A Component-Based and Parallel Multi-Stencil DSL

Héléne Coullon, Christian Perez

RESEARCH REPORT
N° 7003
April 2015
Project-Teams Avalon

ISSN 0249-6399 ISRN INRIA/RR--7003--FR+ENG

A Component-Based and Parallel Multi-Stencil DSL

Héléne Coullon[*][†], Christian Perez[‡]

Project-Teams Avalon

Research Report n° 7003 — April 2015 — 20 pages

Abstract: abstract

Key-words: No keywords

[*] Footnote for first author
[†] Shared foot note
[‡] Footnote for second author

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Exemple de document utilisant le style rapport de recherche Inria

Résumé : Resume en francais

Mots-clés : mots-cles

Contents

1 Introduction

2 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 all 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*. In the following Section, formal definitions of a *stencil program* and its computations are given. Then, the different parallelization techniques which can be applied on such program, are presented.

2.1 Time, mesh and data

A mesh $\mathcal{M}$ defines the discretization of the continuous space domain Ω of a set of PDEs and is defined as followed.

Definition *A mesh is a connected undirected graph $\mathcal{M} = (V, E)$, where V is the set of vertices and E the set of edges. The set of edges E of a mesh $\mathcal{M} = (V, E)$ does not contain bridges.*

Definition D_i is a set of elements of a mesh $\mathcal{M} = (V, E)$, constructed by a function dom_i which defines a precise association between V and E, $dom_i : V \times E \to D_i$.

For example, the set of cells D_0 in a Cartesian 2D mesh could be defined by exactly four vertices and four edges connected as a cycle. But we could also define another set of elements D_1 as the simple set of vertices V etc.

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

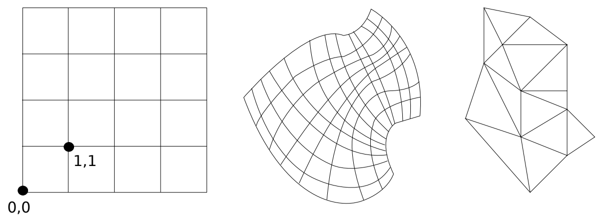


Figure 1: From left to right, Cartesian, curvilinear and unstructured meshes.

Definition The discretization of the continuous time domain $\mathcal{T}$ is denoted T such that $\forall\ t_i,\ t_{i+1} \in T$, $\exists\ \Delta t \in \mathbb{R}$, $t_{i+1} = t_i + \Delta t$. Thus, T is responsible for the iteration time steps of the numerical simulation.

In a numerical simulation a set of data, or quantities, are applied onto the mesh and represent the set of values to compute, or to use, for computation.

Definition The set of data applied on the mesh is denoted by Δ, such that $\delta \in \Delta$ is a function which associates each element $d \in D_i$ (the domain it is applied on) to a value $v \in V$, $\delta : D_i \to V$. In the rest of this paper, the domain of a data δ can be given by the function $domain(\delta) = D_i$.

One can notice that in applied mathematics, the signature of δ would be $\delta : D_i \times T \to V$, however when programming a numerical simulation it is not wise to store all values of each time iteration.

2.2 Computations

Definition A numerical expression exp is a function which represents how to compute a value for an element $d \in domain(w) = D_i$, using the set $R \subset \Delta$ of input data (read data), exp : $R \times D_i \to w \times D_i$.

Definition A computation c of a numerical simulation is defined as $c(R, w, \text{exp})$, where $R \subset \Delta, w \in \Delta$ and exp a numerical expression.

It has to be noticed that at each time iteration, all the elements of a mesh are computed. However, it happens that computations of the mesh elements are splitted in different domains, as for example the computation of the physical border. In this case additional D_i can be specified for the mesh $\mathcal{M}$.

Definition The set of n ordered computations 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, and c_j can be computed only when c_i is finished.

Definition A *multi-stencil program* is defined by the quadruplet $\mathcal{MSP}(T, \mathcal{M}, \Delta, \Gamma)$.

As already mentioned, the ordered list Γ can be composed of two different types of computations, stencil and local computations, which will be defined in the rest of this Section.

Definition The neighborhood $\mathcal{N}$ of an element $d \in D_i$ is a function to obtain a set of elements in any $D_k \subset \mathcal{M}$, $\mathcal{N} : D_i \to D_k \times D_k \times \ldots$.

The function $\mathcal{N}$ is also sometimes called the *stencil shape*, or the *stencil* in applied mathematics. In this paper we distinguish a stencil shape from a *stencil computation* defined as followed:

Definition A *stencil computation* is defined as a quadruplet $s(R, w, \exp, \mathcal{N})$, where $R \subset \Delta$, and $w : D \to V \in \Delta$.

In a stencil computation s, $\forall d \in D$, the stencil numerical expression exp is applied such that $w(d) = \exp(R(d), R(\mathcal{N}(d)))$. In this work, a stencil computation $s(R, w, \exp, \mathcal{N})$ always verifies $R \cap \{w\} = \emptyset$, otherwize an implicit numerical scheme has to be solve which is over the scope of this paper. As a result, the ordered list Γ of a multi-stencil program can be composed of a set of stencil computations applied on one or more stencil shapes.

Figure 2 gives an example of a stencil computation $s(R, w, \exp, \mathcal{N})$, where $\mathcal{M}(V, E)$ is a two dimensional Cartesian mesh. A single domain $D = \mathit{domain}(w)$ is defined in this example and is composed of cells formed by a cycle of four vertices $v \in V$ and four edges $e \in E$. Furthermore, in this example $R = \{A\}$, $w = B$, and for $(x, y) \in D$ the neighborhood function is

$$\mathcal{N} : (x, y) \to \{(x, y+1), (x, y-1), (x+1, y), (x-1, y)\}.$$

Finally, the numerical expression of this example is

$$\exp(A(x, y), A(\mathcal{N}(x, y)) = B(x, y) = A(x, y) + (A(x, y+1) + A(x, y-1) + A(x+1, y) + A(x-1, y))/4.$$

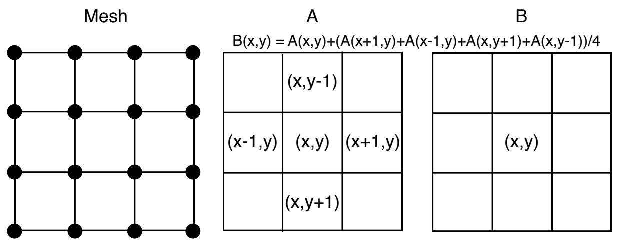


Figure 2: Example of a stencil computation.

The second type of numerical computation is a local computation.

Definition A local computation is a triplet $l(R, w, \text{exp})$, where *exp* does not involve a neighborhood function $\mathcal{N}$.

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

3 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.

3.1 Data parallelism

In a data parallelization technique, the idea is to split the data on which the program is computed in 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 $\mathcal{MSP}(T, \mathcal{M}, \Delta, \Gamma)$ 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_{data}.

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 $c_j = s_j(R_j, w_j, \text{exp}_j)$, and $\{w_i\} \subset R_j$. 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 the neighborhood function $\mathcal{N}$ 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 $\text{update}(\{w_i\} \cap R_j) = \text{update}(w_i)$.

Definition An *updated-computation* is a computation which needs a set of data synchronizations before it can be performed. The computation c_j is transformed to an updated-computation $c_j^*(R_j, w_j, \text{exp}_j, *_j)$ if $\bigcup_i \{w_i\} \neq \emptyset$, $\forall i < j$, and $c_i \lessdot\!\!< c_j$. In such a case $*_j = \mathit{update}(\bigcup_i \{w_i\})$ is performed before the evaluation of exp_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_{data} is obtained from the call $\Gamma_{data} = T_{data}(\Gamma, 0)$, where T_{data} is the recursive function defined as

$$T_{data}(\Gamma, i) = \begin{cases} [\Gamma[i]] \cdot T_{data}(\Gamma, i+1) & \text{if } \forall j < i,\, c_j \not\prec\!\!\!\prec c_i \\ [c_i^*(\Gamma[i], \text{update}(\bigcup_k \{w_k\})] \cdot T_{data}(\Gamma, i+1) & \forall k < i,\, c_k \prec\!\!\!\prec c_i \\ [] & \text{if } i = |\Gamma|. \end{cases}$$

The construction of Γ_{data} implies that if a synchronization is needed it is always proceeded before the computation wich needs it. One can notice that a combination of synchronizations is possible in some cases, and thus it is possible to synchronize more than one data at a time. For this reason, we define a reduction which can be applied to Γ_{data}, and which is performed as a post-transformation. This reduction, could propose better performance depending on the used network speed.

Proposition 3.1 *Denoting four computations c_l, c_k, c_i and c_j of Γ, with $l < k$, $i < j$, and where $c_l \prec\!\!\!\prec c_k$ and $c_i \prec\!\!\!\prec c_j$. Then, computations c_k and c_j are tansformed to $c_k^*(c_k, update(w_l))$ and $c_j^*(c_j, update(w_i))$. If $i < k$, a single update call can be performed at k such that $c_k^*(c_k, update(w_l \cup w_i))$, and in this case, the computation c_j is not transformed.*

Proof By definition, the synchronization of w_i can be performed between c_i and c_j. As $i < k$ the computation c_i is not performed between k and j. As a result, the synchronization of w_i can be performed between c_i and c_k.

In other words, for each synchronization in Γ_{data}, if the computation c_i from which c_j is synchronized occurs before the last synchronization, the two synchronizations can be reduced to one with the union of the data to update.

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

$$\mathcal{MSP}_k(T, \mathcal{M}_k, \Delta_k, \Gamma_{data}), \tag{1}$$

is performed.

3.2 Task 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. In a multi-stencil program a task is defined as a computation $c(R, w, \text{exp})$. As a result, the dependency graph of a multi-stencil program represents the dependencies between local and stencil computations of Γ.

Definition For two computations $c_i(R_i, w_i, \text{exp}_i)$ and $c_j(R_j, w_j, \text{exp}_j)$, 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*.

Proposition 3.2 *The binary relation $\prec$ is not transitive.*

Proof Considering three computations $c_0(R_0, w_0, \text{exp}_0)$, $c_1(R_1, w_1, \text{exp}_1)$ and $c_2(R_2, w_2, \text{exp}_2)$, where $\{w_0\} \cap R_1 \neq \emptyset$ and $\{w_1\} \cap R_2 \neq \emptyset$. In this case two dependencies can be extracted $c_0 \prec c_1$ and $c_1 \prec c_2$. However, the dependency $c_0 \prec c_2$ is not true as $\{w_0\} \cap R_2 = \emptyset$.

It has been proved that the binary relation $\prec$ is not transitive. This property is due to the fact that $\prec$ is defined as a data dependency between computations. However, another type of dependency, only due to time, is created from $\prec$.

Definition For two computations $c_i(R_i, w_i, \text{exp}_i)$ and $c_j(R_j, w_j, \text{exp}_j)$, a time dependency is denoted $c_i \blacktriangleleft c_j$ and means that c_i has to be computed before c_j.

As a result, the relation $\blacktriangleleft$ is more general than $\prec$.

Proposition 3.3 *For two computations $c_i(R_i, w_i, exp_i)$ and $c_j(R_j, w_j, exp_j)$ such that $c_i \prec c_j$, $c_i \blacktriangleleft c_j$ is verified.*

Proof By definition $c_i \prec c_j$ means that $\{w_i\} \cap R_j \neq \emptyset$ and that c_i has to computed before c_j.

Proposition 3.4 *The binary relation $\blacktriangleleft$ is transitive.*

Proof Considering three computations $c_0(R_0, w_0, \text{exp}_0)$, $c_1(R_1, w_1, \text{exp}_1)$ and $c_2(R_2, w_2, \text{exp}_2)$, where $c_0 \blacktriangleleft c_1$ and $c_1 \blacktriangleleft c_2$. c_0 is computed before c_1 and c_1 is computed before c_2, as a result c_0 is computed before c_2 and the relation $c_0 \blacktriangleleft c_2$ is verified.

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, a directed dependency graph $\Gamma_{task}(V, A)$ can be built finding all pairs of computations $c_i(R_i, w_i, \text{exp}_i)$ and $c_j(R_j, w_j, \text{exp}_j)$, with $i < j$, such that $c_i \blacktriangleleft c_j$. Those time dependencies can be found from pairs of data dependencies $c_i \prec c_j$.

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 an ordered list Γ of computations $c(R, w, \text{exp})$, a directed dependency graph $\Gamma_{task}(V, A)$ is obtained from the call $T_{task}(\Gamma, 0)$, where T_{task} is the recursive function

$$T_{task}(\Gamma, i) = \begin{cases} (\{\}, \{\}) & \text{if } i = |\Gamma| \\ (c_i, \{\widehat{(c_k, c_i)}, \forall k < i,\, c_k \prec c_i\}) \cup T_{task}(\Gamma, i+1) & \text{if } i < |\Gamma| \end{cases}$$

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

Proposition 3.5 *The directed graph Γ_{task} is an acyclic graph.*

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

Using the function T_{task} to build Γ_{task}, however, duplication of dependencies may occur because of the transitivity of the relation $\blacktriangleleft$. Actually, as the relation $\prec$ verifies the relation $\blacktriangleleft$, and as $\blacktriangleleft$ is transitive, if for three computations $c_k(R_k, w_k, \text{exp}_k)$, $c_i(R_i, w_i, \text{exp}_i)$ and $c_j(R_j, w_j, \text{exp}_j)$, with $k < i < j$ and where $c_k \prec c_i$, $c_i \prec c_j$ and $c_k \prec c_j$, then the transitivity $c_k \blacktriangleleft c_j$ is verified

$$\begin{tikzcd}
c_0 \arrow[r] & c_1 \arrow[r] \arrow[rrr, bend right] & c_2 \arrow[r] & c_3 \arrow[r] & c_4
\end{tikzcd}$$

Figure 3: Useless duplication of the dependency $c_1 \blacktriangleleft c_4$

because of the relations $c_k \blacktriangleleft c_i$ and $c_i \blacktriangleleft c_j$. However, $c_k \blacktriangleleft c_j$ is also directly represented by the binary relation $c_k \prec c_j$. As a result, a duplication of the dependency is created in Γ_{task}. In Figure 3 an example of duplication is given for the relation $c_1 \blacktriangleleft c_4$.

Another view of the relations $\prec$ and $\blacktriangleleft$ is that $c_i \blacktriangleleft c_j$ in Γ_{task} is a path from c_i to c_j of any size, while $c_i \prec c_j$ is a path of size 1 from c_i to c_j. And in any case a path in the graph represents a time dependency between computations. Scheduling a graph dependency which do not contains duplication of information is easier.

Definition A DAG is transitive if it contains an arc $(\widehat{u,v})$ between any two vertices such that there is a path from u to v. The transitive closure of a DAG $G = (V, E)$, is the DAG $G_T = (V, E_T)$ for which E_T is the minimal subset of $V \times V$ that includes E and makes G_T transitive.

Definition An arc $(\widehat{u,v})$ of a DAG is redundant if there is a path from u to v in the DAG that does not include the edge. A DAG that does not contain any redundant arc is called minimal. The transitive reduction of a DAG G is its unique minimal sub-graph having the same transitive closure.

Thus, the removal of the redundant dependencies of Γ_{task} consists in applying a transitive reduction.

3.3 Hybrid parallelism

It is also possible to combine data and task parallelization techniques to get hybrid parallelism, sometimes more efficient on hybrid architectures. A coarse-grain data parallelism creates for k resources k multi-stencil computations $\mathcal{MSP}_k(T, \mathcal{M}_k, \Delta_k, \Gamma_{data})$, where Γ_{data} is built from an ordered list of computations and is itself an ordered list of computations c and updated-computations c^*. On the other hand, the task parallelization technique builds, from an ordered list of computation, a graph dependency Γ_{task}. As a result T_{data} and T_{task} can be composed to build Γ_{hybrid}. Thus, k multi-stencil programs

$$\mathcal{MSP}_k(T, \mathcal{M}_k, \Delta_k, \Gamma_{hybrid})$$

are responsible for an hybrid parallelization of a multi-stencil program. The set of computation Γ_{hybrid} is a dependency graph between computations c and updated-computations c^* and can be built from the call to

$$T_{task}(T_{data}(\Gamma, 0), 0).$$

The rest of this paper deals with the hybrid parallelization of multi-stencil programs described above. The first reason for this choice is that this paper does not adress the portability problem, which however will be a perspective. Second, by adressing the hybrid parallelism case, we also are able to adress data parallelism and task parallelism, from which hybrid parallelism is built.

4 Series-parallel graphs and tree decompositions

In this section we argue that the graphs Γ_{hybrid} or Γ_{task}, previously defined, on which an approximation will be defined, are minimal series-parallel graphs. For this reason, the structure of those graphs can be represented as binary trees of parallel and series compositions of sub-graphs, also called *binary decomposition tree* [6]. First, the needed definitions on series-parallel graphs are given, and second the case of the dag Γ_{hybrid} is studied.

4.1 GSP and MSP classes

A vertex v of a DAG G is a *source* if no edge of G enters v. Similarly, a vertex v is a *sink* if no edge of G leaves v. In 1982, Valdes & Al [6] have defined the class of minimal series-parallel DAGs (MSP).

Definition Minimal Series Parallel

- The DAG having a single vertex and no edges is MSP.
- If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two MSP DAGs, so is either of the DAGs constructed by the following operations:
  - Parallel composition: $G_p = (V_1 \cup V_2, E_1 \cup E_2)$.
  - Series composition: $G_s = (V_1 \cup V_2, E_1 \cup E_2 \cup (N_1 \times R_2))$, where N_1 is the set of sinks of G_1 and R_2 is the set of sources of G_2.

Definition A DAG is *General Series Parallel* (GSP) if and only if its transitive reduction is a MSP DAG.

A *binary decomposition tree* is a tree having a leaf for each vertex of the MSP DAG it represents, and whose internal nodes are labelled S or P to indicate respectively the series or parallel composition of the MSP sub-DAGs represented by the subtrees rooted at S or P. Figures 4a, 4b and 5 respectively give an example of a GSP DAG, its transitive reduction which is MSP, and its tree decomposition.

$$\begin{tikzcd}
& & c \arrow[r] \arrow[dr] & d \arrow[dr] & \\
a \arrow[r] \arrow[urrr, bend left] \arrow[ddr] & b \arrow[ur] \arrow[dr] & & & g \\
& & e \arrow[r] \arrow[ur] & f \arrow[ur] & \\
& h \arrow[r] \arrow[rr, bend right] & i \arrow[r] & j &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
& & c \arrow[r] \arrow[dr] & d \arrow[dr] & \\
a \arrow[r] \arrow[ddr] & b \arrow[ur] \arrow[dr] & & & g \\
& & e \arrow[r] \arrow[ur] & f \arrow[ur] & \\
& h \arrow[r] & i \arrow[r] & j &
\end{tikzcd}$$

(b)

Figure 4: (a) GSP and (b) MSP DAGs

Valdes & Al [6] have also proposed a linear algorithm to know if a DAG is MSP and, if it is, to decompose it to its associated binary decomposition tree. This algorithm is based on the duality of the class of MSP DAGs, with the class of *Two Terminal Series Parallel* DAGs (TTSP) from which a binary tree decomposition can be performed in linear time. As it will be

$$\begin{tikzcd}[column sep=tiny]
& c & & d & & e & & f & & & & & & \\
& & \mathcal{S} \arrow[ul] \arrow[ur] & & & & \mathcal{S} \arrow[ul] \arrow[ur] & & & & & & & \\
& & & & \mathcal{P} \arrow[ull] \arrow[urr] & & & & g & & & i & & j \\
& & b & & & & \mathcal{S} \arrow[ull] \arrow[urr] & & h & & & & \mathcal{S} \arrow[ul] \arrow[ur] & \\
& & & & \mathcal{S} \arrow[ull] \arrow[urr] & & & & & & \mathcal{S} \arrow[ull] \arrow[urr] & & & \\
& & & & & a & & \mathcal{P} \arrow[ulll] \arrow[urrr] & & & & & & \\
& & & & & & \mathcal{S} \arrow[ul] \arrow[ur] & & & & & & &
\end{tikzcd}$$

Figure 5: Binary decomposition tree of the MSP of Figure 4b

explained later, in this paper we are interested in DAGs that we already know to be MSP. Thus, we only give the needed definitions to understand the binary decomposition algorithm, and not the recognition algorithm. To let the reader understand this algorithm, which is modified in this work, the needed definitions are given.

Definition The *line digraph* of a digraph G is a digraph $L(G)$ that has:

- a vertex $f(e)$ for each edge e of G; and
- an edge $(f(e_1), f(e_2))$ for each pair of edges of G of the form $e_1 = (u, v)$, $e_2 = (v, w)$.

Definition Two Terminal Series Parallel

- A digraph consisting of two vertices joined by a single edge is TTSP.
- If G_1 and G_2 are TTSP digraphs, so is the digraph obtained by either of the following operations:
  - *Two terminal parallel composition*: identify the sourc of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2.
  - *Two terminal series composition*: identify the sink of G_1 with the source of G_2.

Theorem If the DAG G is a MSP graph, its *inverse line DAG* $L^{-1}(G)$ is TTSP.

It has to be indicated that for a DAG G, $L^{-1}(G)$ is not unique. Thus, in the work of Valdes & Al, and in this work, $L^{-1}(G)$ will refer to the unique digraph having a single source and a single sink whose line digraph is $L(L^{-1}(G)) = G$.

The binary decomposition tree algorithm is based on the fact that the decomposition can be obtained as a byproduct of a reduction process on $L^{-1}(G)$, which is TTSP. In order to obtain the decomposition, Valdes & Al associate a label with each edge of the digraph being reduced. Initially the label of each edge is a trivial binary tree consisting of a single node. As the reduction process introduces new edges the rules of Figure 6 are used to compute the binary trees used

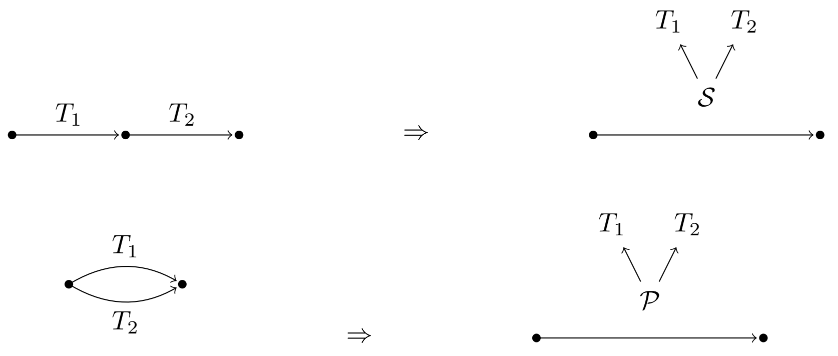


Figure 6: Reduction rules of the decomposition tree algorithm.

to label them. The algorithm ends when the TTSP graph is reduced to its minimum, i.e. two vertices and a single edge between them.

Finally, Valdes & Al [6] have identify a forbidden shape, or subgraph, called N and represented in Figure 7, such that

Theorem A DAG G is GSP if and only if its transitive closure does not contain N as a subgraph.

$$\begin{tikzcd}
a \arrow[r] & b \\
c \arrow[r] \arrow[ur] & d
\end{tikzcd}$$

Figure 7: Forbidden N subgraph shape for a DAG to be GSP.

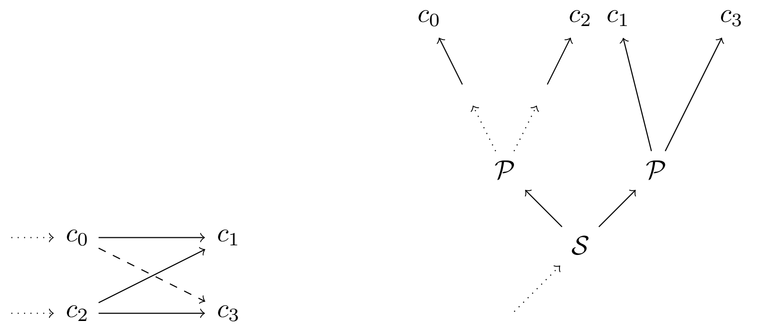
4.2 Multi-stencil programs.

We are interested in GSP and MSP classes of graphs to be able to represent computations of a multi-stencil program as a set of sequences and parallel executions. Actually, such a representation can directly be dumped to a parallel language, in which *sequence* of instructions and *parallel* execution of instructions are defined. Series-parallel trees can also be used as input of scheduling optimizations [4, 7] to improve task parallelism efficiency, which opens more perspectives to this work. Finally, in this paper is presented that such a series-parallel representation can also be dumped to component models by defining specific *control components.* Thus, the proposed DSL inheritates software engineering advantages of component models, such as code re-use, productivity and maintainability.

Returning back to the parallelization formalism of Section 3, two important steps have to be performed to make Γ_{hybrid} a MSP graph:

- transitive reduction,
- deletion of the forbidden N shape.

The transitive reduction has already been applied on Γ_{hybrid} by the transitivity of the relation $\blacktriangleleft$. However, even by applying this transitive reduction it is possible to obtain a DAG which is not MSP. Actually, it is possible in a multi-stencil program to have a set of computations such that their dependencies form the forbidden N subgraph. For example, if c_0, c_1, c_2 and c_3 are computations of a $\mathcal{MSP}$ such that $c_0 \prec c_1$, $c_2 \prec c_1$ and $c_2 \prec c_3$, the *zigzag* relation $c_0 \prec c_1 \succ c_2 \prec c_3$ which form a forbidden subgraph of MSP DAGs is found in Γ_{hybrid}. For this reason, we have made the choice to over-constrain such a case by adding the relation $c_0 \prec c_3$ such that a complete graph is created and can be translated to a series-parallel decomposition as illustrated in Figure 8.



(a) Over-constraint on the forbidden N shape.

(b) Series-parallel tree associated to the over-constraint

Figure 8: Deletion of forbidden subgraphs.

We consider this approximation acceptable for multi-stencil programs, because of the relative homogeneity of computations. Actually, all computations, except the ones on the physical border of the domain, are performed on an entire domain of the mesh, and as a computation performs a single quantity at a time (w_i of c_i is a singleton), the amount of arithmetic operations in a computation are quite homogeneous. However, this approximation can also be responsible for a useless wait for computation c_3 and less performance. For this reason, perspectives of this work open to a new kind of component models.

After these over-constraints are applied, Γ_{hybrid} is a MSP DAG. As a result, the binary tree decomposition algorithm of Valdes & Al can be applied on Γ_{hybrid}. However, because of updated-computations inserted in Γ_{data}, we have to define additional reduction rules for the algorithm. The initial label of the edges of the graph $L^{-1}(\Gamma_{hybrid})$ to reduce are the computations of the multi-stencil program.

A complete example of reduction is given in Figure 8. The initial Γ_{hybrid} DAG and its inverse line graph are given first, followed by the set of reductions. The label of the final single edge corresponds to the binary tree decomposition of the initial Γ_{hybrid} graph.

At this point, the $\mathcal{MSP}$ program can be dumped to any parallel language in which a sequence and a parallel execution are available. For example, we could imagine a parallel functionnal language with a binary *sequence* function and a binary *parallel* function.

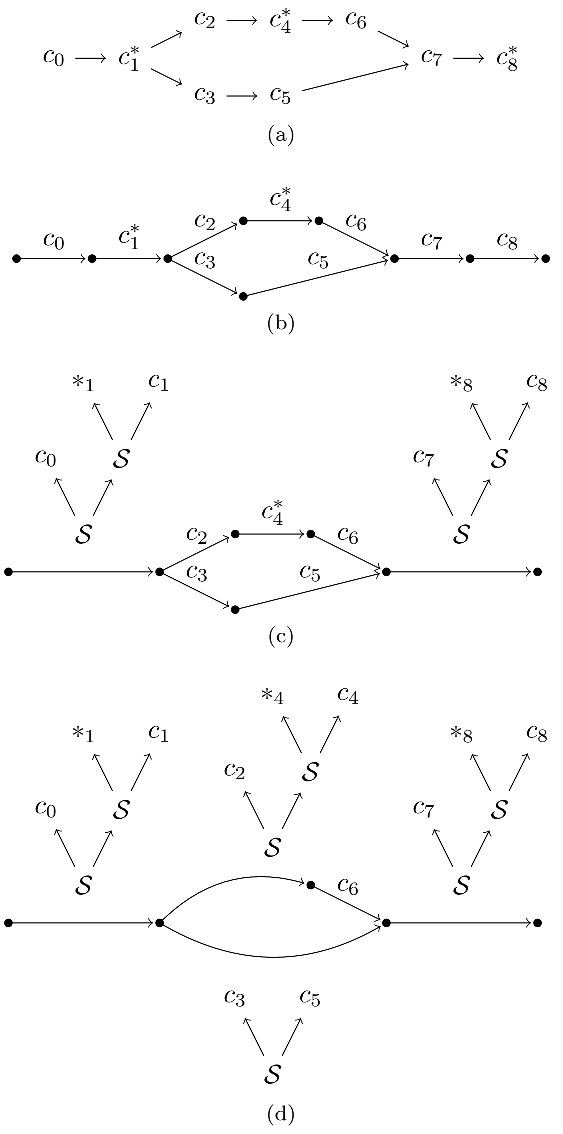
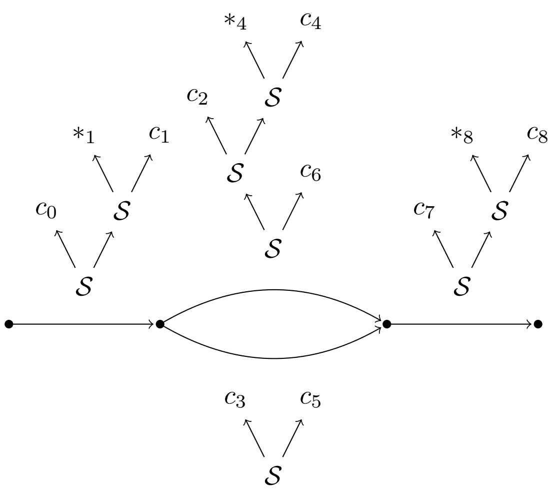


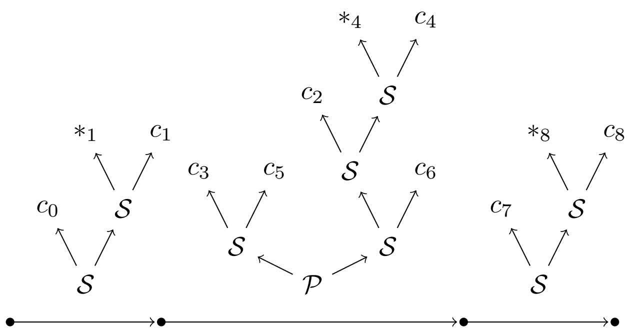
Figure 9: Big example

5 Component models

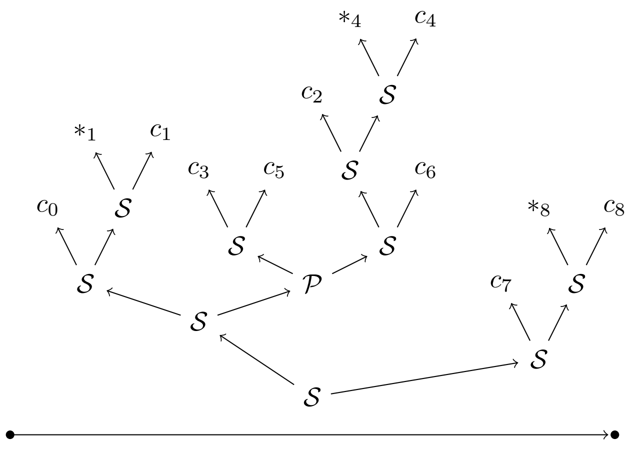
Instead of dumping the series-parallel tree decomposition of Γ_{hybrid} (or Γ_{task}) to a functionnal high abstraction level language, or to an imperative high abstraction level language, such as OpenMP, we have chosen to dump it to a component model assembly. Component model is an interesting domain of software engineering [5] which improves code re-use, scalability and maintainability of applications [2,5]. Moreover, component models bring a separation of concerns in the application by a clear division of functionnalities in different components. Recent work on component models have shown a simultaneous answer to performance, maintainability and



(e)



(f)



(g)

Figure 8: Big example

portability of application [1] which makes component models a serious and interesting candidate for maintainable modern high performance computing.

Proposing a DSL which is dumped to a component assembly has many advantages:

- as for any DSL, the user manipulates a high-abstraction level language;
- advantages of component models are inherited for the DSL itself. Thus, the DSL is easier to maintain, scalable, portable and still efficient;
- the DSL conception phase is improved because of code re-use and productivity;
- the DSL conception phase is facilitated by the separation of concerns;
- the DSL is light as its compilation does not generate codes but a light component assembly.

Component-based software engineering [5] extends the concept of class by specifying in its interfaces not only the services it offers, or public methods, but also all its possible interactions with outer world, including the services it requires. As a result, each component is an independant entity composed of a set of services its provides, and a set of services it requires (and uses).

A *port* is an entity embbedded in the component which makes possible an *assembly* of components. An assembly of components (of instanciated components) is a way to actually connect components together and to produce a complete application, as a set of components and their interactions. An interface of provided services of a component is associated to a *provide* port, while required services are associated to *use* ports. It is also possible to group more than one required interface into a single port, as a list, called a *use-multiple* port. As illustrated in Figures 10, a provide port will be represented by a white circle, a use port by a black circle and a use-multiple port by a black circle with a white m in it.

Many component models exist, each of them with its particularities. However, components, ports, interfaces and assemblies are common concepts to all component models. The rest of this section will only use those concepts to define how the series-parallel tree decomposition, described in the previous Section, can be dumped to a component assembly, whatever the component model is.

5.1 Control-components

As already explained, our aim is to dump the series-parallel tree decomposition of Γ_{hybrid} to a component assembly. As a result, specific components have to be proposed such that $\mathcal{S}$ and $\mathcal{P}$ nodes of the decomposition can be associated to components. For this reason, we propose three specific *control-components*. Those components are called control-components because they are used in a different way than usual components. Actually, as explained before, a component represents a precise functionnality of the application, while control-components represent a way to control the execution of components. Each control-component contains a single provide port, this port is associated to a single method which corresponds to the execution of the control-component.

Sequence component. A sequence component, denoted SEQ, is a control-component for which the provided service is to garantee that the required services (provided by other components) are used one after the other in a precise order. Thus, the sequence component apply synchronous and ordered calls to required services. To do so, an ordered use-multiple port is needed.

Synchronized-sequence component. A synchronized-sequence component, denoted $SSEQ$, is a control-component for which the provided service is to garantee that a set of data synchronizations is performed before a sequence of other required services. A $SSEQ$ component is thus composed of two use-multiple ports, one not necessarily ordered to link the $SSEQ$ to data to synchronize (could be more than one), and the second, ordered, to connect $SSEQ$ to other services, as SEQ does.

Parallel component. A parallel component, denoted PAR, is a control-component for which the provided service is to call simultaneously (asynchronously) a set of required services, and to wait the end of all asynchronous calls before terminating. Thus, PAR is composed of a single use-multiple port which does not need to be ordered.

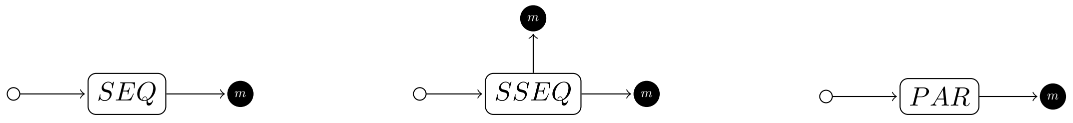


Figure 9: SEQ, $SSEQ$ and PAR control-components.

To be able to dump the series-parallel tree decomposition of Γ_{hybrid} to a component assembly, each computation c_i has to be associated to a component. Such a component contains a single provide port for which the service is to perform the computation. As a computation $c_i(R_i, w_i, \exp_i)$ manipulates data R and w, the component needs a use-multiple port to be linked to data.

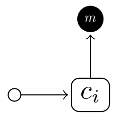


Figure 10: Computation component

Every use-multiple port of a control-component can be linked in the assembly to another control-component or to a computation component.

5.2 Canonical series-parallel tree decomposition

In the series-parallel binary tree decomposition of Γ_{hybrid}, $\mathcal{S}$ and $\mathcal{P}$ are binary operators. On the other hand, definitions of SEQ, $SSEQ$ and PAR are not binary (use-multiple), mainly because of HPC concerns. Actually, as less as the number of calls is, the better the performance is. In the work of Finta & Al [3], a canonical form of the series-parallel binary tree decomposition has been introduced. The canonical form consists in merging two identical and successive compositions ($\mathcal{S}$ or $\mathcal{P}$) as a single one. In the case of Γ_{hybrid}, however, a particular case appear when $\mathcal{S}$ is the source node of a leaf of updates $*_i$. In this case, it is forbidden to merge two sequence compositions $\mathcal{S}$. The canonical form of the final binary tree decomposition of Figure 8, is represented in Figure 11.

From this canonical representation of the multi-stencil program $\mathcal{MSP}$, a direct dump to a component assembly is possible following the transformation rules of Figure 12.

$$\begin{tikzcd}
 & & & & & & & & *_4 & & c_4 & & & & & \\
 & & & & c_3 & & c_5 & c_2 & & \mathcal{S} \arrow[ul] \arrow[ur] & & c_6 & & & & \\
 & *_1 & & c_1 & & & & & & & & & & *_8 & & c_8 \\
 & & & & & \mathcal{S} \arrow[uul] \arrow[uur] & & & & \mathcal{S} \arrow[uull] \arrow[uu] \arrow[uurr] & & & & & & \\
c_0 & & \mathcal{S} \arrow[uul] \arrow[uur] & & & & & \mathcal{P} \arrow[ull] \arrow[urr] & & & & & c_7 & & \mathcal{S} \arrow[uul] \arrow[uur] & \\
 & & & & & & & \mathcal{S} \arrow[ulllllll] \arrow[ulllll] \arrow[u] \arrow[urrrrr] \arrow[urrrrrrr] & & & & & & & &
\end{tikzcd}$$

Figure 11: Canonical form of the example of Figure 8.

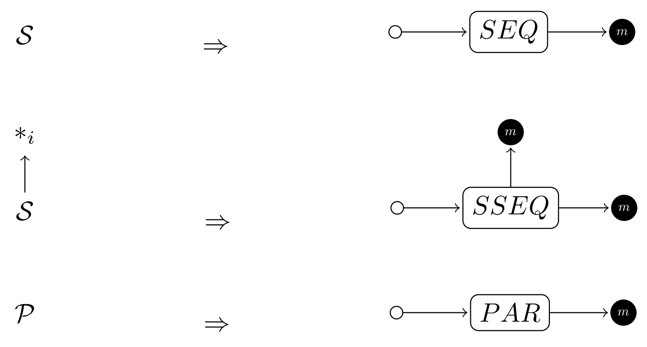


Figure 12: Direct dump rules from the canonical form to the component assembly.

For example, the canonical decomposition tree of Figure 11 is dumped into the component assembly of Figure 13.

5.3 Complete component assembly

At this point of the work, the computation part of the parallel $\mathcal{MSP}$ program is solved and dumped to a component assembly. Of course, this component assembly does not represent the entire $\mathcal{MSP}$ program. Actually, additionnal services are needed. For example, computations are performed on quantities, or data, thus those data have to be accessible somewhere through services of components in the assembly. Those quantities are mapped onto a distributed mesh, which also means that the mesh has to be accessible by the data. In addition to this, and as explained in Section 3.1, the data parallelization needs the management of specific updated-computations c_i^*, which means that the complete assembly have to manage an update service, as well as the partitioning of the mesh and the quantities mapped onto it.

However, there is not a unique way to design the rest of the assembly and to make accessible the different services (data, update, mesh etc.). The work presented until now can be considered as a generic solution to solve the parallel computation part, but the rest of the assembly is dependant of the implementation choice. However, the implementation presented in this work, and detailed in the next Section, is inspired by the SIPSim model [] which is an interesting

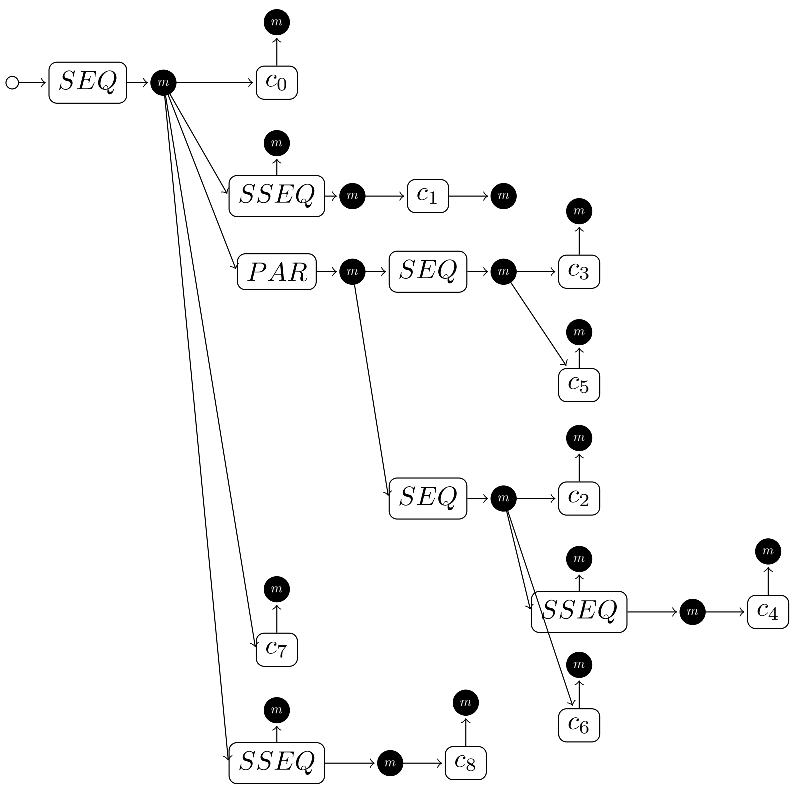


Figure 13: Direct dump rules from the canonical form to the component assembly.

implicit parallelism model to dump onto component models.

6 Component Stencil Language

6.1 Component assembly

6.2 Language

6.3 Compiler

7 Evaluation

8 Related work

1 stencil compilation: Pochoir, PATUS etc.
stencil programs: Lizst, OP2
control in components: X-MAN (The New Component Model), STCM
DSL to component : X

9 Conclusion

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.

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

ISSN 0249-6399