



**HAL**  
open science

## Extensibility and Composability of a Multi-Stencil Domain Specific Framework

Hélène Coullon, Julien Bigot, Christian Pérez

► **To cite this version:**

Hélène Coullon, Julien Bigot, Christian Pérez. Extensibility and Composability of a Multi-Stencil Domain Specific Framework. *International Journal of Parallel Programming*, 2017, 10.1007/s10766-017-0539-5 . hal-01650998

**HAL Id: hal-01650998**

**<https://hal.science/hal-01650998v1>**

Submitted on 11 Jan 2018

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Extensibility and Composability of a Multi-Stencil Domain Specific Framework

Hélène Coullon · Julien Bigot ·  
Christian Perez

the date of receipt and acceptance should be inserted later

**Abstract** As the computation power of modern high performance architectures increases, their heterogeneity and complexity also become more important. One of the big challenges of exascale is to reach programming models that give access to high performance computing (HPC) to many scientists and not only to a few HPC specialists. One relevant solution to ease parallel programming for scientists is Domain Specific Language (DSL). However, one problem to avoid with DSLs is to mutualize existing codes and libraries instead of implementing each solution from scratch. For example, this phenomenon occurs for stencil-based numerical simulations, for which a large number of languages has been proposed without code reuse between them. The Multi-Stencil Framework (MSF) presented in this paper combines a new DSL to component-based programming models to enhance code reuse and separation of concerns in the specific case of stencils. MSF can easily choose one parallelization technique or another, one optimization or another, as well as one back-end implementation or another. It is shown that MSF can reach same performances than a non component-based MPI implementation over 16.384 cores. Finally, the performance model of the framework for hybrid parallelization is validated by evaluations.

---

Hélène Coullon  
DAPI IMT Atlantique, LS2N, Inria. Nantes, France  
E-mail: helene.coullon@inria.fr

Julien Bigot  
Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay,  
91191 Gif-sur-Yvette, France  
E-mail: julien.bigot@cea.fr

Christian Perez  
Univ. Lyon, Inria, CNRS, ENS de Lyon. Lyon, France  
E-mail: christian.perez@inria.fr

**Keywords** Component programming models · Domain Specific Language (DSL) · Stencil · Numerical simulation · Data parallelism · Task parallelism · Scheduling · MPI · OpenMP

## 1 Introduction

As the computation power of modern high performance architectures increases, their heterogeneity and complexity also become more important. For example, the current fastest supercomputer Tianhe-2 <sup>1</sup> is composed of multi-cores processors and accelerators, and is able to reach a theoretical peak performance of about thirty peta-flops (floating-point operations per second). However, to be able to use such machines, multiple programming models, such as MPI (Message Passing Interface), OpenMP, CUDA, etc., and multiple optimization techniques, such as cache optimization, have to be combined. Moreover, current architectures evolution seems to indicate that heterogeneity and complexity in HPC will continue to grow in the future.

One of the big challenges to be able to use those upcoming Exascale computers is to propose programming models that give access to high performance computing (HPC) to many scientists and not only to a few HPC specialists [15]. Actually, applications that run on supercomputers and need such computation power (*e.g.* physics, weather or genomic) are typically not implemented by HPC specialists but by domain scientists.

Many general purpose languages and frameworks have improved the simplicity of writing parallel codes. For example PGAS models [23] or task-based frameworks, such as OpenMP [13], Legion [4] or StarPU [2], partially hide intricate details of parallelism to the user. For non-expert users however, these languages and frameworks are still difficult to use. Moreover, tuning an application for a given architecture is still very complex to achieve with these solutions. An interesting approach that combines simplicity of use, due to a high abstraction level, with efficient execution are domain specific languages (DSL) and domain specific frameworks (DSF). These solutions are specific to a given domain and propose a grammar or an API which is easy to understand for specialists of this domain. Moreover, knowledge about the targeted domain can be embedded in the compiler that can thus automatically apply parallelization and optimization techniques to produce high performance code. Domain specific solutions are therefore able to separate end-user concerns from HPC concerns which is a requirement to make HPC accessible to a wider audience.

Many domain specific languages and frameworks have been proposed. Each one claims to handle a distinct specific optimization or use case. Each solution is however typically re-implemented from scratch. In this paper, we claim that the sharing of common building blocks when designing DSLs or DSFs would increase re-use, flexibility and maintainability in their implementation. It would also ease the creation of approaches and applications combining multiple DSLs and DSFs.

---

<sup>1</sup> [www.top500.org](http://www.top500.org)

For example, some of the approaches to numerically solve partial differential equations (PDEs) lead to *stencil computations* where the values associated to one point in space at a given time are computed from the values at the previous time at the exact same location together with a few neighbor locations. Many DSLs have been proposed for stencil computations [7, 8, 14, 26, 30] as detailed in Section 7. Many of them use the same kind of parallelization, data structures or optimization techniques, however each one has been built from scratch.

We propose the Multi-Stencil Framework (MSF) that is built upon a meta-formalism of multi-stencil simulations. MSF produces a parallel orchestration of a multi-stencil program without being aware of the underlying implementation choices (*e.g.*, distributed data structures, task scheduler etc.). Thanks to this meta-formalism MSF is able to easily switch from one parallelization technique to another and from one optimization to another. Moreover, as MSF is independent from implementation details, MSF can easily choose one back-end or another, thus easing code reuse of existing solutions. To ease composition of existing solutions, MSF is based on component-based programming [29], where applications are defined as an assembly of building blocks, or components.

After a short overview of the Multi-Stencil Framework given in Section 2, the paper is organized as follows. The meta-formalism of a multi-stencil program is presented in Section 3; from this formalism are built both a light and descriptive domain specific language, namely MSL, as well as a generic component assembly of the application both described in Section 4; the compiler of the framework is described in Section 5; finally a performance evaluation is detailed in Section 6 .

## 2 The Component-Based Multi-Stencil Framework

This section first presents a background on component models and particularly on the Low Level Components. This background is needed to understand the second part of the section which gives an overview of the overall Multi-Stencil Framework (MSF).

### 2.1 Background on component models

Component-based software engineering (CBSE) is a domain of software engineering [29] which aims at improving code re-use, separations of concerns, and thus maintainability. A component-based application is made of a set of component instances linked together, this is also called a component *assembly*. A component is a black box that implements an independent functionality of the application, and which interacts with its environment only through well defined interfaces: its ports. For example, a port can specify services provided or required by the component. With respect to high performance computing, some works have also shown that component models can achieve the needed

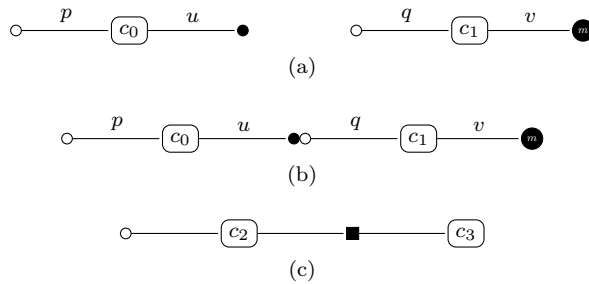


Fig. 1: Example of components and their ports representation. a) Component  $c_0$  has a provide port ( $p$ ) and a use port ( $u$ ); Component  $c_1$  has also a provide port ( $q$ ) but also a use multiple port ( $v$ ). b) A use port is connected to a (compatible) provide port. c) Component  $c_2$  and  $c_3$  shares an MPI communicator.

level of performance and scalability while also helping in application portability [1, 6, 27].

Many component models exist, each of them with its own specificities. Well known component models include, for example, the CORBA Component Model (CCM) [24], and the Grid Component Model (GCM) [3] for distributed computing, while the Common Component Architecture (CCA) [1], and Low Level Components (L<sup>2</sup>C) [5] are HPC-oriented. This work makes use of L<sup>2</sup>C for the experiments.

L<sup>2</sup>C [5] is a minimalist C++ based HPC-oriented component model where a component extends the concept of class. The services offered by the components are specified through *provide* ports, those used either by *use* ports for a single service instance, or *use – multiple* ports for multiple services instances. Services are specified as C++ interfaces. L<sup>2</sup>C also offers *MPI* ports that enable components to share MPI communicators. Finally, components can also have attribute ports to be configured. In this paper, and as illustrated in Figure 1, a *provide* port is represented by a white circle, a *use* port by a black circle, a *use – multiple* port by a black circle with a white  $m$  in it. MPI ports are connected with a black rectangle. A L<sup>2</sup>C-based application is a static *assembly* of components instances and the connections between their ports. Such an assembly is described in LAD, an XML dialect, and is managed by the L<sup>2</sup>C runtime system that minimize overheads by loading simple dynamic libraries. One can also notice that L<sup>2</sup>C can achieve performance if the granularity of components is high enough and attentively chosen by the user. The typical overhead of a L<sup>2</sup>C is a C++ indirect virtual method invocation.

## 2.2 Multi-Stencil Framework overview

The Multi-Stencil Framework helps end-users to produce high performance parallel applications for the specific case of multi-stencils. The multi-stencil

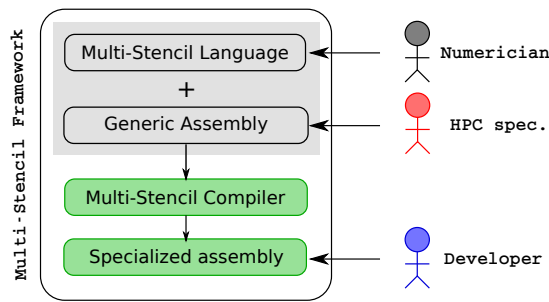


Fig. 2: The Multi-Stencil Framework (MSF) is composed of the Multi-Stencil Language (MSL), the Generic Assembly (GA) and the Multi-Stencil Compiler (MSC) to produce a specialized assembly of components. The numerician, or mathematician uses MSL to describe its simulation. The developer will implement components responsible for numerical codes. A third party HPC specialist can interact with MSF to propose different version of HPC components.

domain will be formally defined in the next section. A multi-stencil program numerically solves PDEs using computations that can use neighborhood values around an element, also called a *stencil* computation.

Figure 2 gives an overview of the Multi-Stencil Framework that is entirely detailed throughout this paper. It is composed of four distinct parts described hereafter. As illustrated in Figure 2, MSF targets two different kinds of end-users: the *numerician*, in other words the mathematician, and the *developer*. Most of the time numericians do not have programming knowledge, however as it is not their core domain and because of a lack of time, development is often left to engineers according to numerician needs. MSF has the interesting particularity to propose a clear separation of concerns between these two end-users by distinguishing the description of the simulation from the implementation of numerical codes.

MSF also has the interesting capability to be more flexible than existing solutions thanks to a possibility for a third party to interact with the framework. This third party is a High Performance Computing (HPC) specialist as displayed in Figure 2.

**Multi-Stencil Language** The Multi-Stencil Language, or MSL, is the domain specific language proposed by the framework for the numerician. It is a descriptive language, easy to use, without any concern about implementation details. It fits the need of a mathematician to describe the simulation. The description written with MSL can be considered as an input of the framework. MSL is described in details in Section 4. The language is built upon the formalism described in Section 3.

**Generic Assembly** In addition to the language MSL, used by the numerician to describe its simulation, MSF needs a Generic Assembly (GA) of a

multi-stencil program as input. What is called a GA is a component assembly for which meta-types of components are represented and for which some parts need to be generated or specialized. A GA could be compared respectively to a template or a skeleton in object programming languages (such as C++) or functional languages. From this generic assembly will be built the final specialized assembly of the simulation where component types will be specified, and where parts of GA will be transformed. As well as MSL, this generic assembly is described in Section 4 and is built upon the meta-formalism described in Section 3.

**Multi-Stencil Compiler** The core of the framework is the Multi-Stencil Compiler, or MSC. It is responsible for transforming the generic assembly into the final parallel assembly which is specific to the simulation described by the numerician with MSL. MSC is described in Section 5.

**Specialized assembly** Finally, the output of MSF is the component assembly generated by MSC. It is an instantiation and a transformation of the generic component assembly, by adding component types, transforming some part of the assembly, and by adding specific components generated by MSC. From this final component assembly which is specific to the simulation initially described with MSL, the developer will finally write components associated to numerical codes, or directly re-use existing components from other simulations. This final specialized component assembly is a parallel orchestration of the computations of the simulation initially described by the numerician. Finally, the specialized assembly produced by MSF is written in L<sup>2</sup>C.

### 3 Formalism of a Multi-Stencil Program

The numerical solving of partial differential equations relies on the discretization of the continuous time and space domains. Computations are typically iteratively (time discretization) applied onto a mesh (space discretization). While the computations can have various forms, many direct methods can be expressed using three categories only: *stencil* computations involve access to neighbor values only (the concept of neighborhood depending on the space discretization used); *local* computations depend on the computed location only (this can be seen as a stencil of size one); finally, *reductions* enable to transform variables mapped on the mesh to a single scalar value.

This section gives a complete formal description of what we call a *multi-stencil program* and its computations. This formalism is general enough to be common to any existing solution already proposed for stencil computations. As a result it can be considered as a meta-formalism or a meta-model of a Multi-Stencil Program. This meta-formalism will be used to define MSL and GA in the next section.

### 3.1 Time, mesh and data

Let us introduce some notations.  $\Omega$  is the continuous space domain of a numerical simulation (typically  $\mathbb{R}^n$ ). A mesh  $\mathcal{M}$  defines the discretization of the continuous space domain  $\Omega$  and is defined as follows.

**Definition 1** *A mesh is a connected undirected graph  $\mathcal{M} = (V, E)$ , where  $V \subset \Omega$  is the (finite) set of vertices and  $E \subseteq V^2$  the set of edges. The set of edges  $E$  of a mesh  $\mathcal{M} = (V, E)$  does not contain bridges. It is said that the mesh is applied onto  $\Omega$ .*

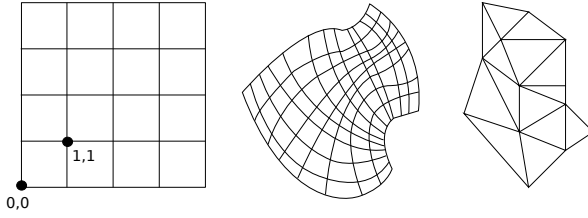


Fig. 3: From left to right, Cartesian, curvilinear and unstructured meshes.

A mesh can be structured (as Cartesian or curvilinear meshes), unstructured, regular or irregular (without the same topology for each element) as illustrated in Figure 3.

#### Definitions (mesh)

- An *entity*  $\phi$  of a mesh  $\mathcal{M} = (V, E)$  is defined as a subset of its vertices and edges,  $\phi \subset V \cup E$ .
- A *group of mesh entities*  $\mathcal{G} \in \mathcal{P}(V \cup E)$  represents a set of entities of the same topology.
- The *set of all groups of mesh entities* used in a simulation is denoted  $\Phi$ .

For example, in a 2D Cartesian mesh, an entity could be a cell, made up of four vertices and four edges. A group of entities could contain all the cells, another would for example contain the vertical edges at the frontier between cells. Both groups would be part of  $\Phi$ . This example is illustrated in Figure 4a.

**Definition 2** *The finite sequence  $T : (t_n)_{n \in \llbracket 0, T_{max} \rrbracket}$  represents the discretization of the continuous time domain  $\mathcal{T} = \mathbb{R}$ .*

The time discretization can be as simple as a constant time-step with a fixed number of steps. The time-step and the number of steps can also change on the fly during execution.

#### Definitions (quantity)



- $\Delta$  are the *mesh variables*. A mesh variable  $\delta \in \Delta$  associates to each couple entity and time-step a value  $\delta : \mathcal{G} \times T \mapsto \mathcal{V}_\delta$  where  $\mathcal{V}_\delta$  is a value type.
- The group of entities a variable is mapped on is denoted  $entity(\delta) = \mathcal{G}$ .
- $\mathcal{S}$  are the *scalar variables*. A scalar variable  $s \in \mathcal{S}$  associates to each time-step a value  $s : T \mapsto \mathcal{V}_s$  where  $\mathcal{V}_s$  is a value type.
- $\mathbb{V} = \Delta \cup \mathcal{S}$  is the set of *variables* or *quantities*.
- Among the scalar variables is one specific boolean variable  $conv \in \mathcal{S}$ , the convergence criteria, whose value is 0 except at the last step where it is 1. This scalar can be updated on the fly according to other variables, typically by using a reduction as detailed later.

### 3.2 Computations

#### Definitions

- A computation domain  $D$  is a subpart of a group of mesh entities,  $D \subseteq \mathcal{G} \in \Phi$ .
- The set of computation domains of a numerical simulation is denoted  $\mathcal{D}$ .
- $\mathcal{N}$  is the set of neighborhood functions  $n : \mathcal{G}_i \mapsto \mathcal{G}_j^m$  which for a given entity  $\phi \in \mathcal{G}_i$  returns a set of  $m$  entities in  $\mathcal{G}_j$ . One can notice that  $i = j$  is possible. Most of the time, such a neighborhood is called a *stencil shape*.

**Definition 3** A computation kernel  $k$  of a numerical simulation is defined as  $k = (S, R, (w, D), comp)$ , where

- $S \in \mathcal{S}$  is the set of scalar to read,
- $w \in \mathbb{V}$  is the single quantity (variable) modified by the computation kernel,
- $D$  is the computation domain on which  $w$  is computed,  $D \subseteq entity(w)$ , or is null if  $w \in \Delta$ ,
- $R \in \Delta \times \mathcal{N}$  is the set of tuples  $(r, n)$  representing the data read where  $r$  is a mesh variable read by the kernel to compute  $w$ , and  $n : entity(w) \rightarrow entity(r)^m$  is a neighborhood function that indicates which entity of  $r$  are read to compute  $w$ .
- Finally,  $comp$  is the numerical computation which returns a value from a set of  $n$  input values,  $comp : \mathcal{V}_i^n \rightarrow \mathcal{V}_j$ , where  $\mathcal{V}_i$  and  $\mathcal{V}_j$  are value types. Thus,  $comp$  represents the actual numerical expression computed by a kernel.

In a Multi-Stencil simulation, at each time-step, a set of computations is performed. During a computation kernel, it can be considered that a set of old states  $(t - 1)$  of quantities are read ( $S$  and  $R$ ), and that a new state  $(t)$  of a single quantity is written ( $w$ ). Such a definition of a computation kernel covers a large panel of different computations. For example, the four usual types of computations (stencil, local, boundary and reduction) performed into a simulation can be defined as follow :

- A computation kernel  $k(S, R, (w, D), comp)$  is a *stencil kernel* if  $\exists (r, n) \in R$  such that  $n \neq identity$ .

- A *boundary kernel* is a kernel  $k(S, R, (w, D), comp)$  where  $D$  is a specific computation domain at the border of entities, and which does not intersect with any other computation domain.
- A computation kernel  $k(S, R, (w, D), comp)$  is a *local kernel* if  $\forall(r, n) \in R, n = identity$ .
- A computation kernel  $k(S, R, (w, D), comp)$  is a *reduction kernel* if  $w$  is a scalar. A reduction can for example be used to compute the convergence criteria of the time loop of the simulation.

Since we only consider explicit numerical schemes in this paper, a kernel cannot write the same quantity it reads, except locally, *i.e.* if  $\exists(w, n)$  where  $w \in R \Rightarrow n = identity$ .

It could seem counter-intuitive to restrict kernels to the computation of a single quantity. As a matter of fact, one often performs multiple computations in a single loop, for example for performance reasons (cache optimization, temporary computation factorization, etc.) or for readability (code length, semantically close computations, etc.). One can however notice that it is always possible to re-express a computation modifying  $n$  quantities as  $n$  computations modifying a single quantity each. Both approaches are therefore equivalent from an expressiveness point of view.

Modifying multiple quantities in a single loop nest does however not always improve performance. For example, it reduces the number of concurrent tasks available and limits the potential efficiency on parallel resources as will be shown in Section 6. We therefore introduce the concept of *fusion* in Section 5 where multiple logical kernels can be executed in a single loop nest that modifies multiple quantities. This transformation is much easier to implement than splitting a kernel would be, leaving more execution choices open.

In addition, the modification of multiple quantities in a single loop nest can lead to subtle ordering errors when executing in parallel as it will be discussed in Section 5.4. Automatically detecting kernels that can be fused instead of leaving this to the responsibility of the domain scientist avoids these potential errors. We have therefore chosen to restrict kernels to the computation of a single quantity.

**Definition 4** *The set of  $n$  ordered computation kernels of a numerical simulation is denoted  $\Gamma = [k_i]_{0 \leq i \leq n-1}$ , such that  $\forall k_i, k_j \in \Gamma$ , if  $i < j$ , then  $k_i$  is computed before  $k_j$ .*

**Definition 5** *A multi-stencil program is defined by the octuplet*

$$\mathcal{MSP}(\mathcal{M}, \Phi, \mathcal{D}, \mathcal{N}, \Delta, \mathcal{S}, T, \Gamma) \quad (1)$$

**Example** For example, in Figure 4b, assuming that the computation domain (full lines) is denoted  $dc1$  and the stencil shape described by the neighborhood function is  $n1$ , the stencil kernel can be defined as:

$$R : \{(B, n1)\}, \quad w : A, \quad D : dc1,$$

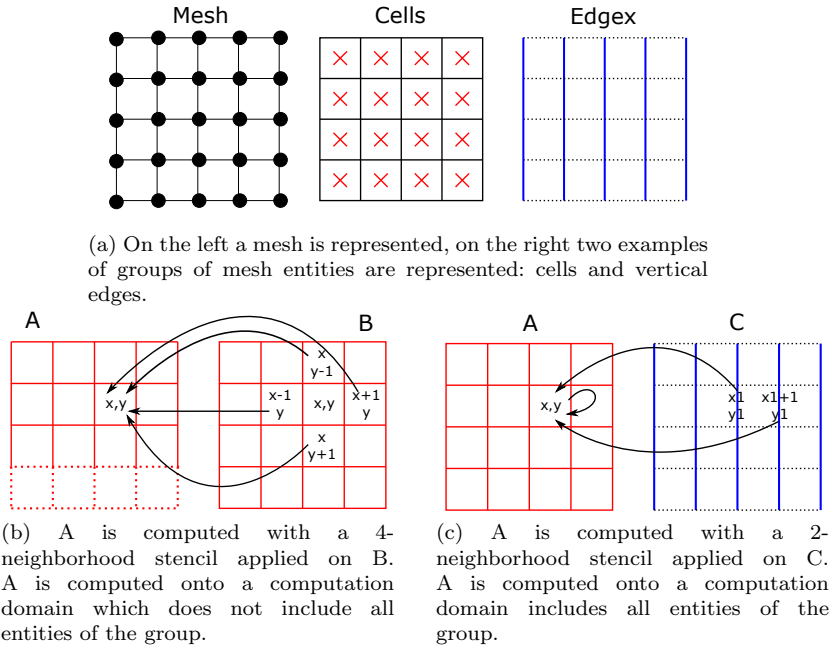


Fig. 4: (a) a Cartesian mesh and two kind of groups of mesh entities, (b) an example of stencil kernel on cells, (c) an example of stencil kernel on two different groups of mesh entities.

$$\text{comp} : A(x, y) = B(x + 1, y) + B(x - 1, y) + B(x, y + 1) + B(x, y - 1).$$

On the other hand, in the example of Figure 4c, assuming the computation domain is  $dc2$  and the stencil shape is  $n2$ , the stencil kernel is defined as:

$$R : \{(C, n2), (A, \text{identity})\}, \quad w : A, \quad D : dc2,$$

$$\text{comp} : A(x, y) = A(x, y) + C(x1, y1) + C(x1 + 1, y1).$$

In this section, we have formally defined a stencil program. This formalism is mainly composed of a mesh abstraction and a simple definition of computation. In fact, this formalism is generic enough to be common to many existing modelizations of a stencil computation or a stencil simulation. Thus, the formalism summarized by Equation (1) can be compared to a meta-model of a multi-stencil program. In the next section, we use this meta-model (or meta-formalism) to define both a the domain specific language MSL, and the generic assembly of a multi-stencil program GA.

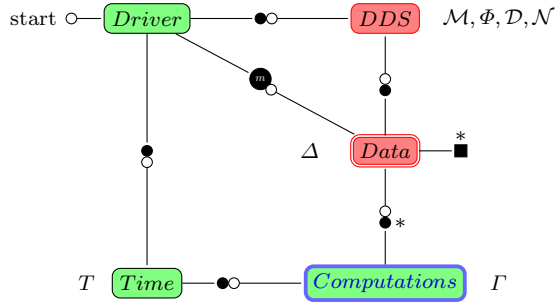


Fig. 5: Generic Assembly according to the Multi-Stencil program formalism. Components circled by a double line identify those that will be instantiated multiple times by MSC. Component colors represent actors of Figure 2 responsible for the component implementation: green for those automatically generated by the compiler, red for those implemented by HPC specialists and blue for those implemented by the developer.

#### 4 Generic Assembly and The Multi-Stencil Language

From the octuplet of Equation (1) both the Generic Assembly (GA) of a multi-stencil program and the Multi-Stencil Language (MSL) can be built. GA and MSL are both described in this section.

##### 4.1 Generic Assembly

As illustrated in Figure 5, the GA has five components: *Driver*, *Time*, *DDS*, *Data*, and *Computations*. These components are generic components or abstract components. It means that interfaces of these components are well defined but that they are not implemented yet in GA. They can be compared to abstract classes and templates in C++ for which an implementation must be given as well as specific parameters.

**Driver** This component can be compared to the main function of a usual program. It is responsible for both the initialization and the execution of other components (like variable initialization and function calls). This component is generated by MSC (represented in green).

**Time** This component is responsible for the time  $T$  defined in Equation (1). It is composed of a time loop and potentially of a convergence reduction. This component is generated by MSC (represented in green).

**DDS** This component is responsible for the mesh and its entities  $\mathcal{M}$  and  $\Phi$ , the set of computation domains  $\mathcal{D}$ , and the set of neighborhood functions  $\mathcal{N}$ . When the generic assembly is instantiated and specialized by MSC, an implementation of DDS is selected to handle a specific type of mesh. The interfaces exposed by this component are well defined and any component providing these interfaces can be indifferently used. A third party specialist can therefore propose new implementation of DDS. In this paper, both data and task parallelism are used. In the case of data parallelism DDS handles mesh partitioning and provides a synchronization interface as detailed in Section 5. The implementation of this component is the responsibility of HPC specialists (represented in red).

**Data** This component is responsible for the set of mesh variables  $\Delta$ . Each instance of the component uses the DDS component to handle one single mesh variable. It is closely related to *DDS* and its implementation is typically provided by the same HPC specialist as *DDS* (represented in red).

**Computations** This component is responsible for  $\Gamma$ , *i.e.*, the computations of the simulation. It is automatically replaced by a sub-assembly of components produced by MSC for which the parallel part is automatically generated. On the other hand, components responsible for the numerical kernels are filled by the developer. This is why this component is represented in green and blue in Fig. 5. The sub-assembly generation is described in Section 5.

## 4.2 The Multi-Stencil Language

The second element of MSF which is built upon the meta-model represented by Equation (1) is the Multi-Stencil Language and its grammar. This grammar is light and descriptive only. However it is sufficient (in addition to GA) for MSC to automatically extract a parallel pattern of the simulation, which is finally dumped as a specialized instantiation of GA.

The grammar of the Multi-Stencil Language is given in Figure 6 and an example is provided in Figure 7. A Multi-Stencil program is composed of eight parts that match those of Equation (1).

1. The **mesh** keyword (Fig. 6, l.1) introduces an identifier for  $\mathcal{M}$ , the single mesh of the simulation. For example **cart** in Fig. 7, l.1. The language, based on the meta-model is independent of the mesh topology, thus this identifier is actually not used by the compiler.
2. The **mesh entities** keyword (Fig. 6, l.2) introduces identifiers for the groups of mesh entities  $\mathcal{G} \in \Phi$ . For example **cell** or **edgex** in Fig. 7, l.2.
3. The **computation domains** keyword (Fig. 6, l.3) introduces identifiers for the computation domains  $D \in \mathcal{D}$ . For example **d1** and **d2** in Fig. 7, l.4-5. For reference, each domain is associated to a group of entities (Fig. 6, l.12) such as **cell** for **d1** in Fig. 7, l.4.

```

1  program ::= "mesh:" meshid
2           "mesh_entities:" listgroup
3           "computation_domains:" listcompdom
4           "independent:" listinde
5           "stencil_shapes:" liststencil
6           "mesh_quantities:" listquantities
7           "scalars:" listscalar
8           listloop
9
10 listgroup ::= groupid "," listgroup | groupid
11 listcompdom ::= compdom listcompdom | compdom
12 compdom ::= compdomid "in" groupid
13 listinde ::= inde listinde | inde
14 inde ::= compdomid "and" compdomid
15 liststencil ::= stencil liststencil | stencil
16 stencil ::= stencilid "from" groupid "to" groupid
17 listquantities ::= quantity listquantities | quantity
18 quantity ::= groupid listquantityid
19 listquantityid ::= quantityid "," listquantityid | quantityid
20 listscalar ::= scalarid "," listscalar | scalarid
21 listloop ::= loop listloop | loop
22 loop ::= "time:" iteration
23         "computations:" listcomp
24 iteration ::= num | scalarid
25 listcomp ::= comp listcomp | comp
26 comp ::= written "=" compid "(" listread ")"
27 written ::= quantityid "[" compdomid "]" | scalar
28 listread ::= dataread listread | dataread
29 dataread ::= quantityid "[" stencilid "]" | quantityid | scalar

```

Fig. 6: Grammar of the Multi-Stencil Language.

4. The **independent** keyword (Fig. 6, l.4) offers a way to declare that computation domains do not intersect, such as **d1** and **d2** in Fig. 7, l.7. This is used by the compiler to compute dependencies between computations.
5. The **stencil shapes** keyword (Fig. 6, l.5) introduces identifiers for each stencil shape  $n \in \mathcal{N}$ . For each  $n$ , the source and destination group of entities (Fig. 6, l.16) are specified. For example **nec** in Fig. 7, l.11 is a neighborhood from **edgex** to **cell**.
6. The **mesh quantities** keyword (Fig. 6, l.6) introduces identifiers for  $\delta \in \Delta$ , the mesh variables with the group of entities they are mapped on (Fig. 6, l.16). For example the quantities **C** and **H** are mapped onto the groups of mesh entities **edgex**.
7. The **scalars** keyword (Fig. 6, l.7) introduces identifiers for  $s \in \mathcal{S}$ , the scalars. For example **mu** and **tau** in Fig. 7, l.15.
8. Finally, the last part (Fig. 6, l.8) introduces the different computation loops of the simulation. Each loop is made of two parts:
  - the **time** keyword (Fig. 6, l.22) introduces either a constant number of iterations or *conv*, the convergence criteria that is a scalar (Fig. 6, l.24). For example, 500 iterations are specified in Fig. 7, l.16,

```

1 mesh : cart
2 mesh entities : cell , edgex
3 computation domains :
4   d1 in cell
5   d2 in edgex
6 independent :
7   d1 and d2
8 stencil shapes :
9   ncc from cell to cell
10  nec from cell to edgex
11  nec from edgex to cell
12 mesh quantities :
13   cell A,B,D,E,F,G,I , J
14   edgex C,H
15 scalars : mu, tau
16 time : 500
17 computations :
18   B[d1] = k0(tau ,A)
19   C[d2] = k1(B[nec])
20   D[d1] = k2(C)
21   E[d1] = k3(C)
22   F[d1] = k4(D,C[nce])
23   G[d1] = k5(mu,tau ,E)
24   H[d2] = k6(F)
25   I[d1] = k7(G,H)
26   J[d1] = k8(mu, I[ncc])

```

Fig. 7: Example of program using the Multi-Stencil Language.

- the `computations` keyword (Fig. 6, l.23) introduces identifiers for each computation  $k = (S, R, (w, D), comp) \in \Gamma$ . Each computation (Fig. 6, l.26) specifies:
  - the quantity  $w$  written and its domain  $D$ , for example in Fig. 7, l.22, kernel `k4` computes the variable `F` on domain `d1`,
  - the read scalars  $S$  and mesh variables with their associated stencil shape ( $R$ ). For example in Fig. 7, l.16, `k4` reads `C` with the shape `ncc` and `D` with the default *identity* shape; it does not read scalars.

One can notice that in the example of Figure 7, there are no kernel associated to the scalars `mu` and `tau` (reduction). In this case, those scalars are in fact constants. One can also notice that the computation to execute for each kernel is not specified. Only an identifier is given to each kernel, for example  $k_4$ . The numerical code is indeed not handled by MSL that generates a parallel orchestration of computations only. The numerical computation is specified after MSC compilation by the developer (Fig. 2).

## 5 The Multi-Stencil Compiler

In a computation  $k(S, R, (w, D), comp)$ , the `comp` part is provided by the developer after the MSC compilation phase. This part does therefore not

have any impact on compilation concerns. Thus, to simplify notations in the rest of this paper, we use the shortcut notation  $k(S, R, (w, D))$  instead of  $k(S, R, (w, D), comp)$ .

### 5.1 Data parallelism

In a data parallelization technique, the idea is to split data, or quantities, on which the program is computed into sub-domains, one for each execution resource. The same program is applied to each sub-domain simultaneously with some additional synchronizations to ensure coherence.

More formally, the data parallelization of a multi-stencil program of equation (1) consists in a partitioning of the mesh  $\mathcal{M}$  in  $p$  sub-meshes  $\mathcal{M} = \{\mathcal{M}_0, \dots, \mathcal{M}_{p-1}\}$ . This step can be performed by an external graph partitioner [11, 21, 25] and is handled by the DDS implementation of the third party HPC specialist.

As entities and quantities are mapped on the mesh, the set of groups of mesh entities and the set of quantities  $\Delta$  are partitioned the same way as the mesh:  $\Phi = \{\Phi_0, \dots, \Phi_{p-1}\}$ ,  $\Delta = \{\Delta_0, \dots, \Delta_{p-1}\}$ .

The second step of the parallelization is to identify in  $\Gamma$  the synchronizations required to update data. It leads to the construction of a new ordered list of computations  $\Gamma_{sync}$ .

**Definition 6** For  $n$  the number of computations in  $\Gamma$ , and for  $i, j$  such that  $i < j < n$ , a synchronization is needed between  $k_i$  and  $k_j$ , denoted  $k_i \ll k_j$ , if  $\exists (r_j, n_j) \in R_j$  such that  $w_i = r_j$  and  $n_j \neq \text{identity}$  ( $k_j$  is a stencil computation). The quantity to synchronize is  $\{w_i\}$ .

A synchronization is needed for the quantity read by a stencil computation (not local), if this quantity has been written before. This synchronization is needed because a neighborhood function  $n \in \mathcal{N}$  of a stencil computation involves values computed on different resources.

However, as a multi-stencil program is an iterative program, computations that happen after  $k_j$  at the time iteration  $t$  have also been computed before  $k_j$  at the previous time iteration  $t - 1$ . For this reason another case of synchronization has to be defined.

**Definition 7** For  $n$  the number of computations in  $\Gamma$  and  $j < n$ , if  $\exists (r_j, n_j) \in R_j$  such that  $n_j \neq \text{identity}$  and such that for all  $i < j$ ,  $k_i \not\ll k_j$ , a synchronization is needed between  $k_i^{t-1}$  and  $k_j^t$ , where  $j < l < n$ , denoted  $k_i^{t-1} \ll k_j^t$ , if  $w_l = r_j$ . The quantity to synchronize is  $\{w_l\}$ .

**Definition 8** A synchronization between two computations  $k_i \ll k_j$  is defined as a specific computation

$$k_{i,j}^{sync}(S, R, (w, D)),$$

where  $S = \emptyset$ ,  $R = \{(r, n)\} = \{(w_i, n_j \in \mathcal{N})\}$ ,  $(w, D) = (w_i, \bigcup_{\phi \in D_j} n_j(\phi))$ . In other words,  $w_i$  has to be synchronized for the neighborhood  $n_j$  for all entities of  $D_j$ .



**Definition 9** If  $k_i \ll k_j$ ,  $k_j$  is replaced by the list

$$[k_{i,j}^{sync}, k_j]$$

where the synchronization operation has been added.

When data parallelism is applied, the other type of computation which is responsible for additional synchronizations is the reduction. Actually, the reduction is first applied locally on each subset of entities, on each resource. Thus,  $p$  (number of resources) scalar values are obtained. For this reason, to perform the final reduction, a set of synchronizations are needed to get the final reduced scalar. As most parallelism libraries (MPI, OpenMP) already propose a reduction synchronization with their own optimizations, we simply replace the reduction computation by itself annotated by *red*.

**Definition 10** A reduction kernel  $k_j(S_j, R_j, (w_j, D_j))$ , where  $w$  is a scalar, is replaced by  $k_j^{red}(S_j, R_j, (w_j, D_j))$ .

**Definition 11** The concatenation of two ordered lists of respectively  $n$  and  $m$  computations  $l_1 = [k_i]_{0 \leq i \leq n-1}$  and  $l_2 = [k'_i]_{0 \leq i \leq m-1}$  is denoted  $l_1 \cdot l_2$  and is equal to a new ordered list  $l_3 = [k_0, \dots, k_{n-1}, k'_0, \dots, k'_{m-1}]$ .

**Definition 12** From the ordered list of computation  $\Gamma$ , a new synchronized ordered list  $\Gamma_{sync}$  is obtained from the call  $\Gamma_{sync} = F_{sync}(\Gamma, 0)$ , where  $F_{sync}$  is the recursive function defined in Algorithm 1.

Algorithm 1 follows previous definitions to build a new ordered list which includes synchronizations. In this algorithm, lines 7 to 19 apply Definition (6), lines 20 to 29 apply Definition (7), and finally lines 34 and 35 apply Definition (10). Finally, line 37 of the algorithm is the recursive call.

The final step of this parallelization is to run  $\Gamma_{sync}$  on each resource. Thus, for each resource  $0 \leq r \leq p-1$  the multi-stencil program

$$\mathcal{MSP}_r(\mathcal{M}_r, \Phi_r, \mathcal{D}_r, \mathcal{N}, \Delta_r, \mathcal{S}, T, \Gamma_{sync}), \quad (2)$$

is performed.

**Example** Figure 7 gives an example of a  $\mathcal{MSP}$  program. From this example, the following ordered list of computation kernels is extracted:

$$\Gamma = [k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8]$$

From this ordered list of computation kernels  $\Gamma$ , and from the rest of the multi-stencil program, synchronizations can be automatically detected from the call to  $F_{sync}(\Gamma, 0)$  to get the synchronized ordered list of kernels:

$$\Gamma_{sync} = [k_0, k_{0;1}^{sync}, k_1, k_2, k_3, k_{1;4}^{sync}, k_4, k_5, k_6, k_7, k_{7;8}^{sync}, k_8], \quad (3)$$

where

$$k_{0;1}^{sync} = (\emptyset, \{(B, nce)\}, (B, \cup_{\phi \in D_1} nce(\phi))), \quad (4a)$$

$$k_{1;4}^{sync} = (\emptyset, \{(C, nec)\}, (C, \cup_{\phi \in D_4} nec(\phi))), \quad (4b)$$

$$k_{7;8}^{sync} = (\emptyset, \{(I, ncc)\}, (I, \cup_{\phi \in D_8} ncc(\phi))). \quad (4c)$$

**Algorithm 1**  $F_{sync}$  recursive function

---

```

1: procedure  $F_{sync}(\Gamma, j)$ 
2:    $k_j = \Gamma[j]$ 
3:    $list = []$ 
4:   if  $j = |\Gamma|$  then
5:     return  $list$ 
6:   else if  $\exists (r_j, n_j) \in R_j$  such that  $n_j \neq identity$  then
7:     for all  $(r_j, n_j) \in R_j$  such that  $n_j \neq identity$  do
8:        $found = false$ 
9:       for  $0 \leq i < j$  do
10:         $k_i = \Gamma[i]$ 
11:        if  $k_i \prec k_j$  then
12:           $found = true$ 
13:           $S = \emptyset$ 
14:           $R = \{(w_i, n_j)\}$ 
15:           $(w, D) = (w_i, \bigcup_{\phi \in D_j} n_j(\phi))$ 
16:           $list.[k_{i;j}^{sync}(S, R, (w, D))]$ 
17:        end if
18:      end for
19:      if  $!found$  then
20:        for  $j < i \leq n$  do
21:           $k_i = \Gamma[i]$ 
22:          if  $k_i \prec k_j$  then
23:             $S = \emptyset$ 
24:             $R = \{(w_i, n_j)\}$ 
25:             $(w, D) = (w_i, \bigcup_{\phi \in D_j} n_j(\phi))$ 
26:             $list.[k_{i;j}^{sync}(S, R, (w, D))]$ 
27:          end if
28:        end for
29:      end if
30:       $list \cdot [k_j]$ 
31:    end for
32:  else if  $w_j \in S$  then
33:     $list.[k_j^{red}]$ 
34:  else
35:     $list.[k_j]$ 
36:  end if
37:  return  $list \cdot F_{sync}(\Gamma, j + 1)$ 
38: end procedure

```

---

## 5.2 Hybrid parallelism

A task parallelization technique is a technique to transform a program as a dependency graph of different tasks. A dependency graph exhibits parallel tasks, or on the contrary sequential execution of tasks. Such a dependency graph can directly be given to a dynamic scheduler, or can statically be scheduled. In this paper, we consider a computation kernel as a task and we introduce task parallelism by building the dependency graph between kernels of the sequential list  $\Gamma_{sync}$ . Thus, as  $\Gamma_{sync}$  already takes into account data parallelism, we introduce hybrid parallelism.

**Definition 13** For two computations  $k_i$  and  $k_j$ , with  $i < j$ , it is said that  $k_j$  is dependent from  $k_i$  with a read after write dependency, denoted  $k_i \prec_{raw} k_j$ ,

if  $\exists(r_j, n_j) \in R_j$  such that  $w_i = r_j$ . In this case,  $k_i$  has to be computed before  $k_j$ .

**Definition 14** For two computations  $k_i$  and  $k_j$ , with  $i < j$ , it is said that  $k_j$  is dependent from  $k_i$  with a write after write dependency, denoted  $k_i \prec_{waw} k_j$ , if  $w_i = w_j$  and  $D_i \cap D_j \neq \emptyset$ . In this case,  $k_i$  also has to be computed before  $k_j$ .

**Definition 15** For two computations  $k_i$  and  $k_j$ , with  $i < j$ , it is said that  $k_j$  is dependent from  $k_i$  with a write after read dependency, denoted  $k_i \prec_{war} k_j$ , if  $\exists(r_i, n_i) \in R_i$  such that  $w_j = r_i$ . In this case,  $k_i$  also has to be computed before  $k_j$  is started so that values read by  $k_i$  are relevant.

These definitions are known as *data hazards classification*. However, a specific condition on the computation domain, due the multi-stencils specific case, is introduced for the write after write case. One can note that the **independent** keyword of Fig. 6 is useful in this case as the user explicitly indicates that  $D_i \cap D_j = \emptyset$ .

**Definition 16** A directed acyclic graph (DAG)  $G(V, A)$  is a graph where the edges are directed from a source to a destination vertex, and where, by following edges direction, no cycle can be found from a vertex  $u$  to itself. A directed edge is called an arc, and for two vertices  $v, u \in V$  an arc from  $u$  to  $v$  is denoted  $(u, v) \in A$ .

From the ordered list of computations  $\Gamma_{sync}$  and from the MSL description, a directed dependency graph  $\Gamma_{dep}(V, A)$  can be built finding all pairs of computations  $k_i$  and  $k_j$ , with  $i < j$ , such that  $k_i \prec_{raw} k_j$  or  $k_i \prec_{waw} k_j$  or  $k_i \prec_{war} k_j$ .

**Definition 17** For two directed graphs  $G(V, A)$  and  $G'(V', A')$ , the union  $(V, A) \cup (V', A')$  is defined as the union of each set  $(V \cup V', A \cup A')$ .

**Definition 18** From the synchronized ordered list of computation kernels  $\Gamma_{sync}$ , the dependency graph of the computations  $\Gamma_{dep}(V, A)$  is obtained from the call  $F_{dep}(\Gamma_{sync}, 0)$ , where  $F_{dep}$  is the recursive function defined in Algorithm 2.

This constructive function is possible because the input is an ordered list. Actually, if  $k_i \prec k_j$  then  $i < j$ . As a result,  $k_i$  is already in  $V$  when the arc  $(k_i, k_j)$  is built.

One can note that  $\Gamma_{dep}$  only takes into account a single time iteration. A complete dependency graph of the simulation could be built. This is a possible extension of this work.

**Proposition 19** The directed graph  $\Gamma_{dep}$  is an acyclic graph.

As a result of the hybrid parallelization, each resource  $0 \leq r \leq p - 1$  perform a multi-stencil program, defined by

$$\mathcal{MSP}_r(\mathcal{M}_r, \Phi_r, \mathcal{D}_r, \mathcal{N}, \Delta_r, T, \Gamma_{dep}).$$

The set of computations  $\Gamma_{dep}$  is a dependency graph between computation kernels  $k_i$  of  $\Gamma$  and synchronizations of kernels added into  $\Gamma_{sync}$ .  $\Gamma_{dep}$  can be built from the call to

$$F_{dep}(F_{sync}(\Gamma, 0), 0).$$

**Algorithm 2**  $F_{dep}$  recursive function

---

```

1: procedure  $F_{dep}(\Gamma_{sync}, j)$ 
2:    $k_j = \Gamma_{sync}[j]$ 
3:   if  $j = |\Gamma_{sync}|$  then
4:     return  $(\{\}, \{\})$ 
5:   else if  $j < |\Gamma_{sync}|$  then
6:      $G = (\{\}, \{\})$ 
7:     for  $0 \leq i < j$  do
8:        $k_i = \Gamma_{sync}[i]$ 
9:       if  $k_i \prec_{raw} k_j$  or  $k_i \prec_{waw} k_j$  or  $k_i \prec_{war} k_j$  then
10:         $G = G \cup (k_j, \{(k_i, k_j)\})$ 
11:      end if
12:    end for
13:    return  $G \cup F_{dep}(\Gamma_{sync}, j + 1)$ 
14:  end if
15: end procedure

```

---

*Example* Figure 7 gives an example of  $MSP$  program. From  $\Gamma_{sync}$  that has been built in Equation (3), the dependency DAG can be built. For example, as  $k_4$  computes  $F$  and  $k_6$  reads  $F$ ,  $k_4$  and  $k_6$  becomes vertices of  $\Gamma_{dep}$ , and an arc  $(k_4, k_6)$  is added to  $\Gamma_{dep}$ . The overall  $\Gamma_{dep}$  built from the call to  $F_{dep}(\Gamma_{sync}, 0)$  is drawn in Figure 8. By building synchronizations as defined in Definitions (6), (7) and (8), dependencies are respected. For example,  $k_{0;1}^{sync}$  read and write  $B$  which guarantees that  $k_{0;1}^{sync}$  is performed after  $k_0$  and before  $k_1$ .

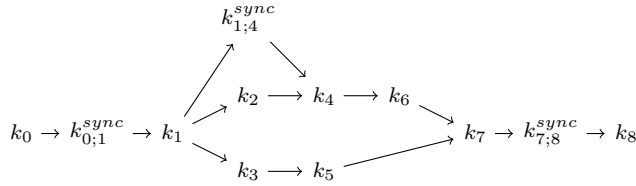


Fig. 8:  $\Gamma_{dep}$  of the example of program of Figure 7

### 5.3 Static scheduling

In this section we detail a static scheduling of  $\Gamma_{dep}$  by using minimal series-parallel directed acyclic graphs. Such a static scheduling may not be the most efficient one, but it offers a simple fork/join task model which makes possible the design of a performance model. Moreover, such a scheduling offers a simple way to propose a fusion optimization.

In 1982, Valdes & Al [31] have defined the class of Minimal Series-Parallel DAGs (MSPD). Such a graph can be decomposed as a serie-parallel tree,

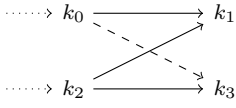


Fig. 9: Over-constraint on the forbidden  $N$  shape.

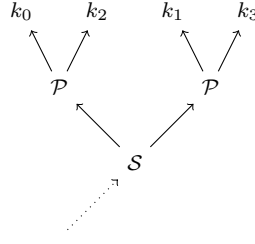


Fig. 10: TSP tree of Fig. 9.

denoted  $TSP$ , where each leaf is a vertex of the MSPD it represents, and whose internal nodes are labeled  $S$  or  $P$  to indicate whether the two sub-trees form a sequence or parallel composition. Such a tree can be considered as a fork-join model and as a static scheduling. An example is given in Fig. 10.

Valdes & Al [31] have identified a forbidden shape, or sub-graph, called  $N$ , such that a DAG without this shape is MSPD.

Thus, as  $\Gamma_{dep}$  is a DAG, by removing N-Shapes it is transformed to a MSPD. The intuition is illustrated in Fig. 9. Considering the figure without the dashed line, the sub graph forms a "N" shape. The fact is that this shape cannot be represented as a composition of sequences or parallel executions. To remove such forbidden N-shapes of  $\Gamma_{dep} = (V, E)$ , we have chosen to apply an over-constraint with the relation  $k_0 \prec k_3$ , such that a complete bipartite graph is created for the sub-dag as illustrated in Figure 9. By adding this arc to the DAG, it is possible to identify its execution as  $sequence(parallel(k_0; k_2); parallel(k_1; k_3))$  represented by the TSP tree of Fig. 10.

After these over-constraints are applied,  $\Gamma_{dep}$  is MSPD. Valdes & Al [31] have proposed a linear algorithm to know if a DAG is MSPD and, if it is, to decompose it to its associated binary decomposition tree. As a result, the binary tree decomposition algorithm of Valdes & Al can be applied on  $\Gamma_{dep}$  to get the  $TSP$  static scheduling of the multi-stencil program.

**Example** From  $\Gamma_{dep}$  illustrated in Fig. 8 the TSP tree represented in Fig. 11 can be computed.

#### 5.4 Fusion optimization

Using MSL, it is possible to ask for data parallelization of the application, or for an hybrid parallelization. Even though the MSL language is not dedicated to produce very optimized independent stencil codes, but to produce the parallel orchestration of computations, building the  $TSP$  tree makes available an easy optimization when the data parallelization technique is the only one used. This optimization consists in proposing a valid merge of some computation kernels inside a single space loop. This is called a fusion. As previously

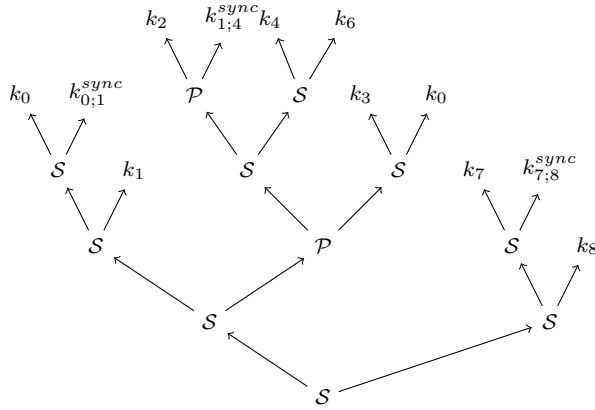


Fig. 11: Serie-Parallel tree decomposition of the example of program of Figure 7

explained in Section 3, MSL restrict the definition of a numerical computation by writing a single quantity at a time which avoids errors in manual fusion or counter-productive fusions for task parallelization. MSF guarantees that proposed fusions are correct and will not cause errors in the final results of the simulation.

Those fusions can be computed from the canonical form of the *TSP* tree decomposition. The canonical form consists in recursively merging successive *S* vertices or successive *P* vertices of *TSP*.

The fusion function  $F_{fus}$  is described in Algorithm 3, where the  $parent(k)$  function returns the parent vertex of  $k$  in the tree, and where  $k_{i;j}^{fus}$  represents the fusion of  $k_i$  and  $k_j$  keeping the sequential order  $i;j$  if  $i$  is computed before  $j$  in *TSP*. Finally,  $type(k)$  returns *comp* if the kernel is a computation kernel, and *sync* or *red* otherwise.

We are not arguing that such a simple fusion algorithm could be as good as complex cache optimization techniques which can be found in stencil DSLs [30] for example. However, this fusion takes place at a different level and can bring performance improvements as illustrated in Section 6. This fusion algorithm relies on the following observations.

First, two successive computation kernels  $k_i$  and  $k_j$  which are under the same parent vertex *S* in *TSP* are, by construction, data dependent. As a result, what is written by the first one is read by the second one. Thus,  $w_i$  the quantity written by  $k_i$  is common to these computations. Thus, if the computation domains verify  $D_i = D_j$ , the fusion of  $k_i$  and  $k_j$  will decrease cache misses.

Second, two successive computation kernels  $k_i$  and  $k_j$  which are under the same parent vertex *P* in *TSP* are not, by construction, data dependent. However, if the computation domains verify  $D_i = D_j$ , and if  $R_i \cap R_j \neq \emptyset$

**Algorithm 3**  $F_{fus}$ 


---

```

1: procedure  $F_{fus}(TSP(V, E))$ 
2:   for  $(k_i, k_j) \in V^2$  do
3:     if parent( $k_i$ )==parent( $k_j$ ) then
4:       if type( $k_i$ )==type( $k_j$ )==comp then
5:         if parent( $k_i$ )== $S$  then
6:           if  $D_i == D_j$  then
7:             propose the fusion  $k_{i;j}^{fus}$ 
8:           else
9:             if  $\exists n : D_i \rightarrow D_j \in (N)$  and  $\bigcup_{\phi \in D_i} n(\phi) = D_j$  then
10:              propose the fusion  $k_{i;j}^{scatter}$ 
11:            end if
12:          end if
13:        else if parent( $k_i$ )== $P$  then
14:          if  $D_i == D_j$  and  $R_i \cap R_j \neq \emptyset$  then
15:            propose the fusion  $k_{i;j}^{fus}$ 
16:          end if
17:        end if
18:      end if
19:    end if
20:  end for
21: end procedure

```

---

cache misses could also be decreased by the fusion  $k_{i;j}^{fus}$ . These two cases are illustrated by Fig. 12 and Fig. 13.

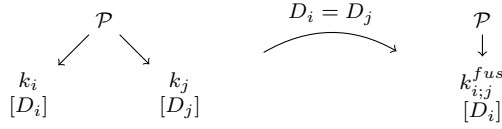


Fig. 12: First fusion case.

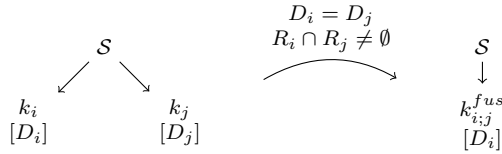


Fig. 13: Second fusion case.

Third, an additional fusion case is possible and more tricky to find. Similarly to the first observation, two successive computation kernels  $k_i$  and  $k_j$  which are under the same parent vertex  $S$  in TSP are data dependent and

what is written by the first one is read by the second one. The construction of the tree also guarantees that synchronizations are not needed between these computations, otherwise a  $k^{sync}$  would have been inserted between them (inherited from  $\Gamma_{sync}$ ). Thus,  $w_i$  the quantity written by  $k_i$  is common to these computations. Considering the following:

- $D_i \neq D_j$ , which means that loop fusion is by default not possible,
- $(r_j, n_j)$  is the pair read by  $k_j$  for which  $r_j = w_i$  and for which  $n_j : D_j \rightarrow D_i^m$

the fusion of  $k_i$  and  $k_j$  is possible if and only if  $\exists n : D_i \rightarrow D_j \in \mathcal{N}$  such that

$$\bigcup_{\phi \in D_i} n(\phi) = D_j$$

This means that even if domains are different, a loop fusion is possible if an adequate neighborhood function can be found. One can note that this particular fusion case is equivalent to a *scatter* optimization, often used when using unstructured meshes. One can also note that the computation  $k_j$  will be written in a different manner if a scatter fusion is performed or not. This particular case is illustrated in Fig. 14.

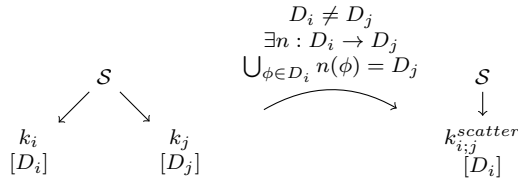


Fig. 14: Third fusion case.

The developer will be notified of fusions in the output of MSC. This is not a problem by using MSF as the fusion is proposed before the developer actually write the numerical code of  $k_j$ .

## 5.5 Overall compilation process

MSC takes a MSL file written using the grammar described in Section 4, as well as the Generic Assembly presented in Fig. 5 as inputs, and generates a specialized component assembly that manages the parallel orchestration of the computations of the simulation. In this final assembly, that could be compared to a pattern or a skeleton of the simulation, the developer still has to fill-in the functions corresponding to the various computation kernels by using the DDS instantiation chosen into the specialized assembly. The overall behavior of the compiler is as follows:



1. it parses the MSL input file and generates  $\Gamma$ , the list of computation kernels,
2. from  $\Gamma$ , it builds  $\Gamma_{sync}$ , the list including synchronizations for data parallelism using Algorithm  $F_{sync}$  introduced in Section 5,
3. from  $\Gamma_{sync}$ , it builds  $\Gamma_{dep}$ , the DAG supporting hybrid parallelism using Algorithm  $F_{dep}$  introduced in Section 5,
4. it then removes the N-Shapes from  $\Gamma_{dep}$  to get a MSPD graph, and generates its serie-parallel binary tree decomposition  $TSP$ ,
5. it performs the fusion of kernels in  $TSP$  if required (data parallelization only),
6. it transforms GA to generate its output specialized component assembly.

The last step of this compilation process is detailed below. It is composed of four steps:

1. it instantiates *DDS* and *Data* components by using components implemented by a third party HPC specialist,
2. it generates the structure of *K* components responsible for each computation kernel of the simulation,
3. it generates a new *Scheduler* component,
4. it replaces the *Computations* component of GA by a generated sub-assembly that matches  $TSP$  by using *Scheduler*, *K* and *Sync* components.

New components have been introduced above and need to be explained. A *K* component is a component into which the developer will write numerical code. It could represent a single computation kernel described by the numerician using MSL, or it could represent the fusion of multiple computation kernels. In any case the name of the generated component will use kernel identifiers used in the MSL description. A *K* kernel is composed of  $m$  use ports that are used to be connected to the  $m$  quantities needed by the computation (*i.e.*, the numerical code). The component also exposes a provide port to be connected to the *Scheduler* component. Interfaces of a *K* component are represented in Fig. 15a.

A *Sync* component is a static component (not generated) composed of a use-multiple port which is used to request synchronizations for all quantities it is linked to (*Data*). The component also exposes a provide port to be connected to the *Scheduler* component. The *Sync* component is represented in Fig. 15b.

Finally, the *Scheduler* component is the component responsible for implementing the  $TSP$  tree computed by MSC. Thus, this component represents the specific parallel orchestration of computations. It exposes as many use ports as there are instances of *K* components to call (*i.e.*, computations and fusions of computations). The component also exposes a provide port to be connected to the *Time* component. Interfaces of a *Scheduler* component are represented in Fig. 15c.

To illustrate how a specialized assembly is generated, the specialized assembly of the example that has been used throughout this paper is represented in Fig. 16.

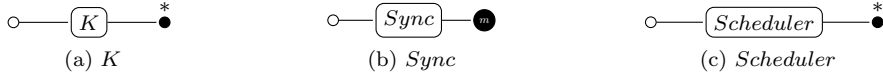


Fig. 15: Specific components used to transform GA to the specialized component assembly of the simulation.

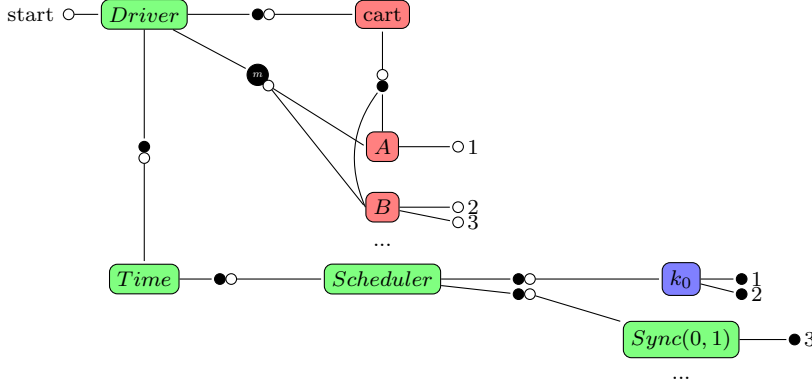


Fig. 16: Sub-part of the specialized assembly generated by MSC from the example of the example of Fig.7 used throughout the paper. For readability some connections are represented by numbers instead of lines. The entire assembly is generated by MSC, however some components are automatically generated by MSC (in green), some are written by HPC specialists (in red) and others by the developer (in blue).

## 5.6 Performance model

In this subsection we introduce two performance models, one for the data parallelization technique, and one for the hybrid data and task parallelization technique, both previously explained.

The performance model for the data parallelization technique is inspired by the Bulk Synchronous Parallel model. We consider that each process handles its own sub-domain that has been distributed in a perfectly balanced way. The performance model describes the computation time as the sum of the sequential time divided by the number of processes, and of the time spent in communications between processes. Thus, for

- $T_{SEQ}$  the sequential reference time,
- $P$  the total number of processes,
- $T_{COM}$  the communications time,

the total computation time is

$$T = \frac{T_{SEQ}}{P} + T_{COM}. \quad (5)$$

Thus, when the number of processes  $P$  increase in data parallelization, the performance model limit is  $T_{COM}$

$$\lim_{P \rightarrow +\infty} T = T_{COM}. \quad (6)$$

As a result, the critical point for performance is when  $T_{COM} \geq \frac{T_{SEQ}}{P}$ , which happens naturally in data parallelization as  $T_{COM}$  will increase with the number of processes, and  $\frac{T_{SEQ}}{P}$  decrease with the number of processes.

This limitation is always true, but can be delayed by different strategies. First, it is possible to overlap communications and computations. Second, it is possible to introduce another kind of parallelization, task parallelization. Thus, for the same total number of processes, only a part of them are used for data parallelization, and the rest are used for task parallelism. As a result,  $\frac{T_{SEQ}}{P}$  will continue to decrease but  $T_{COM}$  will increase later. This second strategy is the one studied in the following hybrid performance model.

For an hybrid (data and task) parallelization technique, and for

- $P_{data}$  the number of processes used for data parallelization,
- $P_{task}$  the number of processes used for task parallelization, such that  $P = P_{data} \times P_{task}$  is the total number of processes used,
- $T_{task}$  the overhead time due to task parallelization technique,
- and  $F_{task}$  the task parallelization degree of the application,

the total computation time is

$$T = \frac{T_{SEQ}}{P_{data} \times F_{task}} + T_{COM} + T_{task} \quad (7)$$

The time overhead due to task parallelization can be represented as the time spent to create a pool of threads and the time spent to synchronize those threads. Thus, for

- $T_{cr}$  the total time to create the pool of threads (may happened more than once),
- $T_{sync}$  the total time spent to synchronize threads,

the overhead is

$$T_{task} = T_{cr} + T_{sync}.$$

The task parallelization degree of the application  $F_{task}$  is the limitation of a task parallelization technique. As explained before, a task parallelization technique is based on the dependency graph of the application. Thus, this dependency graph must expose enough parallelism for the number of available threads. For this performance model we consider that

$$F_{task} = P_{task},$$

however, as it will be illustrated in Section 6  $F_{task}$  is more difficult to establish. Actually, the lower and upper bounds of  $F_{task}$  are constrained by the dependency graph of the application.

As a result when  $P_{data}$  is small a data parallelization technique may be more efficient, while an hybrid parallelization could be interesting at some point to improve performance. The question is: when is it interesting to use hybrid parallelization ? This paper does not propose an intelligent system to answer this question automatically, however, it offers a way to understand how to answer the question. To answer this question let's consider the two parallelization techniques, data only and hybrid. We denote

- $P_{data1}$  the total number of processes entirely used by the data only parallelization,
- $P_{data2}$  the number of processes used for data parallelization in the hybrid parallelization,
- and  $P_{task}$  the number of processes used for task parallelization in the hybrid parallelization,
- such that  $P_{data1} = P_{data2} \times P_{task}$ .

We search the point where the data parallelization is less efficient than the hybrid parallelization. Thus,

$$\frac{T_{SEQ}}{P_{data1}} + T_{COM1} \geq \frac{T_{SEQ}}{P_{data2} \times P_{task}} + T_{COM2} + T_{task}.$$

This happens when

$$T_{COM1} \geq T_{COM2} + T_{task} \quad (8)$$

This performance model will be validated and will help explain results of Section 6.

## 6 Evaluation

This section first presents the implementation details chosen to evaluate MSF in this paper, and the studied use case. Then, the compilation time of MSC is evaluated before analyzing both available parallelization techniques, data and hybrid (data and task). Finally, the impact of kernels fusions is studied.

### 6.1 Implementation details

The main choices to take when implementing a specialized assembly of GA concern the technologies used for data and task parallelizations, *i.e.*, implementation choices of *DDS* and *Scheduler* components.

For the data-parallelization, as already detailed many times throughout the paper, a third party HPC specialist is responsible for implementing *DDS* and *Data* using a chosen library or external language and by following the specified interfaces of these two components. To evaluate MSF, we have played the role of HPC specialists and have implemented these components using SkelGIS, a

C++ embedded DSL [10] that proposes a distributed Cartesian mesh as well as user API to manipulate structures while hiding their underlying distribution.

For task parallelism, we have chosen to use OpenMP [13] to generate the code of the *Scheduler* component. OpenMP targets shared-memory platforms only. Although the version 4 of OpenMP has introduced explicit support for dynamic task scheduling, our implementation only requires version 3 whose fork-join model is well suited for the static scheduling introduced in Section 5. The use of dynamic schedulers, such as provided by libgomp<sup>2</sup>, StarPU [2], or XKaapi [17], to directly execute the DAG  $\Gamma_{dep}$  is left to future work.

As a result, MSC generates a hybrid code which uses both SkelGIS and OpenMP. It also generates the structure of  $K$  components where the developer must provide local sequential implementations of the kernels using SkelGIS API.

## 6.2 Use case description

All evaluations presented in this section are based on a real case study of the shallow-Water Equations as solved in the FullSWOF2D<sup>3</sup> [10,16] code from the MAPMO laboratory, University of Orl eans. In 2013, a full SkelGIS implementation of this use case has been performed by numericians and developers of the MAPMO laboratory [9, 10, 12]. From this implementation we have kept the code of computation kernels to directly use it into  $K$  components. Compared to a full SkelGIS implementation, where synchronizations and fusions are handled manually, MSF automatically compute where synchronizations are needed and how to perform a fusion without errors. To evaluate MSF on this use case we have described the FullSWOF2D simulation by using MSL. FullSWOF2D contains 3 mesh entities, 7 computation domains, 48 data and 98 computations (32 stencil kernels and 66 local kernels). Performances of the obtained implementation are compared to the plain SkelGIS implementation to show that no overheads are introduced by MSF by using L<sup>2</sup>C.

## 6.3 Multi-Stencil Compiler evaluation

Table 1 illustrates the execution time of each step of MSC for the FullSWOF2D example. This has been computed on a laptop with a dual-core Intel Core i5 1.4 GHz, and 8 GB of DDR3. MSC has been implemented in Python 2. While the overall time of 4.6 seconds remains reasonable for a real case study, one can notice that the computation of the *TSP* tree is by far the longest step. As a matter of fact, the complexity of the algorithm for N-shapes removal is  $O(n^3)$ . If this complexity is not a problem at the moment and onto this use case it could become one for just-in-time compilation or more complex simulations. The replacement of the static scheduling by a dynamic scheduling

<sup>2</sup> <https://gcc.gnu.org/projects/gomp/>

<sup>3</sup> <http://www.univ-orleans.fr/mapmo/soft/FullSWOF/>

using dedicated tools (such as OpenMP 4, StarPU etc.) should solve this in the future.

Step	Parser	$\Gamma_{sync}$	$\Gamma_{dep}$	$TSP$
Time (ms)	1	2	4.2	3998.5
%	0.022	0.043	0.09	86.6

Table 1: Execution times of the MSL compiler

#### 6.4 Data parallelism evaluation

In this part, we disable task-parallelism to focus on data-parallelism. Two versions of the code are compared in this section: first a plain SkelGIS implementation of FullSWOF2D, where synchronizations and fusions are handled manually; second, a MSF over SkelGIS version where synchronizations and fusions are automatically handled. SkelGIS has already been evaluated in comparison with a native MPI implementation for the FullSWOF2D example [10]. For this reason, this section uses the plain SkelGIS implementation as the reference version. This enables to evaluate both the choices made by MSC as well as the potential overheads of using L<sup>2</sup>C [5] that is not used in the plain SkelGIS version. The evaluations have been performed on the Curie supercomputer (TGCC, France) described in Table 2. Each evaluation has been performed nine times and the median is presented in results.

	TGCC Curie Thin Nodes
Processor	2×SandyBridge (2.7 GHz)
Cores/node	16
RAM/node	64 GB
RAM/core	4GB
#Nodes	5040
Compiler [-O3]	gcc 4.9.1
MPI	Bullxmpi

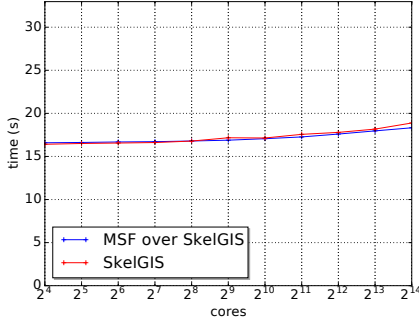
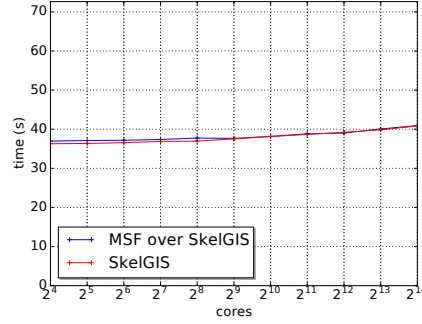
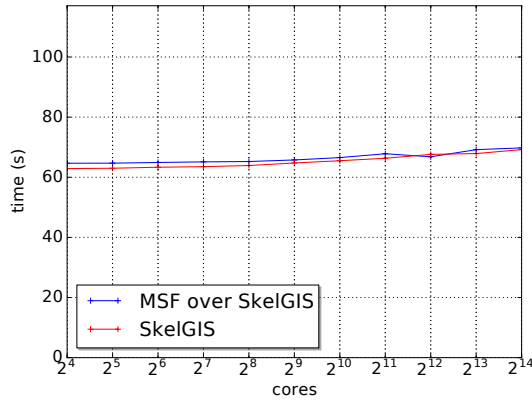
Table 2: Hardware configuration of TGCC Curie Thin nodes.

**Weak scaling** Figures 17, 18 and 19 respectively show weak scaling experiments that we have conducted. Four computation domains are evaluated:  $400 \times 400$  cells by core,  $600 \times 600$  cells by core and  $800 \times 800$  cells by core, from 16 to 16,384 cores, as summarized in Table 3.

From these results, one can notice, first, that performances of MSF are very close to the reference version using plain SkelGIS. This is a very good

Domain size per core	Number of iterations
$400 \times 400$	200
$600 \times 600$	200
$800 \times 800$	200

Table 3: Weak scaling experiments of Fig. 17, Fig. 18 and Fig. 19.

Fig. 17: weak-scaling with  $400 \times 400$  domain per core and 200 time iterations.Fig. 18: weak-scaling with  $600 \times 600$  domain per core and 200 time iterations.Fig. 19: weak-scaling with  $800 \times 800$  domain per core and 200 time iterations.

result which shows first that MSC performs good synchronizations and fusions, and second that overheads introduced by  $L^2C$  are limited thanks to a good component granularity in the Generic Assembly.

However, it seems that a slightly drop of performance happens when the domain size per core increases. This performance decrease is really small though, with a maximum difference between the two versions of 2.83% in Fig. 19.

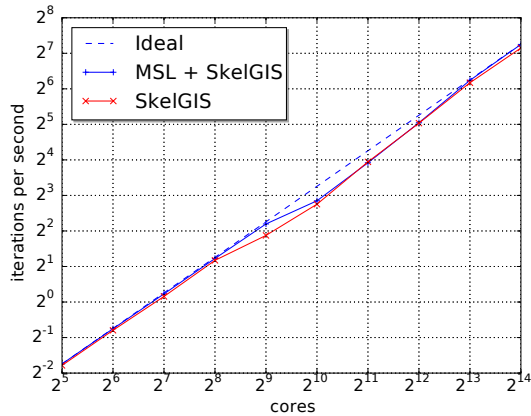


Fig. 20: Strong scaling on a  $10k \times 10k$  domain and 1000 time iterations.

The only noticeable difference between the two versions are due to  $L^2C$  which load dynamic libraries at runtime. Because of this particularity, components of  $L^2C$  are compiled with the `-fpic` compilation flag<sup>4</sup> while the SkelGIS version does not. This flag can have slight positive or negative effects on code performance depending on the situation and might be responsible for the observed difference.

**Strong scaling** Figure 20 shows the number of iteration per second for a  $10k \times 10k$  global domain size from 16 to 16,384 cores. The total number of time iterations for this benchmark is 1000. In addition to the reference SkelGIS version, the ideal strong scaling is also plotted in the figure.

First, one can notice that the strong scaling evaluated for the MSF version is close to the ideal speedup up to 16,384 cores, which is a very good result. Moreover, no overheads are introduced by MSF which shows that automatic synchronizations and automatic fusions enable the same level of performance than the one manually written into the plain SkelGIS version. Finally, no overheads are introduced by components of  $L^2C$ . A small behavior difference can be noticed with  $2^9 = 512$  cores, however this variation is no longer observed with 1024 cores.

## 6.5 Hybrid parallelism evaluation

In this section, we add task parallelism to evaluate the hybrid parallelization offered by MSF. The MSF implementation evaluated in this paper relies on SkelGIS and OpenMP.

<sup>4</sup>  $L^2C$  has been recently extended with the possibility of static linking.



The series-parallel tree decomposition  $TSP$  of this simulation, extracted by MSC, is composed of 17 nodes labeled as sequence  $\mathcal{S}$  and 18 nodes labeled as parallel  $\mathcal{P}$ .

We define the *level of parallelism* as the number of parallel tasks inside one fork of the fork/join model. The fork/join model obtained for FullSWOF2D is composed of 18 fork phases (corresponding to  $\mathcal{P}$  nodes of  $TSP$ ). Table 4 represents the number of time (denoted frequency) a given level of parallelism is obtained inside fork phases.

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

Table 4: Parallelism level and the number of times this parallelism level appears into fork phases.

One can notice that the level of task parallelism extracted from the Shallow water equations is limited by two sequential parts in the application (level 1). Moreover, a level of 16 parallel tasks is reached two times, and five times for the fourth level. This means that if two cores are dedicated to task parallelism, the two sequential parts of the code will not take advantage of these two cores, and that no part of the code would benefit from more than 16 cores. The task parallelism, as proposed in this paper (*i.e.*, where each kernel is a task) is therefore insufficient to take advantage of a single node of modern clusters that typically supports more than 16 cores.

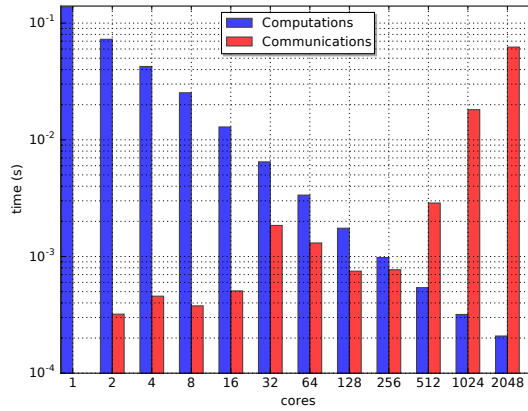


Fig. 21: Computation vs communication times for a single time iteration using the data parallelization technique.

On the other hand, Figure 21 illustrates limitations of data parallelization technique alone. This figure displays the execution time (with a logarithmic scale) of FullSWOF2D while increasing the number of cores for a fix domain size of  $500 \times 500$  with a total of 200 time iterations (*i.e.*, this is a strong scaling). One can note that times are really small. Actually the time represented in Fig. 21 is the time spent into a single time iteration. The speedup of this same benchmark is represented in blue in Figure 22. One can note that the scaling is not as good as the one presented in Figure 20. The main difference between these two benchmarks is the domain size. In the benchmark of Figure 20 the domain size is  $10k \times 10k$  which means that using  $2^8 = 256$  cores, for example, each core has to compute only a  $625 \times 625$  sub-domain. On the other hand, using  $2^8$  cores in Figure 22 each core has to compute a  $31 \times 31$  sub-domain. Figure 21 shows why the speedup is not as good as the one with a bigger domain size.

Actually, in this figure, while the computation time (in blue) decreases linearly with the number of core used, the communication behavior (in red) is much more erratic. Between 2 and 16 cores, communications are performed inside a single node thus the time is small and nearly constant. There is a small oscillation that might be explained by the partitioning differences. SkelGIS performs a two dimensional partitioning strategy. For this reason a smaller number of bytes are communicated using 2 cores than using 4, and using 8 cores than using 16 cores. Starting from 32 cores, each node is fully used and more than one node is used. From this point thus the communication time is typically modeled as  $L + S/B$  where  $L$  is the latency,  $S$  the data size and  $B$  the bandwidth. This explains the decrease of communication time from 32 to 128 cores where the data sizes communicated by each process decreases. The increases observed after 128 cores might be due to the fact that with the increased number of processes the fat-tree becomes deeper and the latencies increase.

All in all, when the number of core increases, the computation/communication ratio becomes poorer and poorer. As a result, the data parallelism alone fails to provide enough parallelism to leverage the whole machine and other sources of parallelism have to be found. As expected, in Figure 22 the speedup bends down from 256 to 2048 cores. The same problem would happened in previous experiment of Figure 20, however as the domain size is larger, the phenomena appears with more cores.

As task parallelism fails to scale from 16 cores, and as data parallelism also fails to scale when the communication cost overpass the execution time, an hybrid parallelization strategy is proposed by MSF and is evaluated below.

In addition to the blue curve, Figure 22 shows speedups for the same example ( $500 \times 500$  domain with 200 iterations) but using an hybrid parallelization. Figure 22 shows a comparison with 2, 4, 8 and 16 cores per MPI process for task parallelization.

For example, the purple curve shows the parallelization which uses for each data parallelization process (*i.e.*, MPI process) 8 additional cores for task parallelization. As a result, for example, when using 2 machines of the

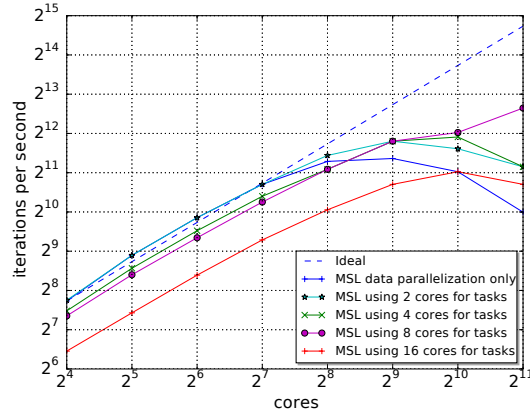


Fig. 22: Strong scaling comparisons between data parallelization and hybrid parallelization. A *close* OpenMP clause is used to bind threads onto cores.

TGCC cluster, with a total of 32 cores, 4 cores are used for SkelGIS MPI processes, for data parallelization, and for each one 8 cores are used for task parallelization ( $4 \times 8 = 32$ ). This respects  $P = P_{data} \times P_{task}$  as presented in Section 5.6. As a result, and as explained in Section 5.6, quantities that are responsible for communications are less divided into sub-domains. Therefore, the effect observed with the blue curve is delayed to a higher number of cores.

From 2 to 8 cores, the improvement of the strong scaling is clear. However, reaching 16 cores, an important initial overhead appears and in addition to this, the curve bends down rapidly instead of improving the one with 8 cores for task parallelization. Two different phenomena happen in this case.

First, thin nodes of the TGCC Curie are built with two NUMA socket each of 8 cores. As a result, when increasing the number of OpenMP cores for task parallelization from 8 to 16 cores, an overhead is introduced by exchanges of data between memories of the two NUMA sockets. This phenomena is illustrated in Figure 23. In this figure, a different binding strategy is used. A binding strategy is the way the scheduler binds threads onto available cores. The strategy used in Figure 23 is called *spread* (instead of *close* in Figure 22). This strategy binds threads on cores in order to spread as much as possible onto resources, which means that the two NUMA sockets are used whatever the number of cores used for tasks. As a result, and as shown in the figure, using 2, 4 and 8 cores an initial overhead is introduced as the one observed in Figure 22. This shows that the initial overhead with 16 cores is due to NUMA effects.

The second phenomena that happens in Figure 22 using 16 cores is due to the level of parallelism introduced by the task parallelization technique. Actually, as illustrated in Table 4, only two forks of *TSP* can take advantage of 16 cores among a total of 18 forks. This phenomena has been mentioned in

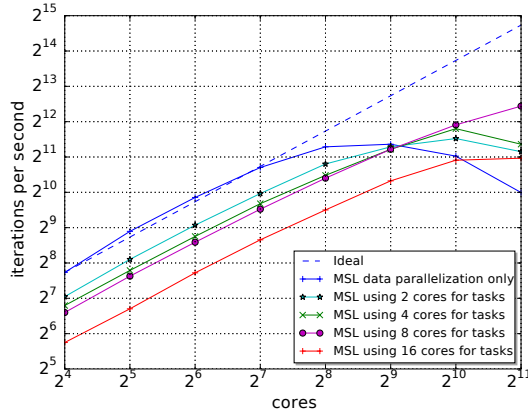


Fig. 23: Strong scaling comparisons between data parallelization and hybrid parallelization. A *spread* OpenMP clause is used to bind threads onto cores.

Section 5.6 by the variable  $F_{task}$  and the fact that it is not always true that  $F_{task} = P_{task}$ . This explains why using 16 cores is less efficient than using 8 cores, even when the two NUMA sockets are always used as in Figure 23.

Finally, to validate the performance model introduced in Section 5.6, and to understand when the hybrid parallelization becomes more interesting than the data parallelization, Figure 24 represents  $T_{COM1}$  and  $T_{COM2} + T_{task}$  of Equation (8), for the best case, *i.e.*, when 8 cores are used in Figure 22. Figure 24 and Table 5 presents results of these measurements. Results perfectly matches Figure 22 for 8 cores per MPI process. As a result, the hybrid parallelization is better for 512 cores or more in this case.

	$T_{COM1}$	$T_{COM2}$	$T_{task}$	Equation (8)
16 cores ( $2 \times 8$ )	0.0005	0.00032	0.013	False
32 cores ( $4 \times 8$ )	0.0018	0.00045	0.0062	False
64 cores ( $8 \times 8$ )	0.0013	0.00038	0.0034	False
128 cores ( $16 \times 8$ )	0.00075	0.0005	0.0023	False
256 cores ( $32 \times 8$ )	0.00077	0.0018	0.001	False
512 cores ( $64 \times 8$ )	0.0029	0.0013	0.00052	True
1024 cores ( $128 \times 8$ )	0.018	0.00075	0.00029	True
2048 cores ( $256 \times 8$ )	0.0623	0.00077	0.00016	True

Table 5: Execution times (seconds) of  $T_{COM1}$ ,  $T_{COM2}$  and  $T_{task}$  for 8 cores for task parallelization. Verification of the Equation (8).

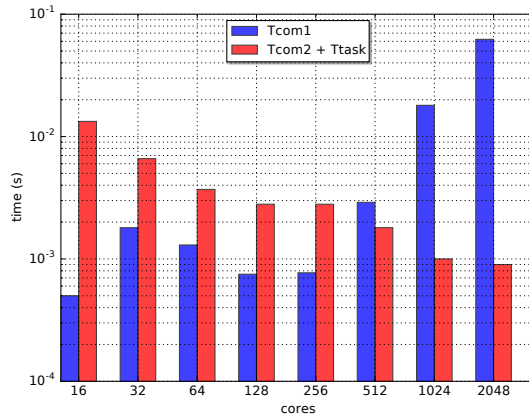


Fig. 24: Execution times (seconds) for a single time iteration of  $T_{COM1}$  and  $T_{COM2} + T_{task}$  for 8 cores for task parallelization. Verification of the Equation (8).

## 6.6 Fusion evaluation

In this section we propose an evaluation of the fusion optimization. From the *TSP* tree computed by MSC it may be possible, according to some specific conditions, to merge the domain loops of some kernels, thus optimizing the use of cache memories. This kind of optimization is called a fusion and three fusion optimizations have been introduced in Section 5.4. Among them, the two first ones (Figures 12 and 13 on page 22) have been automatically detected by MSF in this case study.

Figure 25 shows the number of iterations per second as a function of the number of cores with and without fusions. This benchmark is performed on FullSWOF2D onto a  $500 \times 500$  domain size with 200 time iterations, and by using data parallelism alone (without tasks). As explained in Section 5.4, the MSF loop fusion happens at a high level. Most of the time such fusions are done naturally by a computer scientist. However, an automatic detection of such fusions avoids errors, particularly for a parallel execution. In addition to this, more advanced fusion cases, such as a scatter, are more difficult to deduce. In FullSWOF2D a total of 62 fusions are proposed by MSF over a total of 98 computation kernels. Figure 25 shows that the performance is clearly improved (around 40%) by these fusions.

However, fusion optimizations are not always relevant. To illustrate this, we are using the same benchmark of FullSWOF2D onto a  $500 \times 500$  domain size with 200 time iterations, however we compare data parallelism and hybrid parallelism both with and without fusion.

Blue curves of Figure 26 represent results for data parallelism with and without fusion. One can note that the best performance, as expected, is reached

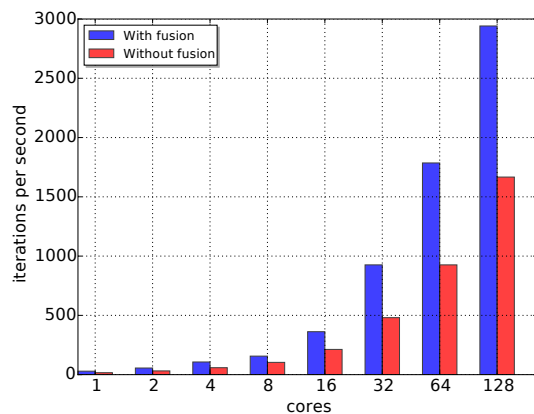


Fig. 25: Strong scaling on a 500x500 domain size with 200 time iterations, with and without fusions proposed by MSF.

by the version using fusions. Red curves represent results by using 2 cores per MPI process dedicated to tasks, with and without fusion again. One can note that the best performance is also reached by the version using fusion.

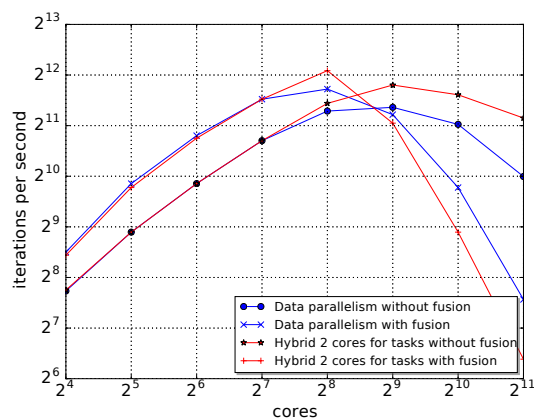


Fig. 26: Strong scaling on a 500x500 domain size with 200 time iterations. Blue curves represent strong scaling for data parallelism with and without fusion. Red curves represent strong scaling by using 2 cores per MPI process dedicated to tasks, with and without fusion.

However, to deeper analyze this results, we propose a second evaluation presented in Figure 27. The blue curves are exactly the same one than in Figure 26. The red curves, on the other hand, represent results by using 8

cores per MPI process dedicated to tasks, with and without fusion. Interesting results appears in this figure as the hybrid version using fusions is less efficient than the one without fusions. As already explained, this result is due to the fact that fusions reduce the number of tasks from 98 to 36 resulting in a non optimized use of eight cores for task parallelism. By using only 2 cores per MPI process (in Figure 26) the 36 computation kernels were enough to feed the two cores, while it is not for eight.

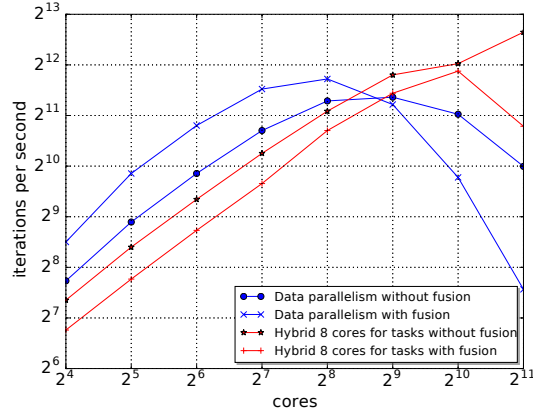


Fig. 27: Strong scaling on a 500x500 domain size with 200 time iterations. Blue curves represent strong scaling for data parallelism with and without fusion, thus are exactly the same than blue curves of Fig. 26. Red curves represent strong scaling by using 8 cores per MPI process dedicated to tasks, with and without fusion.

In conclusion, if fusion optimization incurs a too large reduction of the number of tasks to feed dedicated cores, the problem observed for 16 cores in Figure 22 happens earlier which reduces performance. For this reason, MSF performs fusions only when data parallelization is used alone. This choice could be more intelligent but this is the subject of future work.

## 7 Related work

Many domain specific languages have been proposed for the optimization of single stencil computations. Each one has its own optimization specificities and targets a specific numerical method or a specific kind of mesh. For example, Pochoir [30] focuses on cache optimization techniques for stencils applied onto Cartesian meshes. On the other hand, PATUS [8] proposes to add a parallelization strategy grammar to its stencil language to perform an auto-tuning parallelization. ExaSlang [28] is specific to multigrid numerical methods. Thus,

these stencil compilers target a different scope than the Multi-Stencil Framework presented in this paper, which actually orchestrates a parallel execution of multiple stencil codes together. Hence, an interesting future work would be to combine these stencil compilers with MSF to build very optimized stencil kernels  $K$ .

Some solutions, closer to MSF, have also been proposed to automatically orchestrate multiple stencils and computations in a parallel manner. Among them is Halide [26] that proposes an optimization and parallelization of a pipeline of stencils. However, Halide is limited to Cartesian meshes and is specific to images. Liszt [14], OP2 [18] and Nabla [7] all offer solutions for the automatic parallel orchestration of stencils applied onto any kind of mesh, from Cartesian to unstructured meshes. The needed mesh can be built from a set of available symbols in the grammar of each language. Thus, these languages generalize the definition of a mesh, as it is proposed into the MSL formalism of Section 3. However, neither Liszt, OP2 nor Nabla handle hybrid parallelism as it is proposed by MSF.

MSF offers the MSL Domain Specific Language to the numerician to describe its sequential set of computations. This description, is close to a dataflow representation. However, MSL differs from general purpose dataflow languages or framework for two main reasons. First, MSL is specific to numerical simulations and proposes a mesh abstraction adapted to numerical simulations. Thus, compared to general purpose dataflow runtimes such as Legion [4], HPX [19], PFunc [20], MSL is closer to the semantic of the domain (mesh, stencils etc.) and easier to use for non-specialists. Second, MSL is very light and only descriptive. Numerical codes are left to another language and another user (the developer in Figure 2 on page 5). Furthermore, such dataflow runtimes could actually be used by MSF as back-ends, instead of using SkelGIS or OpenMP.

This flexibility proposed by MSF is due to software engineering capacities introduced by proposing a meta-model and by using a component programming model. Actually, MSF is designed to increase separation of concerns and code-reuse compared to existing solutions. Separation of concerns is illustrated in Figure 2 and all along the paper. The numerician is only responsible for the description of the simulation by using MSL. A HPC specialist can propose new (or updated) components for handling the distributed data structure and quantities of the simulation. MSF generates from these pieces of information the parallel orchestration of computations. Finally, the developer of numerical codes fills computation kernels by using the chosen distributed data structure. In Liszt, OP2 and Nabla, for example, there is no such separation of concerns between the numerician and the developer. Moreover, it is not possible to easily integrate a new distributed data structure in these solutions as a monolithic code is generated. Finally, thanks to components, MSF improves code-reuse. Kernel components as well as any component (except the scheduler component which is specific to the simulation) can be reused from one simulation to another.

To conclude and as far as we know, no component-based framework has been proposed for stencils orchestration.



## 8 Conclusion

In this paper, we have presented MSF, a multi-stencil framework. MSF is built upon a meta-formalism of a multi-stencil program that we have presented in Section 3. From this meta-formalism, we have designed, first, the generic component assembly of a multi-stencil program, and second, the domain specific language MSL that enables the description of a specific application by a numerician. From these entries, MSC, the MSF compiler, automatically generates a parallel component assembly. This assembly represents the parallel orchestration of computations, independently of implementation choices. Two parallelization strategies are supported: data parallelization and hybrid (data and task) parallelization.

By combining a meta-model and component-based programming, MSF has the particularity to enhance separation of concerns, as well as code-reuse and composition of existing solutions (*e.g.*, SkelGIS and OpenMP in this paper). MSF has been evaluated on the real case simulation FullSWOF2D. Results show that the MSF runtime does not induce unwanted overheads for its data parallelization technique on both strong and weak scalings. Results also show that the hybrid parallelization supported by MSF can increase performance when the data parallelism exposed by the simulation is not enough, which is a new contribution compared to existing solutions. Finally, we have evaluated the fusion optimization and shown that it often increases performance but it is not always a good choice, particularly when performing hybrid parallelism.

Many perspectives of future work raise from this paper. First, even if using one distributed data structure or another is possible and facilitated by MSF, the HPC specialist still has to understand component programming models and the meta-model of the framework to add new implementation of *DDS* and *Data* components. We think it could be interesting to also facilitate the work of the HPC specialist by proposing a semi-automated framework to add new implementations of these components to MSF.

Second, it has been shown in the performance model and evaluation results that the choice between data parallelism and hybrid parallelism is not trivial. It depends on the time spent in computations, the number of cores and nodes used for the execution, as well as the network behavior (Figure 21 and 22). MSF, thanks to its meta-formalism and to the use of component models, easily handles the activation of one parallelism technique or another. Thus, MSF can be considered as a performance leverage for multi-stencil programs. However, it would be interesting to investigate how an algorithm could choose the best parallelization technique for a given execution of a multi-stencil simulation.

The same choice problem is raised by the fusion optimization. For the same reasons an algorithm could choose the best solution for a given execution. To design such an algorithm different solutions should be studied, such as following an approximated behavior model, or using calibration runs to then use deep learning algorithms, etc.

Finally, by using component models doors are opened to reconfiguration of applications [22] which means that the application could adapt its behavior and structure to external events.

**Acknowledgements** This work has partially been supported by the PIA ELCI project of the French FSN. This work was granted access to the HPC resources of TGCC under the allocations t2015067470, x2016067617 and AP010610191 made by GENCI.

## References

1. B. A. Allan et al. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.
2. C. Augonnet, S. Thibault, R. Namyst, and P-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
3. F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *annals of telecommunications*, 64(1-2):5–24, 2009.
4. Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *International Conf. on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.
5. Julien Bigot, Zhengxiong Hou, Christian Prez, and Vincent Pichon. A low level component model easing performance portability of hpc applications. *Computing*, 96(12):1115–1130, 2014.
6. Julien Bigot and Christian Pérez. Increasing Reuse in Component Models through Genericity. Research Report RR-6941, 2009.
7. Jean-Sylvain Camier. Improving performance portability and exascale software productivity with the  $\&$ numbla; numerical programming language. In *Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC '15*, pages 126–131, Edinburgh, Scotland, UK, 2015. University of Edinburgh.
8. M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, May 2011.
9. Stéphane Cordier, Hélène Coullon, Olivier Delestre, Christian Laguerre, Minh Hoang Le, Daniel Pierre, and Georges Sadaka. Fullswof paral: Comparison of two parallelization strategies (mpi and skelgis) on a software designed for hydrology applications. In *ESAIM: Proceedings*, volume 43, pages 59–79. EDP Sciences, 2013.
10. H. Coullon and S. Limet. The SIPSim implicit parallelism model and the SkelGIS library. *Concurrency and Computation: Practice and Experience*, 2015.
11. Hélène Coullon and Sébastien Limet. Algorithmic skeleton library for scientific simulations: Skelgis. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 429–436, 2013.
12. Hélène Coullon, Sébastien Limet, and Hoang Le Minh. Parallelization of Shallow-Water Equations with the Algorithmic Skeleton Library SkelGIS. In Elsevier, editor, *ICCS*, volume 18 of *Procedia Computer Science*, pages 591–600, Barcelone, Spain, June 2013. Elsevier.
13. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.
14. Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

15. ETP4HPC. ETP4HPC Strategic Research Agenda Achieving HPC leadership in Europe. Technical report, ETP4HPC, 2013.
16. S. Ferrari and F. Saleri. A new two-dimensional shallow water model including pressure effects and slow varying bottom topography. *M2AN Math. Model. Numer. Anal.*, 38(2):211–234, 2004.
17. T. Gautier, J.V.F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13*, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.
18. M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H.J. Kelly. Performance Analysis of the OP2 Framework on Many-core Architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4):9–15, March 2011.
19. Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14*, pages 6:1–6:11, New York, NY, USA, 2014. ACM.
20. Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. Pfunc: Modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, pages 43:1–43:11, New York, NY, USA, 2009. ACM.
21. Cédric Lachat, François Pellegrini, and Cécile Dobrzynski. PaMPA: Parallel Mesh Partitioning and Adaptation. In *21st International Conference on Domain Decomposition Methods (DD21)*, Rennes, France, June 2012. INRIA Rennes-Bretagne-Atlantique.
22. Vincent Lanore and Christian Pérez. A reconfigurable component model for hpc. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE '15*, pages 1–10, New York, NY, USA, 2015. ACM.
23. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.
24. Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.
25. François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1996*, pages 493–498, London, UK, UK, 1996. Springer-Verlag.
26. J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 519–530, New York, NY, USA, 2013. ACM.
27. J. Richard, V. Lanore, and C. Pérez. Towards application variability handling with component models: 3d-fft use case study. In *Proc. of The 8th Workshop on UnConventional High Performance Computing (UCHPC)*, Vienna, Austria, August 2015. To appear.
28. C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14*, pages 42–51, Piscataway, NJ, USA, 2014. IEEE Press.
29. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
30. Y. Tang, R.A. Chowdhury, B.C. Kuszmaul, C-K Luk, and C.E. Leiserson. The pochoir stencil compiler. In Lance Fortnow and Salil P. Vadhan, editors, *SPAA*, pages 117–128. ACM, 2011.
31. Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, STOC '79*, pages 1–12, New York, NY, USA, 1979. ACM.