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)

16th November 2015

Motivation

+ Domain Specific Languages

- 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
  - Productivity and code-reuse
  - Separation of concerns

Motivation

Component models

- Divide an application into several functionnal black boxes
- Each component defines its connections with outer world
- Each component are built independently from the others
- 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 efficient ?
- Is it feasible ?
- Does it improve issues of DSLs ?
  - maintainability
  - portability
  - productivity

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

Table of contents

Multi-Stencil Language

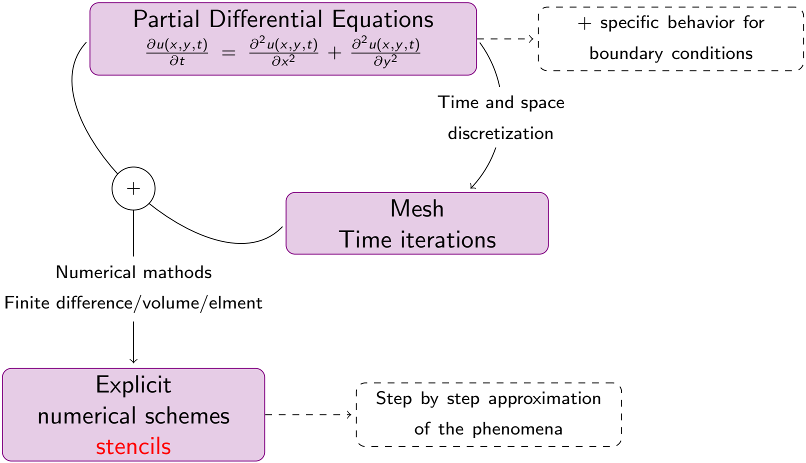
Overview

Compiler

Evaluation

Conclusion and perspectives

Numerical simulation = Multi-Stencil application



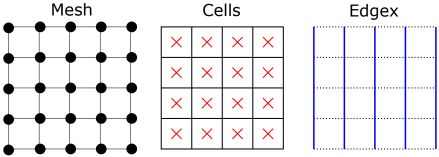
Time and Mesh

Time

At each time iteration of the simulation are applied the *computations 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

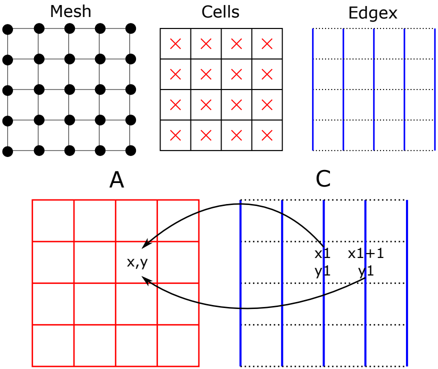
Data

A 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 computaion
- A numerical expression
- A computation domain
  - Subset of mesh entities

Computation kernel : Examples



Compute A using $\{(A, local), (C, n2)\}$ on the computation domain called $dc2$ with the numerical expression

$$A(x, y) = A(x, y) + C(x1, y1) + C(x1 + 1, y1).$$

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
- Δ the set of data
- Γ the set of computations

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

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
- DSL to component-based runtime : ?

MSL to Component-based runtime

Ready-to-fill parallel pattern

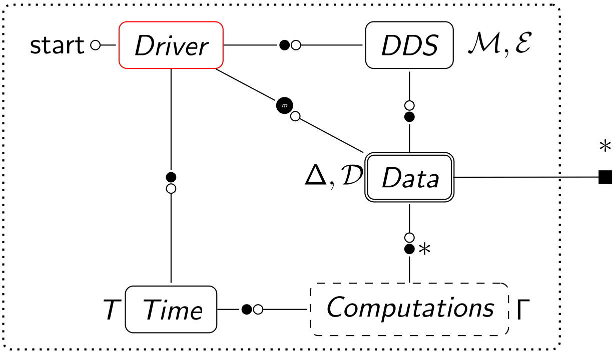
- Data parallelism
  - External distributed data structure
  - Automatic detection of synchronizations
- Task parallelism (mid-grain)
  - Compile a static scheduling of computation kernels

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

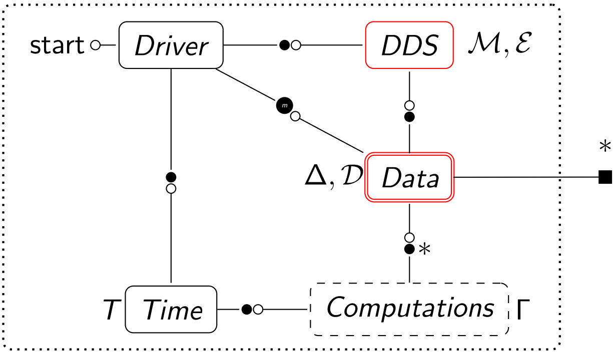
$$\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$



Duplicated on each processor/core

MSL to Component-based runtime

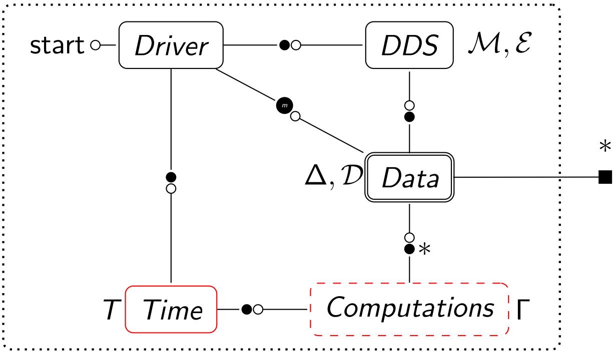
$$\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \Gamma)$$



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

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

Synchronization

For n the number of computations in Γ, and for i, j such that $i < j < n$, a *synchronization* is needed between c_i and c_j, denoted $c_i \lessdot c_j$, if c_j read a data written by c_i with a stencil shape and not locally.

Mid-grain task parallelism

Build the read/write dependency graph.

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

Data parallelism + Task parallelism

Which scheduling ?

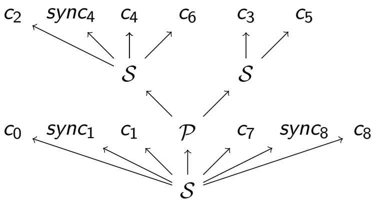
From the dependency graph we can :

- Use a dynamic scheduler
- Compute a static scheduling

In this work,

- We build a static scheduling of the multi-stencil application
- This static scheduling can be dumped to a component-based runtime

Series-Parallel Tree

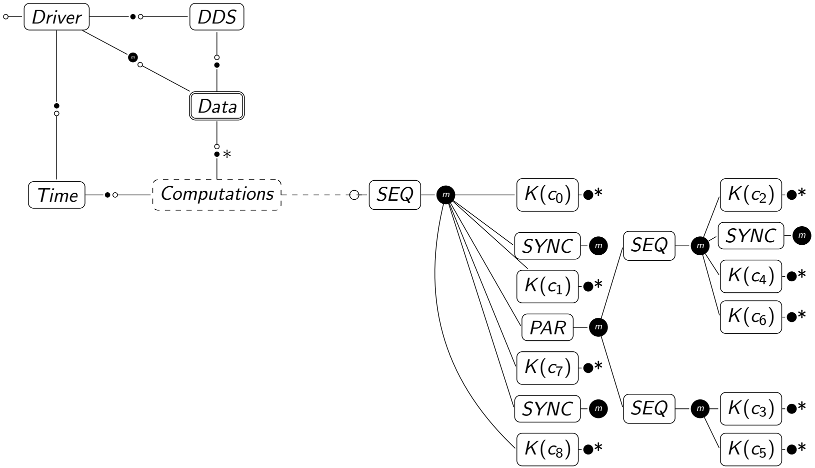


Specific components

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

Direct dump to the component assembly

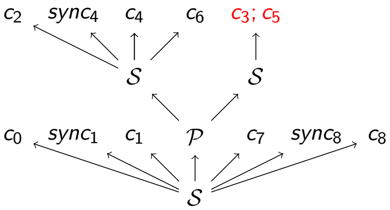
Component-based runtime



Back to Data Parallelism

Back to Data Parallelism only ?

Loop Fusion : Same computation domain under a sequence or a parallel node



Canonical form : $[c_0, sync_1, c_1, c_2, sync_4, c_4, c_6, \{c_3, c_5\}, c_7, sync_8, c_8]$

Implementation and evaluation

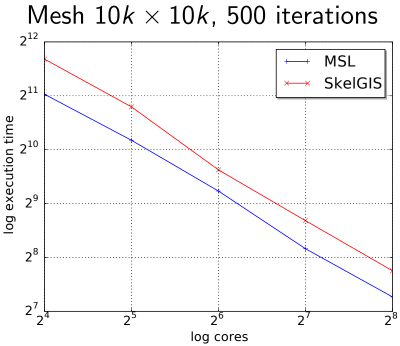
Implementation of MSL : Python, SkelGIS and L^2C

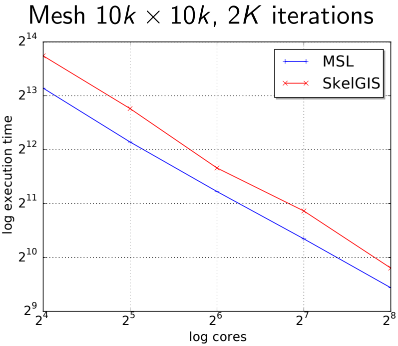
Shallow-water equations : 1 mesh, 3 mesh entities, 7 computation domains, 48 data, 98 computations (32 stencils, 66 local kernels)

Evaluation of the data parallelism

- Full SkelGIS implementation (DDS + specific interfaces to hide communications)
- MSL implementation which uses the SkelGIS DDS
- Cluster Edel of Grid'5000 : 2 8-cores Intel Xeon E5520, 24GB RAM / nodes

Evaluations





Conclusion

- A DSL for Multi-Stencil applications (MSL)
- The compilation of MSL to get a static scheduling
- The dump to a component-based runtime
- No overhead introduced by the solution

Perspectives

- A comparison with an improved version of SkelGIS
- More evaluations on TGCC Curie (CEA) up to 32k cores
- An evaluation of the Data+Task parallelism (mixing the static scheduling and OpenMP)
- A new dump with a component responsible for the dynamic scheduling of the application (using OpenMP 4.x)
- A new dump to CPU+GPGPUs (using existing stencil compilers Pochoir, PATUS etc.)
- An alternative to the SkelGIS DDS using Global Arrays (PGAS)

Thank You !