Contents

1 Computational model of Multi-Stencil Programs

To numerically solve a set of PDEs, iterative methods (finite difference, finite volume or finite element methods) are frequently used to approximate the solution through a discretized (step by step) phenomena. Thus, the continuous time and space domains are discretized so that a set of numerical computations are iteratively (time discretization) applied onto a mesh (space discretization). In other words, the PDEs are transformed to a set of numerical computations applied at each time step on elements of the discretized space domain. Among those numerical computations is found a set of numerical schemes, also called *stencil computations*, and a set of auxiliary computations also needed to perform the simulation, and also called *local computations*. This section gives formal definitions of what we call a *stencil program* and its computations. Then, the different mid-grain parallelization techniques which can be applied on such program, are presented.

1.1 Time, mesh and data

$\Omega = \mathbb{R}^n$ is the continuous space domain of a numerical simulation. A mesh $\mathcal{M}$ defines the discretization of the continuous space domain Ω of a set of PDEs and is defined as follows.

Definition *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.*

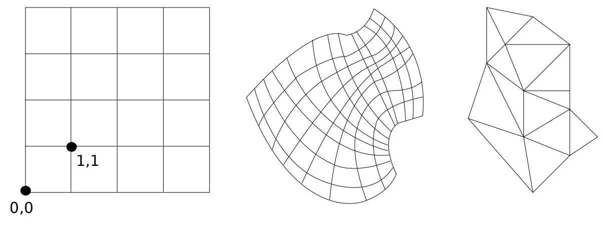


Figure 1: From left to right, Cartesian, curvilinear and unstructured meshes.

Definition *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) and hybrid as illustrated in Figure 1.

Definitions

- *An entity e of a mesh $\mathcal{M} = (V, E)$ is defined as a subset of its vertices and edges, $e \subset (V \cup E)$.*
- *A mesh entities group is denoted E such that $E \in \mathcal{P}(V \cup E)$. It represents a set of entities of the same type.*
- *Finally the set of mesh entities groups of a simulation is denoted $\mathcal{E}$.*

For example, in a 2D Cartesian mesh an entity could be a cell, made of four vertices and four edges, or simply a vertex. As a result a group of cells and a group of vertices could be defined in $\mathcal{E}$.

This covers the space discretization, however there is also the time dimension which has to be discretized in the simulation.

Definitions

- *We denote a scalar as an identifier associated to a numerical value and applied onto a mesh $\mathcal{M}$ where $dim(\mathcal{M}) = 0$. In other words a scalar can be seen as a variable containing a numerical value. The set of scalars is denoted $\mathcal{S}$.*
- *$\mathcal{T} = \mathbb{R}$ is the continuous time domain of a numerical simulation.*
- *The discretization of the continuous time domain $\mathcal{T}$ is denoted as the pair $T(\Delta t, conv)$.*
  - *$\Delta t \in \mathcal{S}$ represents the time interval of the numerical simulation, such that for a current time iteration t_i, the next time iteration is $t_{i+1} = t_i + \Delta t$. The default value of Δt is 1.*
  - *$conv : \mathcal{S}^n \rightarrow bool$ is a function which returns a boolean from a set of scalar variables. This function represents the convergence criteria of the simulation.*
  - *At a given time step, the convergence criteria is evaluated such that if conv returns true the next time step can start such that $t_{i+1} = t_i + \Delta t$, otherwise the simulation ends. For example, the simplest conv is typically to have two scalars: t the time, and a fix number of iterations denoted it. At each time step $\Delta t = 1$, and conv returns the boolean expression $t < it$.*

In a numerical simulation, as a set of scalars can be applied onto a mesh with a nil dimension, a set of data elements, or quantities, can be applied onto meshes of dimension superior or equal to zero. Those quantities represent, as well as scalars, the set of values to compute, or to use, for computations.

Definitions

- *A quantity is a function $\delta : E_\delta \mapsto V_\delta$ which associates each entity of a group $e \in E_\delta$ to a value $v \in V_\delta$ where V_δ is a data type, typically $\mathbb{R}$, $\mathbb{N}$ or $\mathbb{C}$.*
- *The set of quantities applied onto the mesh is denoted Δ.*

- *In the rest of this paper, the group of mesh entities on which a quantity δ is mapped is denoted $entity(\delta) = E_\delta$.*

Another option, closer to the applied mathematics domain, would have been to define δ as the function over $E_\delta \times T$; however the approach chosen for now in this work is to let the user be aware of the number of data he is using and what exactly for.

1.2 Computations

In this section are considered two different types of computations, a reduction or a numerical kernel.
Definitions

- *A computation domain D is a subpart a mesh entities group, $D \subset E \in \mathcal{E}$.*
- *The set of computation domains of a numerical simulation is denoted $\mathcal{D}$.*
- *A neighborhood n is a function which for a given entity $e \in E_i$, returns a set of m entities in E_j, $n : E_i \to E_j^m$. One can notice that $i = j$ is possible.*
- *The set of neihborhood functions in a numerical simulation is denoted $\mathcal{N}$.*
- *A single data is written during a computation and is denoted $w \in \Delta$. Thus, the computation domain of the computation is $D \subset E_w = entity(w)$.*
- *A set of data to read and use in a computation is denoted R. Each element of R is a tuple (r, n), where $r \in \Delta$ and n is a neighborhood function such that $n : E_w \to entity(r)^m$. The neighborhood indicates which entities, on which r is applied, will be read during the computation.*
- *A numerical expression $exp_w : entity(w) \times R^n \to V_w$ is a function that computes the value of the written quantity $w \in \Delta$ for a given entity $e \in entity(w) = E_w$, using a set of input data R.*
- *A computation kernel k of a numerical simulation is defined as $c(R, w, exp, D)$, where R is the set of data read, $w \in \Delta$ is the unique data written, exp is a numerical expression, and $D \in \mathcal{D}$ is the computation domain.*

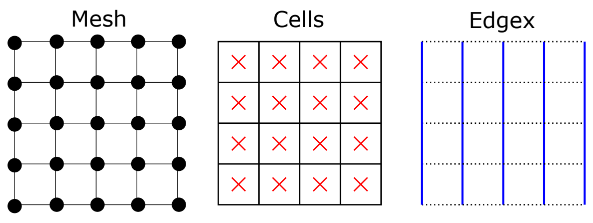
Most of the time we can dissociate two types of kernel computations.
Definitions

- *We denote by identity the identity function $x = x$.*
- *A kernel computation $k(R, w, exp, D) \in \Gamma$ is a stencil kernel $\iff \exists (r, n) \in R$ such that $n \neq identity$.*
- *A kernel computation $k(R, w, exp, D) \in \Gamma$ is a local kernel $\iff \forall (r, n) \in R$, $n = identity$.*

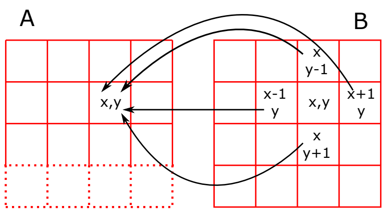
Property *A kernel for which all data read and written are applied onto a mesh of dimension* 0 *is a local kernel.*

A reduction computation is a computation which takes data applied onto meshes and returns a data with a dimension reduced to zero, in other words a scalar. A reduction is typically used to compute the convergence criteria of the time loop of the simulation. Occasionally reductions can also be performed during a time iteration.

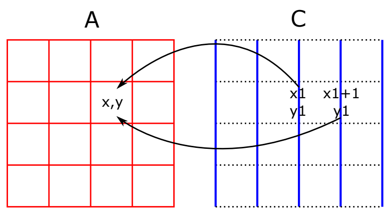
Definitions



(a) Mesh and mesh domains.



(b) 4-neighborhood stencil.



(c) 4-neighborhood stencil.

Figure 2: (a) a Cartesian mesh and two kind of mesh entities, (b) an example of stencil kernel on cells, (c) an example of stencil kernel on two different entities of the mesh.

- *A reduction computation r of a numerical simulation is defined as $r(R, w, red)$, where R is the set of data read, $w \in \mathcal{S}$ is the scalar written, and red is an associative reduction function.*
- *A reduction function $red : \Delta^n \rightarrow \mathcal{S}$ is an operation which reduces a set of data in Δ (on it sentities) to a single scalar variable in $\mathcal{S}$. The reduction function must be associative.*
- *The set of n ordered computations (kernel or reduction) of a numerical simulation is denoted $\Gamma = [c_i]_{0 \leq i \leq n-1}$, such that $\forall c_i, c_j \in \Gamma$, if $i \leq j$, then c_i is computed before c_j.*
- *Finally, a multi-stencil program is defined by the septuplet $\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \mathcal{S}, \Gamma)$.*

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:

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

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

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:

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

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

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

2 Parallelization of Multi-Stencil Programs

Multi-stencil mesh-based numerical simulation can be parallelized in various ways and is an interesting kind of application to take advantage of modern heterogeneous HPC architectures, mixing clusters, multi-cores CPUs, vectorization units, GPGPU and many-core accelerators.

2.1 Data parallelism

In a data parallelization technique, the idea is to split the data 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 $\mathcal{MSP}(T, \mathcal{M}, \mathcal{E}, \mathcal{D}, \Delta, \mathcal{S}, \Gamma)$ 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 [] and is not adressed by this paper. As a data is mapped onto the mesh, the set of data Δ is partitionned the same way than the mesh in $\Delta = \{\Delta_0, \ldots, \Delta_{p-1}\}$. 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 *For n the number of computations in Γ, and for i, j such that $i < j < n$, a synchronisation is needed between c_i and c_j, denoted $c_i \lessdot c_j$, if $\exists(r, n) \in R_j$ with $n \neq identity$ (c_j is a stencil computation), and such that $\{w_i\} \subset R_j$. Moreover, the data to update is $\{w_i\} \cap R_j = \{w_i\}$.*

Actually, a synchronization can only be needed by the data read by a stencil computation (not local), and only if this data 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.

Definition *A synchronization between two computations $c_i \lessdot c_j$ is defined as a specific computation $sync(R, w, exp, D)$ where $R = \{w_i\}$, $w = w_i$, $exp = identity$ and $D = \emptyset$.*

Definition If a synchronization is needed before c_j (on the data w_i), c_j is replaced in Γ_{sync} by the list

$$\Gamma_{sync} = [sync(\{w_i\}, w_i, identity, \emptyset), c_j]$$

Definition The concatenation of two ordered lists of respectively n and m computations $l_1 = [c_i]_{0 \leq i \leq n-1}$ and $l_2 = [c'_i]_{0 \leq i \leq m-1}$ is denoted $l_1 \cdot l_2$ and is equal to a new ordered list $l_3 = [c_0, \ldots, c_{n-1}, c'_0, \ldots, c'_{m-1}]$.

Definition From the ordered list of computation Γ, a new ordered list Γ_{sync} is obtained from the call $\Gamma_{sync} = F_{sync}(\Gamma, 0)$, where F_{sync} is the recursive function defined as

$$F_{sync}(\Gamma, j) = \begin{cases} [\Gamma[j]] \cdot F_{sync}(\Gamma, j+1) & \text{if } \forall i < j,\ c_i \not\lessdot c_j \\ [sync(\{w_i\}, w_i, identity, \emptyset), c_j] \cdot F_{sync}(\Gamma, j+1) & \forall i < j, \text{ and } c_i \lessdot c_j\} \\ [] & \text{if } j = |\Gamma|. \end{cases}$$

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

$$\mathcal{MSP}_k(T, \mathcal{M}_k, \mathcal{E}, \mathcal{D}, \Delta_k, \mathcal{S}, \Gamma_{sync}), \tag{1}$$

is performed.

2.2 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 be statically scheduled. In this paper, we introduce task parallelism by building the dependency graph between computations of the sequential list Γ_{sync} (which itself take into account data parallelism).

Definition For two computations $c_i(R_i, w_i, \exp_i, D_i)$ and $c_j(R_j, w_j, \exp_j, D_j)$, with $i < j$, it is said that c_j is data dependant from c_i, denoted $c_i \prec c_j$, if $\{w_i\} \cap R_j \neq \emptyset$. In this case, c_i has to be computed before c_j. The binary relation $c_i \prec c_j$ represents a *dependency*.

Definition 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, following the direction of edges, 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 $c_i(R_i, w_i, \exp_i, D_i)$ and $c_j(R_j, w_j, \exp_j, D_j)$, with $i < j$, such that $c_i \prec c_j$.

Definition 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 From the ordered list Γ_{sync} of computations $c(R, w, \exp, D)$, a directed dependency graph $\Gamma_{dep}(V, A)$ is obtained from the call $T_{dep}(\Gamma_{sync}, 0)$, where F_{dep} is the recursive function

$$F_{dep}(\Gamma_{sync}, j) = \begin{cases} (\{\}, \{\}) & \text{if } j = |\Gamma_{sync}| \\ (c_j, \{(\widehat{c_i, c_j}), \forall i < j,\ c_i \prec c_j\}) \cup F_{dep}(\Gamma_{sync}, j+1) & \text{if } j < |\Gamma_{sync}| \end{cases}$$

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

Proposition 2.1 *The directed graph Γ_{dep} is an acyclic graph.*

Proof Γ_{dep} is built from Γ_{sync} which is an ordered and sequential list of computations. Moreover, each computation of the list Γ_{sync} is associated to a vertex of V, even if the same computation is represented more than once in Γ_{sync}. As a result it is not possible to go back to a previous computation and to create a cycle.

As a result, the hybrid parallelization of a multi-stencil program $\mathcal{MSP}$, produces k multi-stencil programs

$$\mathcal{MSP}_k(T, \mathcal{M}_k, \mathcal{E}, \mathcal{D}, \Delta_k, \Gamma_{dep}).$$

The set of computations Γ_{dep} is a dependency graph between computations c_i of Γ and the synchronization computations $sync_i$ added into Γ_{sync}. Γ_{dep} can be built from the call to

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

3 The Multi-Stencil Language

4 Static Scheduling

References

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

[2] Julien Bigot and Christian Pérez. Increasing Reuse in Component Models through Genericity. Research Report RR-6941, 2009.

[3] Lucian Finta, Zhen Liu, Ioannis Milis, and Evripidis Bampis. Scheduling uet-uct series-parallel graphs on two processors. *Theor. Comput. Sci.*, 162(2):323–340, August 1996.

[4] Lucian Finta, Zhen Liu, Ioannis Mills, and Evripidis Bampis. Scheduling uet-uct series-parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323 – 340, 1996.

[5] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[6] 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.

[7] Ji-Bo Wang, C.T. Ng, and T.C.E. Cheng. Single-machine scheduling with deteriorating jobs under a series–parallel graph constraint. *Computers & Operations Research*, 35(8):2684 – 2693, 2008. Queues in Practice.

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 0249-0803