- Stencil DSLs (on unstructured meshes) : OP2, Liszt

## Similar work

- Pipeline of stencil computations for image processing : Halide
  - On grids (image), different abstraction level
- DSL to component-based runtime : ?


Julien Bigot (CEA), Hélène Coullon (INRIA), Christian Perez (INRIA)

# MSL to Component-based runtime

## Ready-to-fill parallel pattern

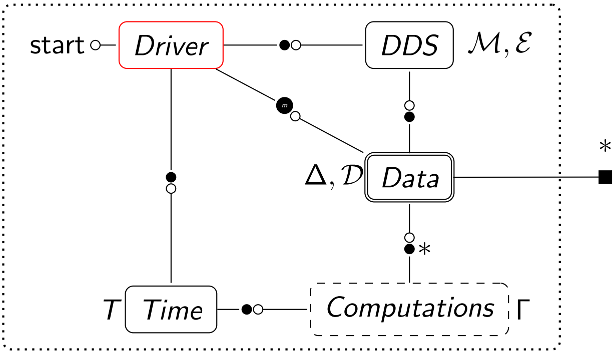
- Data parallelism
  - External distributed data structure
  - Automatic detection of synchronizations
- Task parallelism (mid-grain)
  - Compile a static scheduling of computation kernels

The fine grain task parallelism is left to other languages :

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

# MSL to Component-based runtime

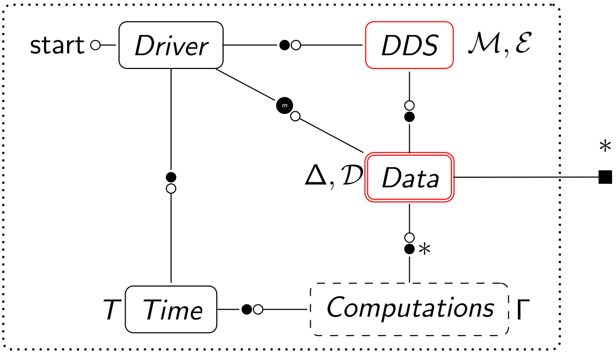
$$\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$




*Duplicated on each processor/core*

# MSL to Component-based runtime

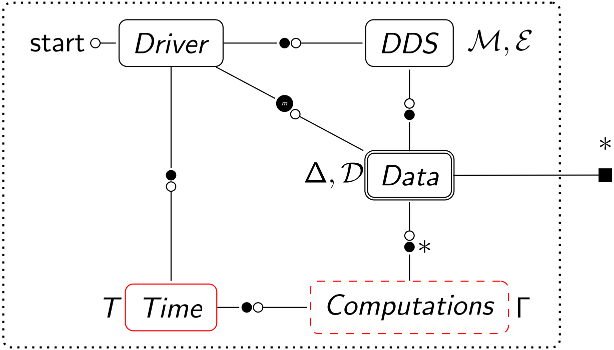
$$\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$




*Duplicated on each processor/core*

# MSL to Component-based runtime

$$\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$




*Duplicated on each processor/core*

# Example

```
mesh: cart
mesh entities: cell,edgex,edgey
computation domains:
  allcell in cell
  alledgex in edgex
  alledgey in edgey
  part1edgex in edgex
  part2edgex in edgex
data:
  a,cell
  b,cell
  c,edgex
  d,edgex
  e,edgey
  f,cell
  g,edgey
  h,edgex
  i,cell
  j,edgex
time:500
computations:
  b[allcell]=c0(a)
  c[alledgex]=c1(b[n1])
  d[alledgex]=c2(c)
  e[alledgey]=c3(c)
  f[allcell]=c4(d[n1])
  g[alledgey]=c5(e)
  h[alledgex]=c6(f)
  i[allcell]=c7(g,h)
  j[partedgex]=c8(i[n1])
```

# Data parallelism

1. Assembly of components duplicated on each resource
2. External Distributed Data Structure to split data among resources
3. Detect when synchronizations are needed

## Synchronization

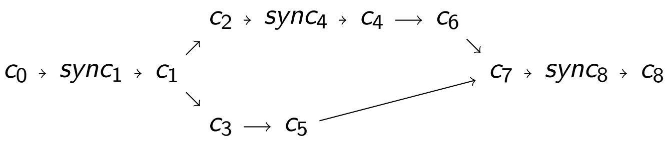
When a computation read a data, usign a stencil shape, that has been written by a previous computation.

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

# Data and task parallelism

## Dependency graph

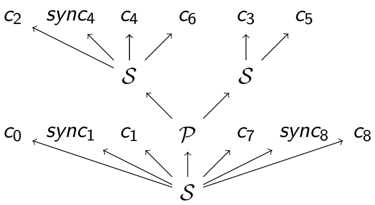
1. Each node is a computation or a synchronization
2. Each edge is a dependency : a computation read a data that has been written before.




Dynamic or static scheduling ?

# Series-Parallel Tree

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

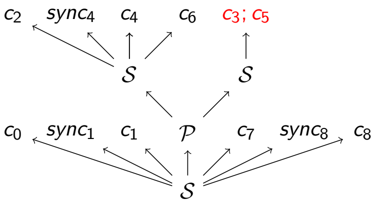



## Specific components

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

# Series-Parallel Tree

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

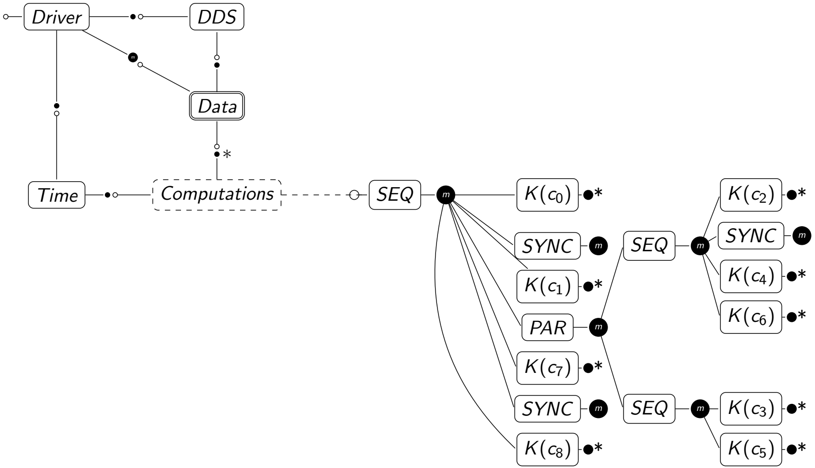



Loop fusion optimization possible

## Specific components

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

# Component-based runtime



# Resume

The MSL compiler can produces :

- A data parallel pattern of the multi-stencil application
- An hybrid (data + task) pattern of the multi-stencil application

# Resume

The MSL compiler can produces :

- A data parallel pattern of the multi-stencil application
- An hybrid (data + task) pattern of the multi-stencil application

# Implementation and evaluation

**Implementation of MSL** : Python, SkelGIS and $L^2C$

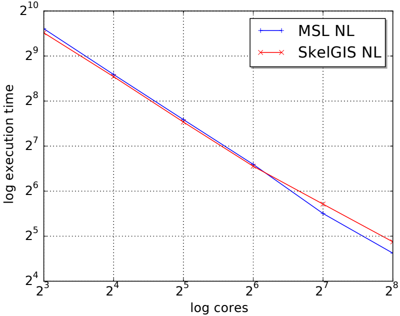
**Shallow-water equations** : 1 mesh, 3 mesh entities, 7 computation domains, 48 data, 98 computations (32 stencils, 66 local kernels)

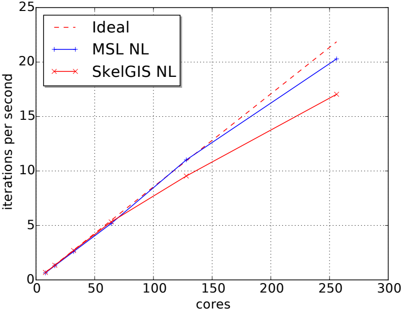
**Evaluation of the data parallelism**

- Full SkelGIS implementation (DDS + specific interfaces to hide communications)
- MSL implementation which uses the SkelGIS DDS
- Thin Nodes TGCC Curie : two 8-cores Intel Sandy Bridge 2.7GHz, 64GB RAM, Infiniband

# Evaluations

Mesh size : $10k \times 10k$ Number of iterations : 500



# Conclusion

## Conclusion

- A DSL for Multi-Stencil applications (MSL)
- The compilation of MSL to get data + task parallelism
- The dump to a component-based runtime
- Data parallelism evaluation : no overhead introduced

## Perspectives

- Scalability up to 32$k$ cores on TGCC Curie (CEA)
- Evaluations of the Data+Task parallelism (OpenMP 3)
- Dynamic scheduling (OpenMP 4), CPU+GPU (Pochoir etc.)

$\hookrightarrow$ Show portability, maintainability introduced by components