The Multi-Stencil Language: orchestrating stencils with a mesh-agnostic DSL

Hélène Coullon, Julien Bigot, Christian Pérez

To cite this version:

Hélène Coullon, Julien Bigot, Christian Pérez. The Multi-Stencil Language: orchestrating stencils with a mesh-agnostic DSL. [Research Report] RR-8962, Inria - Research Centre Grenoble – Rhône-Alpes. 2016, pp.30. hal-01380607v2

HAL Id: hal-01380607

https://hal.inria.fr/hal-01380607v2

Submitted on 19 Oct 2016

The Multi-Stencil Language: orchestrating stencils with a mesh-agnostic DSL

Hélène Coullon, Julien Bigot, Christian Perez

RESEARCH REPORT
N° 8962
October 2016
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8962--FR+ENG

The Multi-Stencil Language: orchestrating stencils with a mesh-agnostic DSL

Hélène Coullon[*][†], Julien Bigot[‡][§], Christian Perez

Project-Team Avalon

Research Report n° 8962 — October 2016 — 30 pages

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 get programming models which gives 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 not design a new DSL each time a new domain or a new problem has to be solved. This phenomenon happens for stencil-based numerical simulations, for which a large number of languages has been proposed without code reuse between them. The Multi-Stencil Language (MSL) presented in this paper is a language common to any kind of mesh used into a stencil-based numerical simulation. It is said that MSL is mesh-agnostic. Actually, from the description of a numerical simulation, MSL produces an empty parallel pattern, or skeleton, of the simulation which will be filled using other existing parallel languages and libraries. Thus, MSL, by finding a common language for different kinds of stencil-based simulation, facilitates code reuse. MSL is evaluated on a real case simulation which solves shallow-water equations. It is shown that MSL does not introduce overheads on data parallelism up to 16.384 cores, and that the hybrid parallelism (data and task) introduced improves performance of the simulation.

Key-words: Domain Specific Language (DSL), stencil, numerical simulation, data parallelism, task parallelism, scheduling, MPI, OpenMP

[*] Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP

[†] {helene.coullon}{christian.perez}@inria.fr

[‡] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Univ. Paris-Saclay

[§] julien.bigot@cea.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Le langage MSL: orchestration de stencils au travers d'un DSL maillage-agnostique

Résumé : Alors que la puissance de calcul des architectures hautes performances augmente, leur hétérogénéité et leur complexité augmente également ce qui rend de plus en plus difficile leur utilisation. Pour cette raison, l'un des sujet de recherche majeur pour prétendre atteindre des machines exascale est de proposer des modèles de programmation permettant l'accès au calcul hautes performances au plus grand nombre et non pas seulement à une poignée de spécialistes. L'une des solutions permettant de faciliter la programmation parallèle aux scientifiques tout en conservant les performances est l'utilisation de langages de domaine spécifiques (DSL). Cependant, l'un des problèmes lié aux DSL est leur conception qui n'est pas envisageable pour chaque nouveau domaine visé car trop chronophage et complexe. Ce problème apparaît notamment dans le domaine de la simulation numérique basé sur les calculs de type stencil. Dans ce domaine un très large ensemble de langages ont été proposés chacun avec ses spécificités et sans aucune ré-utilisation de fonctionnalités et de codes de conception de l'un vers l'autre. Le langage MSL (Multi-Stencil Language) présenté dans ce rapport de recherche est un langage qui extrait le fondement commun des différents types de simulation multi-stencil en proposant une abstraction du type de maillage utilisé. MSL produit un squelette (ou patron) de l'application décrite en entrée et ré-utilise des langages hautes performances existants pour obtenir l'application finale. MSL est évalué sur un cas réel de simulation numérique par lequel il est montré que MSL n'introduit pas de surcoûts par rapport à une utilisation classique des langages sous-jacents. Le passage à l'échelle est évalué jusqu'à 16.384 cœurs de calcul. Il est également montré que la version hybride de code introduite par MSL améliore les performances en suivant un modèle de performances également décrit.

Mots-clés : Langage de domaine spćifique, stencil, simulation numŕique, parallĺisme de donnés, parallĺisme de tâches, ordonnancement, 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 world's top supercomputer Tianhe-2 [1] is composed of multi-cores processors and accelerators, and is able to reach a theoretical peak performance around thirty peta floating point operations par seconds. However, to be able to use such a machine, multiple programming models, such as MPI (Message Passing Interface), OpenMP, CUDA etc., and multiple optimization techniques, such as cache or vectorization optimizations, have to be combined. Moreover, current architectures evolution lets think that heterogeneity and complexity in HPC will continue to grow in future.

One of the big challenges to solve to be able to use exascale computers is to propose programming models which gives access to high performance computing (HPC) to many scientists and not only to a few HPC specialists [10]. Actually, applications which run on supercomputers and which need such a computation power are physics, weather or genomic applications, which are not implemented by HPC specialists most of the time.

One possible runtime execution model for HPC is to propose dynamic scheduling of task graphs combined to message passing models [1, 12, 22] (to be able to use more than one machine). Those models increase HPC code portability and reach an interesting performance onto heterogeneous architectures, which is interesting to reach exascale programming. At a higher abstraction level, general purpose parallel languages, such as OpenMP [8] and OpenCL [18] follow the direction of task graph scheduling proposed by those execution models. However, for non-experts end-users, general purpose languages still are difficult to use and to tune for a given application onto a given architecture. The current easiest (with the higher abstraction level), but still efficient, programming model for end-users is Domain Specific Language (DSL). Such a language is specific to the end-user domain and proposes a grammar which is easy to understand. The DSL compiler is able, because of the specific knowledge onto the domain targetted, to automatically apply parallelization and specific optimizations to produce a high performance back-end code. Thus, a DSL is able to split end-user concerns from HPC concerns which is relevant to reach exascale programming models.

However, many DSLs have been proposed for many domains yet, and, as far as we know, only a few of them have been able to reuse work already done by another language [19]. In other words, software engineering properties have to be integrated into DSL conception, such that a new DSL can be seen as a composition of parallelization, optimizations or even languages semantics already proposed by others.

For example, to numerically solve a set of partial differential equations (PDEs), iterative methods are frequently used to approximate the exact solution through a discretized phenomena. A very well known and usual way to discretize PDEs is to transform them to explicit numerical schemes, also often called *stencils*. Many DSLs have been proposed for stencil computations [3, 4, 9, 16, 20], as it will be detailed in Section 2. Many of them use same kind of parallelization, data structures or optimizations, however each one has been built from scratch to deal with another additionnal specific case. We present the Multi-Stencil Language (MSL) DSL, also for stencil-based numerical simulations. MSL is a language with a light grammar to describe a numerical simulation without implementation details. From the description, the compiler has enough information to extract and build an empty parallel pattern of the simulation, which can be filled, in a second step, by implementation concerns. The parallel pattern generated by the language is able to use different existing languages and libraries as it is independent from implementation choices. Moreover, the parallelization performed by the language is large enough to be compatible with many architectures and back-end languages. Contributions presented in

[1] urlwww.top500.org

this paper are : the computational model of a multi-stencil program and its parallelization formalism; the MSL grammar and its compiler; a back-end implementation and its performance evaluation onto a real case numerical simulation up to 16.384 cores.

Section 2 introduces the related work on DSLs for stencils. Section 3 formally explains the targetted domain and its computational model. From this model can be extracted the grammar of MSL in Section 4. Section 5 shows how parallelism can be extracted from this light grammar of MSL. Sections 6 and 7 detail choices that have been done in this paper to evaluate MSL, and Section 8 shows performance results of the language. Finally, Section 9 concludes and proposes perspectives on this work.

2 Related work

Many domain specific languages have been proposed for stencil computations. Each one has its own specificities and answers to a specific stencil case or to a specific additionnal optimization. For example, Pochoir [20] works on cache optimization techniques for stencils applied onto Cartesian meshes. On the other hand, PATUS [4] proposes to add a parallelization strategy grammar to its stencil language to perform an auto-tuning parallelization strategy. Moreover, Halide [16] proposes an optimization and parallelization of a pipeline of stencil codes, while ExaSlang [17] is specific to multigrid numerical methods etc.

The reason why each of those languages are implemented from scratch to answer its own particularities and its own optimizations is because each language is built and thought as a single block, which makes impossible or difficult code reuse from one language to another. In other words domain specific languages (for stencils or other domains) suffer from a lack of software engineering properties, which could increase the productivity to build a new language, by code-reuse, and also the maintainability of languages, with more separation of concerns. As far as we know, a single work proposes a framework to create DSL, which afterwards will be easier to compose and combine together [19], which is interesting for inter-domains applications. However, the point argued by this paper, and by the Multi-Stencil Language (MSL), is that DSL conception must be studied to maximize its usefulness for different types of applications, while keeping a good performance. This maximization is directly linked to the abstraction level proposed to the end-user, and also to separation of concerns and code reuse improvements.

MSL is itself a domain specific language for stencil-based simulations. It offers a way to give a mesh-agnostic description of a simulation, which could be common to different cases of simulations, and thus facilitates reuse of existing underlying data structures, optimizations or parallelization techniques. In other words, MSL extracts where parallelization and optimizations are needed into the simulation, by producing an empty parallel pattern, but not how to do it, which is left to existing languages (SkelGIS and OpenMP in this paper), and which is an implementation concern.

Liszt [9] and Nabla [3], both offer languages for stencils applied onto any kind of mesh, from Cartesian to unstructured meshes. The mesh which is needed into the simulation can be built from a set of available symbols in the grammar of each language. Thus, those languages generalize the definition of a mesh, as it is proposed into the MSL formalism. However, two main differences can be noticed.

First, the formalism is more flexible in MSL. For example, a computation can be applied onto a subset of the space domain which is not possible with Liszt and Nabla. This functionnality is important in numerical simulations yet. For example, many computations performed in the simulation studied in this paper have to be applied onto a subpart of the overall space domain.

Second, in Liszt and Nabla the description of a simulation is not splitted from its implementation concerns. In other words, the topology of a mesh as well as numerical codes are given at the same time than the description of the different computations to apply into the simulation, while MSL splits those two different phases. Thus, MSL improves separation of concerns. Actually, a numerician could describes the different computations to perform into the simulation, while another one, later, could focuss on the implementation and the numerical code. This also facilitates reuse of other languages.

Finally, the parallelization techniques proposed in this paper, take place at paradigm level, with data parallelism and hybrid (data plus task) parallelism. Thus the parallel pattern of the simulation does not need details onto parallel architectures (distributed or shared memories, with or without accelerators). This makes possible a large panel of backend architectures and languages. In other words MSL improves portability.

3 Computational model of Multi-Stencil Programs

Some approaches to numerically solve partial differential equations (PDEs) are based on direct iterative methods (*e.g.* finite difference, finite volume or finite element methods). They approximate the solution through a discretized process where the continuous time and space domains are discretized and numerical computations are iteratively (time discretization) applied onto a mesh (space discretization). While the computations can have various forms, we focus on three categories. *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 values mapped on the mesh to a single scalar value.

This section gives a complete formal description of what we call a *stencil program* and its computations. The proposed language MSL (Multi-Stencil Language) does not depend on the type of space discretization used. Thus, some details given in the following sections, are useful to understand the domain but do not have to be kept in mind to understand the Multi-Stencil Language.

3.1 Time, mesh and data

Ω 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 Ω of a set of PDEs and is defined as follows.

Definition 1 *A mesh is a connected undirected graph $\mathcal{M} = (V, E)$, where $V \subset \Omega$ is the 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 Ω.*

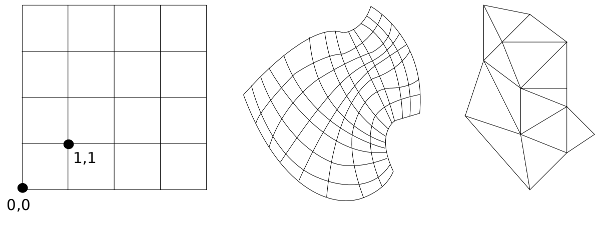


Figure 1: From left to right, Cartesian, curvilinear and unstructured meshes.

Definition 2 *The dimension of a mesh $\mathcal{M} = (V, E)$ applied onto $\Omega = \mathbb{R}^n$ is denoted $dim(\mathcal{M}) = n$.*

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 1. One can notice that more than one type of mesh is also possible inside a single simulation. For example, an hybrid mesh can be defined as an unstructured mesh composed itself of a Cartesian mesh inside each of its vertices. However in this paper, single mesh simulations are addressed.

Definitions (mesh)

- An *entity* ϕ 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 and the mesh of a group is denoted $mesh(\mathcal{G})$, *i.e.* $\mathcal{G} \in \mathcal{P}(V \cup E) \Leftrightarrow mesh(\mathcal{G}) = (V, E)$
- The *set of entities groups* used in a simulation is denoted Φ.

For example, in a 2D Cartesian mesh, a group of entities where each entity is made of four vertices and four edges could form the cells. Another group of entities are the vertices defined as all singletons formed of a single vertex of V. Both groups, then, would be part of Φ. This example is illustrated in Figure 2b.

Definition 3 (Definitions (time)) *The finite sequence $T : (t_n)_{n \in [\![0, T_{max}]\!]}$ represents the discretization of the continuous time domain $\mathcal{T} = \mathbb{R}$. To each discrete time-step $n \in [\![0, T_{max}]\!]$, it associates a time value $t_n \in \mathcal{T}$.*

The time discretization can be as simple as a constant time-step with a fixed number of steps. It can, however, also be defined by recursion with both the time-step and the number of steps depending on the data variables (see definitions below).

Definitions (quantity)

- $\mathbb{V} = \Delta \cup \mathcal{S}$ is the set of *variables* or *quantities*.
- Δ are the *mesh variables* that to each entity of a group and time-step associates a value $\delta : \mathcal{G} \times T \mapsto \mathcal{V}_\delta$ where V_δ 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* that to each time-step associates a value $s : T \mapsto \mathcal{V}_\delta$ where $\mathcal{V}_\delta$ is a value type.
- Amongst the scalar variables is one specific variable $conv \in \mathcal{S}$, the convergence criteria whose value is 0 except at the last step where it is 1. Thus, $\forall t \in [\![0, T_{max-1}]\!]$, $conv(t) = 1$, and $conv(T_{max}) = 1$.

This section has presented the general formalism of meshes, their entities, groups of entities, time discretization, and finally quantities (mapped on mesh and scalars). While the details about mesh and their topologies were useful to specify the following concepts, the Multi-Stencil Language presented in this paper is mesh-agnostic. These details will therefore not be needed in the remaining of the paper.

3.2 Computations

Definitions

- A computation domain D is a subpart of a mesh entities group, $D \subseteq \mathcal{G} \in \Phi$.

- The set of computation domains of a numerical simulation is denoted $\mathcal{D}$.
- $\mathcal{N}$ are the 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 4 *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,*
- *$(w, D) \in \Delta \times \mathcal{D}$ is the single data written by the kernel:*
  - *w is the single quantity (variable) modified by the computation kernel,*
  - *D is the computation domain on which w is computed, $D \subseteq entity(w)$,*
- *$R \in \Delta \times \mathcal{N}$ is the set of tuples (r, n) representing the data read where*
  - *r is a quantity read by the kernel to compute w,*
  - *$n : entity(w) \rightarrow entity(r)^m$ is a neighborhood function that indicates whose values of r are read to compute w.*
- *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 which is computed by a kernel.*

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 (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 kernel computation $k(S, R, (w, D), comp)$ 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 kernel computation $k(S, R, (w, D), comp)$ is a *local kernel* if $\forall(r, n) \in R$, $n = identity$.
- A kernel computation $k(S, R, (w, D), comp)$ is a *reduction kernel* if w is a scalar.

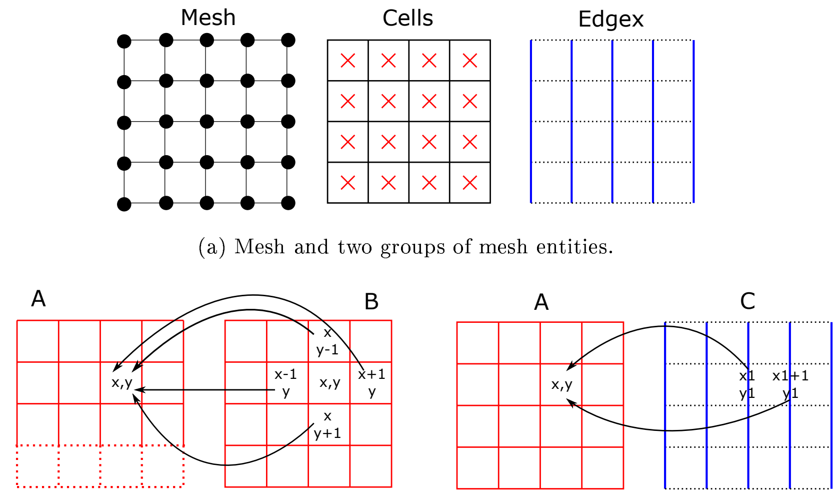
Since we only consider explicit numerical schemes in this paper, a kernel can not write the same quantity it reads, except locally, *i.e.* if $\exists(w, n) \in R \Rightarrow n = identity$

A reduction is typically used to compute the convergence criteria of the time loop of the simulation.

Definition 5 *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 6 *A multi-stencil program is defined by the octuplet*

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



(a) Mesh and two groups of mesh entities.

(b) A is computed with a 4-neighborhood stencil applied on B. A is computed onto a computation domain which does not include all entities of the group.

(c) A is computed with a 2-neighborhood stencil applied on C. A is computed onto a computation domain includes all entities of the group.

Figure 2: (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 entities of the mesh.

Example For example, in Figure 2b, assuming that the computation domain (full lines) is denoted $dc1$ and the stencil shape is $n1$, the stencil kernel can be defined as:

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

$$comp : A(x, y) = B(x + 1, y) + B(x - 1, y) + B(x, y + 1) + B(x, y - 1).$$

On the other hand, in the example of Figure 2c, assuming the computation domain is $dc2$ and the stencil shape is $n2$, the stencil kernel is defined as:

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

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

A stencil program has been formally defined in this section. This formalism is used in the next Section to define two parallelization techniques of a multi-stencil program.

One can note that all definitions given in this section are independent from the topology of the mesh. This property will be kept in the rest of this paper to propose the mesh-agnostic MSL language.

4 The Multi-Stencil Language

From the formalism detailed in the previous section, the Multi-Stencil Language and its grammar can already be given. This grammar is sufficient to automatically extract parallelism from the program as will be explained in the next section.

```
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 | scalar
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
```

Figure 3: Grammar of the Multi-Stencil Language.

The grammar of the Multi-Stencil Language is given in Figure 3 and an example is provided in Figure 4. A Multi-Stencil program is composed of eight parts.

1. The `mesh` keyword (Fig.3, l.1) introduces an identifier for $\mathcal{M}$, the single mesh of the simulation. For example `cart` in Fig.4, l.1. Since the language is independent of the mesh topology, this is not used by the compiler but is syntactic sugar for the user.

2. The `mesh entities` keyword (Fig.3, l.2) introduces identifiers for the groups of entities $\mathcal{G} \in \Phi$. For example `cell` or `edgex` in Fig.4, l.2. Again, this is syntactic sugar only as the compiler does not rely on the mesh topology.

3. The `computation domains` keyword (Fig.3, l.3) introduces identifiers for the computation domains $D \in \mathcal{D}$. For example `d1` and `d2` in Fig.4, l.4-5. For reference, each domain is associated to a group of entities (Fig.3, l.12) such as `cell` for `d1` in Fig.4, l.4.

4. The `independent` keyword (Fig.3, l.4) offers a way to declare that computation domains do not overlap, such as `d1` and `d2` in Fig.4, l.7. This is used by the compiler to compute dependencies between computations.

```
 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    nce 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] = k0(mu,tau,E)
24    H[d2] = k6(F)
25    I[d1] = k7(G,H)
26    J[d1] = k8(mu,I[ncc])
```

Figure 4: Example of program using the Multi-Stencil Language.

5. The `stencil shapes` keyword (Fig.3, l.5) introduces identifiers for $n \in \mathcal{N}$, the stencil shapes. For each shape, the source and destination group of entities (Fig.3, l.16) are specified. For example `nec` in Fig.4, l.11 associates `cell` entities to each `edgex` entity.

6. The `mesh quantities` keyword (Fig.3, l.6) introduces identifiers for $\delta \in \Delta$, the quantities with the group of entities they are mapped on (Fig.3, l.16). For example the quantities `C` and `H` are mapped on `edgex` entities.

7. The `scalars` keyword (Fig.3, l.7) introduces identifiers for $s \in \mathcal{S}$, the scalars. For example `mu` and `tau` in Fig.4, l.15.

8. Finally, the last part (Fig.3, l.8) introduces the computations loops. Each loop is made of two parts:

   - the `time` keyword (Fig.3, l.22) introduces *conv*, the convergence criteria either as a number of iterations or as $0, 1$ valued scalar (Fig.3, l.24). For example, 500 iterations are specified in Fig.4, l.16,
   - the `computations` keyword (Fig.3, l.23) introduces identifiers for each computation $k \in \Gamma = (S, R, (w, D), comp)$. Each computation (Fig.3, l.26) specifies:
     - the quantity w written and its domain D, for example in Fig.4, l.22, kernel `k4` computes the quantity `F` on domain `d1`,

- the read scalars S and mesh quantities with their associated stencil shape R, for example in Fig.4, l.16, `k4` reads `C` with the shape `nce` and `D` with the default *identity* shape, it uses no scalar.

One can notice that in the example of Figure 4, there are no (reduction) kernel associated to the scalars `mu` and `tau`. In this case, those scalars are in fact constants. One can also notice that the computation to execute for each kernel is not specified. This is indeed not handled by MSL, which is mesh-agnostic, and which only generates a skeleton of the application (a parallel pattern of the application). Actually, a numerical computation is naturally mesh-dependent and is left to other languages. The numerical computation is specified elsewhere by referencing the identifier chosen here.

This section has introduced the grammar of MSL. The next section will show how parallelization can be extracted from this simple language and how an empty parallel skeleton (or pattern) of the application can be generated.

5 Parallelization of Multi-Stencil Programs

As previously explained, in a computation $k(S, R, (w, D), comp)$, $comp$ is not handled by MSL. As a result, in the rest of this paper, and to simplify notations, we denote the same computation $k(S, R, (w, D))$.

5.1 Data parallelism

In a data parallelization technique, the idea is to split quantities on which the program is computed into balanced sub-parts, one for each available resource. The same sequential program can afterwards be applied on each sub-part simultaneously, with some additioinal synchronizations between resources to update the data not computed locally, and thus to guarantee a correct result.

More formally, the data parallelization of a multi-stencil program

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

consists in, first, a partitioning of the mesh $\mathcal{M}$ in p balanced sub-meshes (for p resources) $\mathcal{M} = \{\mathcal{M}_0, \ldots, \mathcal{M}_{p-1}\}$. This step can be performed by an external graph partitionner [6,14,15] and is not adressed by this paper.

As entities and quantities are mapped onto the mesh, the set of groups of mesh entities and the set of quantities Δ are partitionned the same way than the mesh: $\Phi = \{\Phi_0, \ldots, \Phi_{p-1}\}$, $\Delta = \{\Delta_0, \ldots, \Delta_{p-1}\}$.

The second step of the parallelization is to identify in Γ the needed synchronizations between resources to update data, and thus to build a new ordered list of computations Γ_{sync}.

Definition 7 *For n the number of computations in Γ, and for i, j such that $i < j < n$, a synchronization is needed between k_i and k_j, denoted $k_i \prec\!\!\prec k_j$, if $\exists (r_j, n_j) \in R_j$ such that $w_i = r_j$ and $n_j \neq identity$ (k_j is a stencil computation). The quantity to synchronize is $\{w_i\}$.*

Actually, a synchronization is needed by the quantity read by a stencil computation (not local), if this quantity has been modified before, which means that it 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 which 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 8 *For n the number of computations in Γ and $j < n$, if $\exists (r_j, n_j) \in R_j$ such that $n_j \neq identity$ and for all $i < j$, $k_i \not\prec\!\!\prec k_j$, a synchronization is needed between k_l and k_j, where $j < l < n$, denoted $k_l \prec\!\!\prec k_j$, if $w_l = r_j$. The quantity to synchronize is $\{w_l\}$.*

Definition 9 *A synchronization between two computations $k_i \prec\!\!\prec k_j$ 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 10 *If $k_i \prec\!\!\prec k_j$, k_j is replaced by the list*

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

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 its own optimizations, we simply choose to replace the reduction computation by itself anotated by *red*.

Definition 11 *A reduction kernel $k_j(S_j, R_j, (w_j, D_j))$, where w is a scalar, is replaced by $k_j^{red}(S_j, R_j, (w_j, D_j))$.*

One can notice that both types of synchronizations are performed by all resources.

Definition 12 *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 13 *From the ordered list of computation Γ, a new synchronized ordered list Γ_{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 (7), lines 20 to 29 apply Definition (8), and finally lines 34 and 35 apply Definition (11). Finally, line 39 of the algorithm is the recursive call.

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

$$\mathcal{MSP}_r(\mathcal{M}_r, \Phi_r, \mathcal{D}_r, \mathcal{N}, \Delta_r, \mathcal{S}, T, \Gamma_{sync}), \qquad (1)$$

is performed.

Algorithm 1 F_{sync} recursive function

```
procedure $F_{sync}(\Gamma, j)$
    $k_j = \Gamma[j]$
    $list = []$
    if $j = |\Gamma|$ then
        return $list$
    else if $\exists (r_j, n_j) \in R_j$ such that $n_j \neq identity$ then
        for all $(r_j, n_j) \in R_j$ such that $n_j \neq identity$ do
            found = false
            for $0 \leq i < j$ do
                $k_i = \Gamma[i]$
                if $k_i \lll k_j$ then
                    found = true
                    $S = \emptyset$
                    $R = \{(w_i, n_j)\}$
                    $(w, D) = (w_i, \bigcup_{\phi \in D_j} n_j(\phi)))$
                    $list.[k_{i;j}^{sync}(S, R, (w, D))]$
                end if
            end for
            if !found then
                for $j < i \leq n$ do
                    $k_i = \Gamma[i]$
                    if $k_i \lll k_j$ then
                        $S = \emptyset$
                        $R = \{(w_i, n_j)\}$
                        $(w, D) = (w_i, \bigcup_{\phi \in D_j} n_j(\phi)))$
                        $list.[k_{i;j}^{sync}(S, R, (w, D))]$
                    end if
                end for
            end if
            $list \cdot [k_j]$
        end for
    else if $w_j \in \mathcal{S}$ then
        $list.[k_j^{red}]$
    else
        $list.[k_j]$
    end if
    return $list \cdot F_{sync}(\Gamma, j + 1)$
end procedure
```

Example Figure 4 gives an example of a $\mathcal{MSP}$ program. From this example, the following ordered list of computation kernels can be extracted:

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

From this ordered list of computation kernels Γ, 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:

$$\Gamma_{sync} = [k_0, k_{0;1}^{sync}, k_1, k_2, k_3, k_{1;4}^{sync}, k_4, k_0, k_6, k_7, k_{7;8}^{sync}, k_8], \tag{2}$$

where

$$k_{0;1}^{sync} = (\emptyset, \{(B, nce)\}, (B, \cup_{\phi \in D_1} nce(\phi))), \tag{3a}$$
$$k_{1;4}^{sync} = (\emptyset, \{(C, nec)\}, (C, \cup_{\phi \in D_4} nec(\phi))), \tag{3b}$$
$$k_{7;8}^{sync} = (\emptyset, \{(I, ncc)\}, (I, \cup_{\phi \in D_8} ncc(\phi))). \tag{3c}$$

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 introduce task parallelism by building the dependency graph between kernels of the sequential list Γ_{sync}. Thus, as Γ_{sync} takes into account data parallelism, we introduce hybrid parallelism.

Definition 14 *For two computations k_i and k_j, with $i < j$, it is said that k_j is dependant 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 15 *For two computations k_i and k_j, with $i < j$, it is said that k_j is dependant 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 16 *For two computations k_i and k_j, with $i < j$, it is said that k_j is dependant 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.*

Those definitions are known as *data hazards classification.* However, a specific condition on the computation domain, due to the specific domain of multi-stencils, is introduced for the write after write case.

Definition 17 *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 $(\widehat{u, v}) \in A$.*

From an ordered list of computations Γ_{sync}, 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 18 *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 19 *From the synchronized ordered list of computation kernels Γ_{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.*

Algorithm 2 F_{dep} recursive function

```
procedure $F_{dep}(\Gamma_{sync}, j)$
    $k_j = \Gamma_{sync}[j]$
    if $j = |\Gamma_{sync}|$ then
        return $(\{\}, \{\})$
    else if $j < |\Gamma_{sync}|$ then
        $G = (\{\}, \{\})$
        for $0 \leq i < j$ do
            $k_i = \Gamma_{sync}[i]$
            if $k_i \prec_{raw} k_j$ or $k_i \prec_{waw} k_j$ or $k_i \prec_{war} k_j$ then
                $G = G \cup (k_j, \{(\widehat{k_i, k_j})\})$
            end if
        end for
        return $G \cup F_{dep}(\Gamma_{sync}, j + 1)$
    end if
end procedure
```

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 $(\widehat{k_i, k_j})$ is built.

One can notice that Γ_{dep} is the dependency graph of the computations of a multi-stencil program, but it 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 20 *The directed graph Γ_{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 Γ_{dep} is a dependency graph between computation kernels k_i of Γ and synchronizations of kernels added into Γ_{sync}. Γ_{dep} can be built from the call to

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

Example Figure 4 gives an example of $\mathcal{MSP}$ program. From Γ_{sync} that has been built in Equation (2), the dependency DAG can be built. For example, as k_0 computes B and k_1 reads B, k_0 and k_1 becomes vertices of Γ_{dep}, and an arc $(\widehat{k_0, k_1})$ is added to Γ_{dep}. The overall Γ_{dep} built from the call to $F_{dep}(\Gamma_{sync}, 0)$ is drawn in Figure 5.

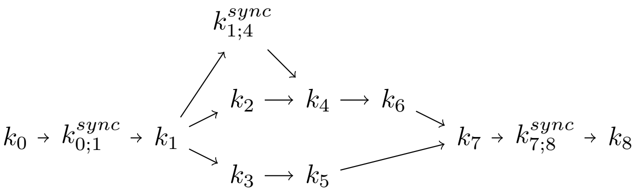


Figure 5: Γ_{dep} of the example of program of Figure 4

6 Static Scheduling and performance model

In this section we detail a static scheduling of Γ_{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 make possible the design of a performance model. Moreover, such a scheduling offers a simple way to propose a fusion optimization.

6.1 Series-Parallel graph

In 1982, Valdes & Al [21] have defined the class of Minimal Series-Parallel DAGs (MSPD). Such a graph can be decomposed as a serie-parallel tree, denoted TSP, where each leaf is a vertex of the MSPD DAG it represents, and whose internal nodes are labelled S or P to indicate respectively the series or parallel composition of sub-trees. Such a tree can be considered as a fork-join model and as a static scheduling.

Valdes & Al [21] have identified a forbidden shape, or subgraph, called N, such that the following property is verified :

Theorem 21 *The transitive reduction of a DAG G is MSPD if and only if it does not contain N as an induced subgraph.*

To remove the forbidden N-shapes from the transitive reduction 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, and can be translated to a series-parallel decomposition, as illustrated in Figure **??**.

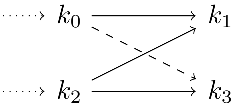


Figure 6: Over-constraint on the forbidden N shape.

After these over-constraints are applied, Γ_{dep} is MSPD. Valdes & Al [21] 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 Γ_{dep} to get the TSP static scheduling of the multi-stencil program.

Example The Serie-Parallel tree decomposition of the example given in Figure 4, which is built from the dependency graph of Figure 5 is given in Figure 7.

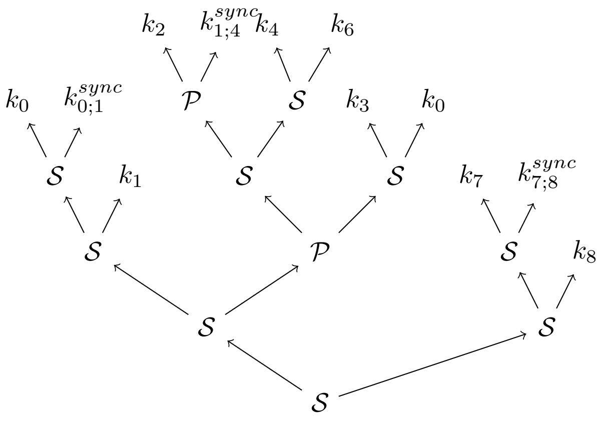


Figure 7: Serie-Parallel tree decomposition of the example of program of Figure 4

6.2 Performance model

In this subsection are introduced two performance model, one for the data parallelization technique, and one for the hybrid data and task parallelization technique, both previously explained.

The performance model of a data parallelization technique is inspired from the Bulk Synchronous Parallel model. Actually data parallelization technique consider that each process has its own part of the domain. Thus the performance model reveals that the computation time is the sum of the reference 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

$$T = \frac{T_{SEQ}}{P} + T_{COM}. \tag{4}$$

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

$$\lim_{P \to +\infty} T = T_{COM}. \tag{5}$$

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 perform communications through the network while computations are performed simultaneously. 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

$$T = \frac{T_{SEQ}}{P_{data} \times F_{task}} + T_{COM} + T_{task} \tag{6}$$

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, the upper bound of F_{task} is constrained by the dependency graph of the application and by the time spent in each task.

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 asked here is when is it interesting to use hybrid parallelization.

To answer this question lets 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

$$T_{COM1} \geq T_{COM2} + T_{task} \tag{7}$$

This performance model will be validated and will help to explain results the Section 8.

6.3 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 stencil codes, but to produces the parallel pattern of the application, building the TSP tree make 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. As a result, the user can use this valid fusion of kernels or not when implementing those.

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 *parent*(k) 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* otherwise.

Algorithm 3 F_{fus}

```
procedure $F_{fus}(TSP(V, E))$
    for $(k_i, k_j) \in V^2$ do
        if parent($k_i$)==parent($k_j$) then
            if $type(k_i) = type(k_j) = comp$ then
                if parent($k_i$)==$S$ then
                    if $D_i == D_j$ then
                        propose the fusion $k_{i;j}^{fus}$
                    end if
                else if parent($k_i$)==$P$ then
                    if $D_i == D_j$ and $R_i \cap R_j \neq \emptyset$ then
                        propose the fusion $k_{i;j}^{fus}$
                    end if
                end if
            end if
        end if
    end for
end procedure
```

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 for example [20]. However, this fusion takes place at a different level and can bring performance improvments as it will be illustrated in Section 8. This fusion algorithm relies on very simple statements:

- Two successive computation kernels k_i and k_j which are under the same parent vertex S in TSP are, by construction, data dependant. As a result, what is written by the first one is read by the second one. Thus, at least one data is common to those computations (the one written by k_i). Thus, if the computation domains verify $D_i = D_j$, the fusion of k_i and k_j will decrease cache misses.
- Two successive computation kernels k_i and k_j which are under the same parent vertex P in TSP are not, by construction, data dependant. However, if the computation domains verify $D_i = D_j$, and if $R_i \cap R_j \neq \emptyset$ cache misses could also be decreased by the fusion $k_{i;j}^{fus}$.

7 Implementation

The MSL compiler takes a file written in the grammar described in Section 4 as input and generates an application skeleton. In this skeleton, the user still has to fill the functions corresponding to the various computation kernels. The overall behavior of the compiler is as follow:

1. it parses the input file and generates Γ, the list of computation kernels,
2. from Γ, it builds Γ_{sync}, the list including synchronizations for data parallelism using Algorithm F_{sync} introduced in Section 5,
3. from Γ_{sync}, it builds Γ_{dep}, the DAG supporting hybrid parallelism using Algorithm F_{dep} introduced in Section 5,
4. it then removes the N-Shapes from Γ_{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,
6. finally it dumps an application structure matching TSP.

The abstract definition of MSL enables to use a very wide range of languages for the dump. In this paper, we use a compiler that targets the Low Level Components [2] (L^2C), a component model for C++ supporting high-performance applications. Similar to classes in object-oriented models, components specify the services they provide, but in addition they also specify the services they require. This enables to build applications by assembling components in a second phase through the matching of requirements of some components with the services provided by others. In the case of MSL, this assembly of components offers a way to clearly separate in distinct components the features that are implemented:

- by reusing the same code more than once, such as for the time iteration logic,
- by generating some code that depends on TSP and thus from the input file, such as for the task parallelism logic,
- or by letting the user fill-in an empty skeleton component, such as for the computation kernels.

By enforcing the use of well defined interfaces in the generated code, L^2C makes easier the substitution of an implementation by another, for example to reconsider an implementation choice. It also simplifies the addition of new features to MSL or the coupling of MSL with another domain-specific language, for example.

The two main other choices in the code generation concern the technologies used for data and task parallelizations. For the data-parallelization, we rely on SkelGIS, a C++ embedded DSL [5] which offers distributed data structures (meshes) for numerical simulations, and programming interfaces to easily write codes using them. SkelGIS is implemented over MPI [13] and can therefore be used on distributed memory architectures such as clusters. Two kind of meshes are offered by SkelGIS: **a)** a distributed two dimensional Cartesian mesh [6], and **b)** a distributed graph of Cartesian meshes (hybrid mesh) [7]. The use of different back-ends for data-parallelism, such as for example the distributed unstructured mesh proposed by PaMPA [14], will be the subject of future work.

For the task parallelism, we use OpenMP [8]. OpenMP targets shared-memory platforms only. However, as the level of parallelism introduced by task parallelism technique is low compared to data parallelism, shared-memory, or intra-node parallelism is a good architecture choice for task parallelism.While the version 4 of OpenMP has introduced explicit support for tasks, our implementation only requires version 3 whose fork-join model is well suited for the static scheduling introduced in Section 6. The use of dynamic schedulers such as provided by libgomp[2], StarPU [1], or XKaapi [12] to directly execute the DAG Γ_{dep} will be the subject of some future work.

As a result, the MSL compiler generates an hybrid code which uses both SkelGIS and OpenMP. It also generates empty components where the user must provide local sequential implementations of the kernels using well defined interfaces of the component to iterate over the local data subdomain. These implementations can of course be written by hand, but ideally one should also be able to generate them using a stencil compiler such as Pochoir or PATUS thus enabling the combinations of approaches used by those compilers. The following section will evaluate this compiler as well as the code it generates.

8 Evaluation

This section evaluates the MSL compiler and the code it generates. We first evaluate the generated code in the case where only data parallelization and fusion are applied and compare this to a full SkelGIS program. Then, we analyze a case taking advantage of the full the hybrid parallelization combining data and task parallelism and validate the performance model proposed in Section 6.1.

All evaluations presented in this section are based on a real case study of the shallow-Water Equations as solved in the FullSWOF2D[3] [5,11] code from the MAPMO laboratory, University of Orléans. We have developed a MSL version of FullSWOF2D that contains 3 mesh entities, 7 computation domains, 48 data and 98 computations (32 stencil kernels and 66 local kernels).

8.1 Compiler

Table 1 illustrates the execution time of each step of the MSL compiler 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. While the overall time of 4.6 seconds remains reasonable, 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 $O(n^3)$. The replacement of the static scheduling by a direct scheduling of the DAG by dedicated tools should solve this in the future.

Step	Parser	Γ_{sync}	Γ_{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

8.2 Data parallelism

In this evaluation, we disable task-parallelism (fix its degree to 1) to focus on data-parallelism. In the implementation evaluated in this paper, the data parallelism is handled using SkelGIS that has already been evaluated in comparison with a native MPI implementation for the

[2] `https://gcc.gnu.org/projects/gomp/`

[3] `http://www.univ-orleans.fr/mapmo/soft/FullSWOF/`

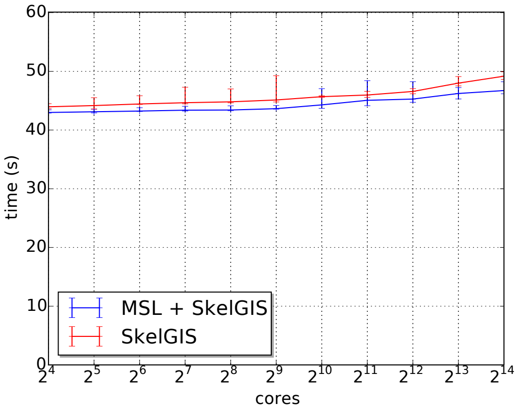


Figure 8: weak-scaling with 400 × 400 domain per core.

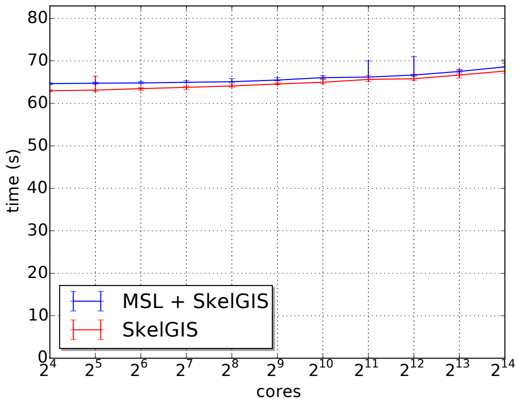


Figure 9: weak-scaling with 800 × 800 domain per core.

FullSWOF2D example [5]. In addition to SkelGIS, MSL automatically inserts the required MPI synchronizations and identifies possible fusion of kernels to reduce cache misses.

Cluster	**TGCC Curie Thin Nodes**
Processor	2×SandyBridge (2.7 GHz)
Cores/node	16
RAM/node	64 GB
RAM/core	4GB
Compiler [-O3]	gcc 4.9.1
MPI	Bullxmpi
Network	fat-tree Infiniband

Table 2: Hardware configuration of TGCC Curie Thin nodes.

This section compares the performance of the code produced by MSL for FullSWOF2D with a plain SkelGIS program where synchronizations and fusion choices have been done by the developer without any automatic support. This enables to evaluate both the choices made by the compiler as well as the potential overheads of using L^2C [2] 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 ten times and the median is presented in results.

Weak scaling Figures 8 and 9 show a weak scaling for a 400 × 400 points by core and a 800 × 800 points by core from 16 cores to 16.384 cores. Minimum and maximum values are also shown as error bars in figures.

One can notice that the MSL code produces a slightly better execution time on the 400 × 400 domain and a small overhead on the 800 × 800 domain. This might be explained by slightly different compiler optimization flags since L^2C requires compilation as dynamic libraries with the *-fpic* compilation flag while the SkelGIS version does not. This flag can have slight positive or negative effect on code performance depending on the situation and might be responsible for the

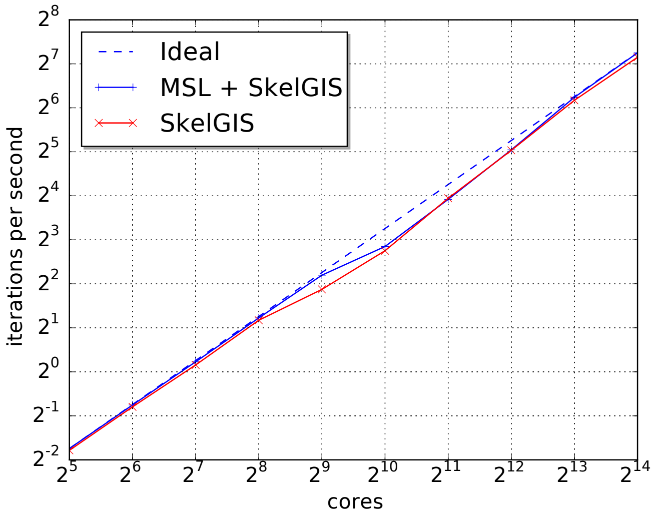


Figure 10: Strong scaling on a $10k \times 10k$ domain.

observed difference. In any case, these difference are barely noticeable and often included in the error bar of the measure and both version globally perform similarly. The MPI synchronizations performed by MSL perform as good as manually specified ones, and L^2C does not introduce any noticeable overhead.

Strong scaling Figure 10 shows the number of iteration per second for a 10k×10k global domain size from 16 to 16.384 cores. The ideal strong scaling is also illustrated.

First, one can notice that the strong scaling evaluated for the code generated by MSL is close to the ideal speedup up to 16.384 cores, which is a very good result. Moreover, no overheads are introduced by MSL which shows that automatic synchronization detections and automatic fusion detections are the same one that the one written manually into the SkelGIS code of FullSWOF2D. 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.

Fusion Finally, to evaluate the data parallelization technique automatically introduced by MSL, the fusion optimization is evaluated. Figure 11 shows the number of iterations per second as a function of the number of cores, for FullSWOF2D with and without fusion optimization and onto a 500×500 domain. As explained in Section 6.3, the MSL fusion happens at a high level and is most of the time done naturally by a computer scientist. However, for a non computer scientist which write its numerical codes, an automatic proposition of such fusions makes the implementation easier. Moreover, one can notice that the performance is clearly improved (around 40%) by this fusion.

8.3 Hybrid parallelism

In this evaluation, we introduce task parallelism to evaluate the hybrid parallelization offered by MSL that mixes data parallelization and task parallelization. In the implementation evaluated in this paper and in addition to SkelGIS, the hybrid parallelization relies on OpenMP.

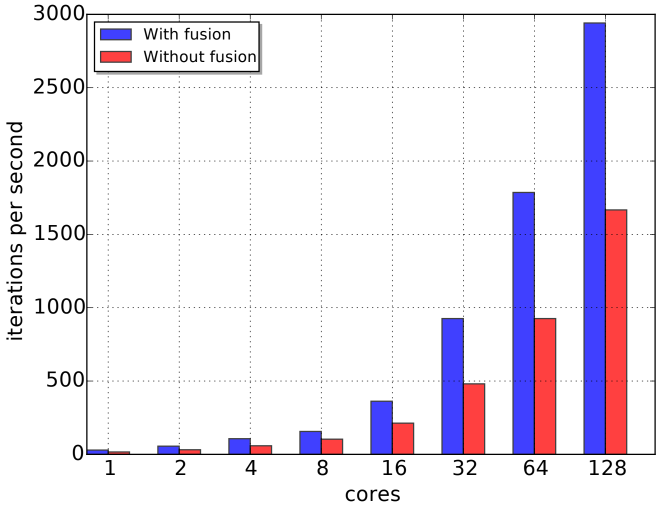


Figure 11: Strong scaling on a 500x500 domain, with and without the fusion optimization.

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

Table 3: Parallelism level (number of parallel tasks) and the number of times this level appears.

The series-parallel tree decomposition TSP of this simulation, extracted by the MSL compiler, is composed of 17 sequence nodes and 18 parallel nodes. Table 3 represents the number of time a given level of parallelism, *i.e.,* the number of tasks to perform concurrently, is observed in the final tree. 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 threads are dedicated to task parallelism, two parts of the code at least will not take advantage of this and that no part of the code would benefit from more than 16 threads. The task parallelism alone is therefore clearly not enough to take advantage of even a single node of a cluster that typically supports today between 16 and 32 threads (16 for Curie).

On the other hand, Figure 12 illustrates limitations of data parallelization technique alone. While the computation time decreases linearly with the number of core used, the communication behavior is much more erratic. Between 2 and 16 cores, the communication happen inside a single node and the time is small and nearly constant. There is a small oscillation that might be explained by difference between number of processors that are power of 4, where the communications are all the same, and those that are nota power of 4, where there is an imbalance. Starting with 32 cores, there are multiple nodes and 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 time from 32 to 128 cores where the data sizes communicated by each process decreases. The increased observed after that 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 used increases, the computation/communication ratio becomes poorer and poorer. As a result, the data parallelism alone fails to provide enough

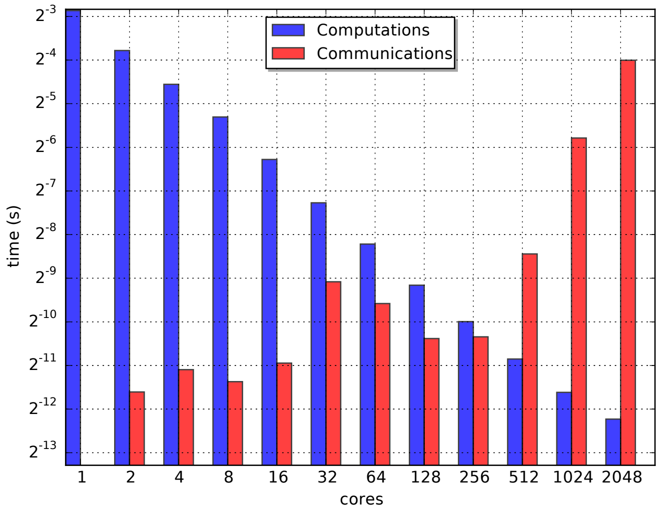


Figure 12: Computation vs communication times in the data parallelization technique.

parallelism to leverage the whole machine and other sources of parallelism have to be found. The blue curve of Figure 13 shows the total execution time for the same example as in Figure 12 and as expected, the speedup bends down from 256 to 2048 cores.

In addition to the blue curve, Figure 13 shows strong scalings for the same example but using an hybrid parallelization. For example, the purple curve shows the parallelization which uses 8 cores, to perform the task parallelization, for each process used for data parallelization (*i.e.,* MPI process). As a result, for example, when using 2 machines of the 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 ($4 \times 8 = 32$).

As a result, and as explained in Section 6.2, data is less divided into sub-domains and the effect which is observe onto the blue curve is delayed. This figure shows a comparison with 2, 4, 8 and 16 cores per MPI process for task parallelization.

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

First, thin nodes of the TGCC Curie are built with two NUMA nodes 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 nodes. This phenomena is illutrated in Figure 14. In this figure, a different strategy is used to bind threads onto available cores (using OpenMP). This strategy, called *spread* (instead of *close* in Figure 13), binds threads on cores in order to spread as much as possible onto resources, which means that the two NUMA nodes are directly used whatever the number of cores kept for task parallelization. As a result, and as shown in the figure, using 2, 4 and 8 cores an initial overhead is introduced.

The second phenomena, which happens in Figure 13 with 16 cores for tasks parallelization, is due to the level of parallelization introduced by the task parallelization technique. Actually, as illsutrated in Table 3, only two steps of the TSP static scheduling generated by the MSL compiler can take advantage of 16 cores among a total of 18 steps. This phenomena has been explained

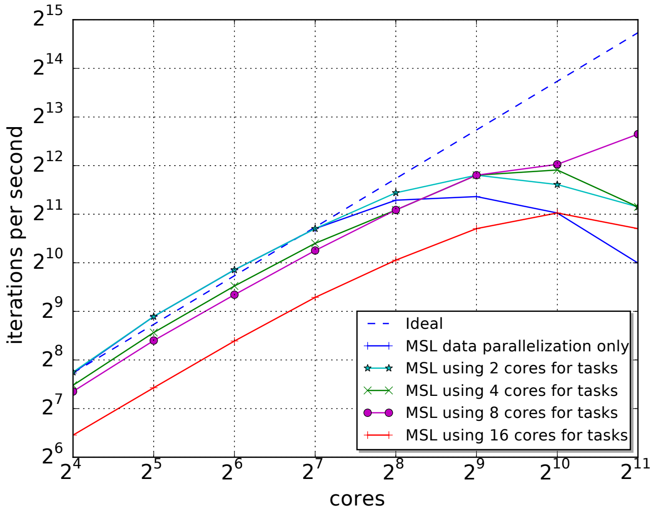


Figure 13: Strong scaling comparisons between data parallelization and hybrid parallelization. A *close* OpenMP clause is used to bind threads onto cores.

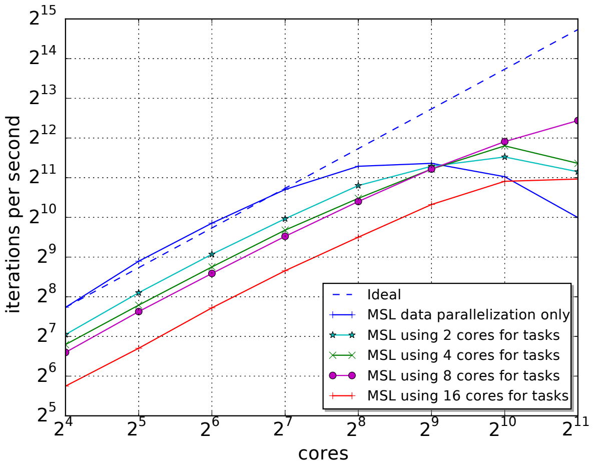


Figure 14: Strong scaling comparisons between data parallelization and hybrid parallelization. A *spread* OpenMP clause is used to bind threads onto cores.

in Section 6.2 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 for task parallelization in Figure 14 is still less efficient than using 8 cores even if the two NUMA nodes are always used in this evaluation.

Finally, to completely validate the performance model introduced in Section 6.2, and to understand when the hybrid parallelization becomes more interesting than the data parallelization, Figure 15 represents T_{COM1} and $T_{COM2} + T_{task}$ of Equation (7), for the best case (*i.e.*, when 8 cores are used for task parallelization), and with a *close* OpenMP bind of threads onto cores. Table 4 gives time details.

	T_{COM1}	T_{COM2}	T_{task}	Equation (7)
16 cores (2 × 8)	0.0005	0.00032	0.013	False
32 cores (4 × 8)	0.0018	0.00045	0.0062	False
64 cores (8 × 8)	0.0013	0.00038	0.0034	False
128 cores (16 × 8)	0.00075	0.0005	0.0023	False
256 cores (32 × 8)	0.00077	0.0018	0.001	False
512 cores (64 × 8)	0.0029	0.0013	0.00052	True
1024 cores (128 × 8)	0.018	0.00075	0.00029	True
2048 cores (256 × 8)	0.0623	0.00077	0.00016	True

Table 4: Execution times (seconds) of T_{COM1}, T_{COM2} and T_{task} for 8 cores for task parallelization. Verification of the Equation (7).

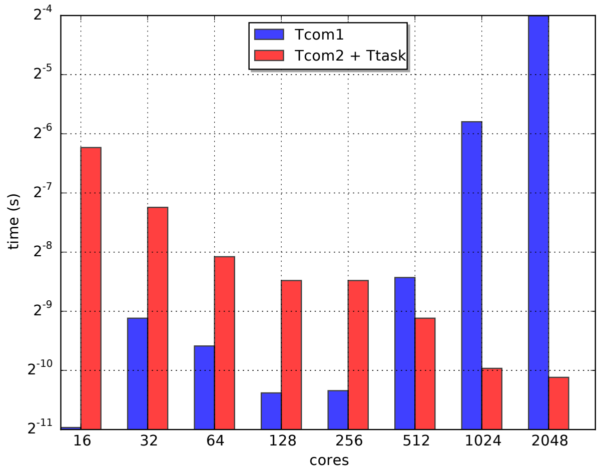


Figure 15: Execution times (seconds) of T_{COM1} and $T_{COM2} + T_{task}$ for 8 cores for task parallelization. Verification of the Equation (7).

Figure 15 and Table 4 perfectly matches results observed in Figure 13 for 8 cores used for task parallelization per core used for data parallelization. As a result, the hybrid parallelization becomes better with a total of 512 cores.

9 Conclusion

We have presented the Multi-Stencil Language (MSL), a domain specific language for stencil-based numerical simulations. Compared to other existing stencil DSLs, MSL proposes a higher abstraction level for end-users. MSL offers a way to describe a numerical simulation independently from the type of mesh used. As a result, the language can be used for a larger number of simulations. It helps to clearly separate the description of the simulation from implementation choices, which facilitates reuse of existing languages and libraries, for example languages for data structures optimizations and parallelization strategies.

In this paper, the formal definition of a multi-stencil program has been given. This formalism helps to understand what is defined into the light grammar of MSL, and also how parallelization can be extracted from it. MSL produces an empty parallel pattern of the numerical simulation described. This pattern indicates where synchronizations are needed, but not how they are performed, which is dependent from the type of mesh and from implementation choices. This second step is left to other existing languages, which are chosen as a back-end of MSL. For example, in this paper the chosen back-end is a code using SkelGIS [5–7], a templated C++ language for distributed meshes (using MPI), and OpenMP [8].

Experiments presented in this paper show that the back-end code produced by MSL, *i.e.,* by a mesh-agnostic language, does not introduce overheads up to 16.384 cores, and that the hybrid (data and task) parallelism automatically introduced by the language, which combines SkelGIS and OpenMP as a back-end, improves performance compared to a data parallelization only. Thus, with the same code asked to the user, two different parallel versions of the simulation can be obtained.

However, if performances are improved by the hybrid parallelization, this performance is also limited by the scheduling strategy used in this paper. Actually, as described in Section 6, a static series-parallel (fork-join) scheduling is performed in this work. This helps to define a predictive performance model, which has been validated in experiments. However, getting rid of global synchronizations (join) could be an interesting improvement to get better performance, especially for irregular simulations, where tasks are not naturally balanced. Our immediate perspective, thus, is to use OpenMP dynamic schedulers [1,12,22] as back-ends to study performance improvments. Using such dynamic schedulers also opens to more heterogeneous architectures, such as GPUs or many-cores accelerators.

Finally, to validate the separation of concerns introduced by MSL, a second back-end for unstructured meshes could be interesting. We think about PaMPA [14] as a distributed data structure backend, instead of SkelGIS. Thus using a single language, the MSL language, two different kinds of numerical simulations could be defined: one onto Cartesian meshes; and one on unstructured meshes.

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 and x2016067617 made by GENCI.

References

[1] 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.

[2] Julien Bigot, Zhengxiong Hou, Christian PÃ©rez, and Vincent Pichon. A low level component model easing performance portability of hpc applications. *Computing*, 96(12):1115–1130, 2014.

[3] Jean-Sylvain Camier. Improving performance portability and exascale software productivity with the &nabla; 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.

[4] 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.

[5] H. Coullon and S. Limet. The SIPSim implicit parallelism model and the SkelGIS library. *Concurrency and Computation: Practice and Experience*, 2015.

[6] 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.

[7] Hélène Coullon and Sébastien Limet. Implementation and performance analysis of skelgis for network mesh-based simulations. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 439–450, 2014.

[8] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[9] 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.

[10] ETP4HPC. ETP4HPC Strategic Research Agenda Achieving HPC leadership in Europe. Technical report, ETP4HPC, 2013.

[11] 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.

[12] 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.

[13] R.L. Graham. The MPI 2.2 Standard and the Emerging MPI 3 Standard. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 2–2, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[19] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag.

[20] 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.

[21] 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.

[22] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *29th IEEE International Parallel & Distributed Processing Symposium*, Hyderabad, India, May 2015.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399