Component-based implicit parallelism model for multi-stencil numerical simulations

Héléne Coullon, Christian Perez

RESEARCH REPORT
N° 7003
April 2015
Project-Teams Avalon

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

Component-based implicit parallelism model for multi-stencil numerical simulations

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

Project-Teams Avalon

Research Report n° 7003 — April 2015 — 16 pages

Abstract: abstract

Key-words: No keywords

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

Exemple de document utilisant le style rapport de recherche Inria

Résumé : Resume en francais

Mots-clés : mots-cles

Contents

1 Introduction 3

2 Stencil programs 3
2.1 Definitions 4
2.2 Stencil program parallelization 5
2.3 Dependencies and synchronizations 6

3 Component-based parallelism 6
3.1 Component models 6
3.2 Stencil program parallelization using components 7

4 Component Stencil Model 12
4.1 Phase, group and kernel 12
4.2 Overall assembly 14

5 Component Stencil Language 16
5.1 CSL language 16
5.2 Performance evaluation 16

6 Related work 16

7 Conclusion 16

1 Introduction

2 Stencil programs

To numerically solve a set of PDEs, iterative methods are frequently used to approximate the solution more easily by a discretization of the time and the space (which is called a mesh). Thus, the PDEs are transformed to a set of numerical computations applied at each time step and on all elements of the mesh. Among the numerical computations is found a set of numerical schemes, also called stencil computations, and a set of auxiliary numerical computations also needed to perform the simulation. In this paper is distinguished the concept of stencil shape (often called a stencil) from the concept of stencil computation. A stencil computation involves values of a simulated quantity (pressure, speed etc.) on a given element of the mesh and also on a given neighborhood of the element, denoted $\mathcal{N}$. A stencil shape is the shape of the needed neighborhood $\mathcal{N}$ in a stencil computation. As a result, a mesh-based numerical simulation is composed of a certain number of stencil computations, which is called in this paper *multi-stencils*, involving one or more stencil shapes. In applied mathematics, an explicit numerical scheme is typically a stencil computation, but, on the other hand, an implicit numerical scheme also requires the resolution of a linear system at each time iteration. This last kind of scheme is above the scope of this work, however explicit schemes already represent a huge part of PDE solvers. In this section is defined the formalism of a *stencil program*, a description of how to parallelize such programs and the associated definitions of a *dependency* and a *synchronization*.

2.1 Definitions

A stencil program or application is a program in which stencil expressions (or computations) are computed among other kind of numerical expressions, which do not involve neighborhood information. If the space domain is denoted Ω, the time domain τ, a stencil program is denoted as

$$\mathcal{P}(\mathcal{M}, \Delta, \Gamma, \mathcal{T}), \tag{1}$$

where $\mathcal{M}$ is a mesh which discretizes Ω, Δ is the set of quantities and auxiliary data mapped onto $\mathcal{M}$, Γ is the ordered set of numerical computations in the program, and $\mathcal{T}$ is the discretization of the time domain τ. $\mathcal{M}$ can be considered as a set of different sets of elements $\{E_0, ..., E_n\}$ and a connectivity between them. For example, $\mathcal{M}$ could be composed of E_0 the set of cells, E_1 the set of nodes and E_2 the set of edges, such that each cell is composed of four edges, and each edge is composed of a source and a destination node. A stencil program is composed of a set of quantities to simulate (pressure, speed etc.), mapped on the mesh, which is denoted $\Delta(E_k), 0 \leq k \leq n$. For all $t \in \mathcal{T}$ the set of numerical computations are applied on the data, which is simply denoted by $\Delta = \Gamma(\Delta)$.In a mathematical numerical expression, a quantity to simulate has d dimensions for the space domain and one additional dimension for the time, for example $\mathcal{A}(x, y, z, t)$. However, the implementation of a quantity in Δ to a data structure δ does not represent the time dimension in the rest of this paper. Thus, for a given quantity $\mathcal{A}$ at t and $t - 1$, eventually two different data structures (arrays) δ, δ' are used.

A numerical computation in Γ is denoted

$$c(R, w, D, e), \tag{2}$$

where $R \subset \Delta, w \in \Delta$ are respectively the set of data read and written during the numerical expression e (a single data is written in a single computation), and D is one of the subsets $E_i \subset \mathcal{M}$. It has to be noticed that during a numerical simulation, at each time iteration, all the elements of a mesh are computed. However, it happens that the computation of the mesh elements is splitted in different computations (for example the computation of the physical border). In this case additional E_i can be specified for the mesh $\mathcal{M}$. In a computation c, $\forall d \in D$, the numerical expression $e(d)$ is applied. If the number of computations in a stencil program $\mathcal{P}(\mathcal{M}, \Delta, \Gamma, \mathcal{T})$ is $card(\Gamma) = m$, such that $\bigcup_{i=0}^{m-1} c_i = \Gamma$, then $\bigcup_{i=0}^{m-1} R_i \cup w_i = \Delta$.

Two different types of computations can be handled by a stencil program: stencil computations or numerical computations which do not involve neighborhood information (as *map* or *zip* in functionnal languages). The needed neighborhood $\mathcal{N}$ of a stencil computation is the shape of the stencil. For regular meshes (as for example in finite difference methods), a stencil shape is not difficult to express, however, in the general purpose, the neighborhood is specific to the kind of mesh it is applied on, and could be difficult to express. For example, the neighborhood for unstructured meshes is composed of different concepts which are explained by the operators of Laszlo and Dobkin []. This paper gives a general definition of a stencil computation, with a generic vision of a neighborhood $\mathcal{N}$. Actually, the solution presented in this paper aims to handle any kind of mesh. A stencil computation is denoted by the general notation

$$s(R, w, D, e, \mathcal{N}), \tag{3}$$

where $R \subset \Delta, w \in \Delta$ are read and written in the numerical expression e, which uses a given neighborhood function denoted $\mathcal{N}$ on a sub-domain D. In a stencil computation s, $\forall d \in D$, the numerical expression is applied such that $w(d) = e(R(d), R(\mathcal{N}(d)))$. Finally, in this work, a stencil computation $s(R, w, D, e, \mathcal{N})$ always verifies $R \cap w = \emptyset$, otherwize an implicit numerical scheme has to be solve (a relation between a a quantity $\mathcal{A}(t)$ and $\mathcal{A}(t)$ itself at the same time iteration).

The second kind of numerical computation is denoted

$$l(R, w, D, e). \tag{4}$$

This is a local numerical expression, where e does not involve $\mathcal{N}(d)$, and where e is applied such that $w(d) = e(R(d))$.

2.2 Stencil program parallelization

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.

Maybe the more important parallelization technique of such simulations is called *SPMD* (Single Program, Multiple Data) which can also be called *coarse-grain data parallelism.* This concept represents the well-known and broadly-used domain decomposition technique to split the mesh in balanced sub-meshes for available resources. Because of the good locality degree of numerical simulations, only a small number of additional elements, called a *recorvery* or a *ghost* area, are needed to exchange information computed by another resource to correctly compute stencils. As the amount of communications of such parallelization are rather small compared to the amount of computations (small neighborhood), and also by the addition of recovery of communications by computations, this parallelization technique has proved many times its efficiency and scalability on numerical simulations []. This parallelization technique can be applied on shared and distributed memory architectures (clusters, multi-core CPUs), and it stays the most efficient parallelization technique if the numerical simulation has to be executed on a distributed memory architecture. For this reason, this parallelization stays a reference in the domain and is still studied for more complex simulations, using unstructured meshes, multi-grids techniques etc.

However, because of energy consumption constraints, most of the *top500* [1] supercomputer centers are not only composed of clusters of multi-core machines, but also of accelerator cards such as GPGPU and many-cores. In addition to this, vectorization units are also available on most CPUs. This introduces another kind of parallelization, linked to the first one, called in this paper *fine-grain data parallelism*, which is composed of *SIMD* (Single Instruction, Multiple Data) and *SIMT* (Single Instruction, Multiple Threads) parallelization techniques. Such techniques typically compute a numerical expression in parallel on a vector, using vectorization units or threads, which perfectly matches the type of computation in the nested loops of numerical simulations.

Finally, the last parallelization technique which can be applied on various architectures (clusters, multi-cores, many-cores, GPGPU), is the *task parallelism* technique. Instead of spliting the data of the simulation (or the vector used in a numerical expression), the different tasks of the simulation are identified and scheduled over the available resources during the execution (at runtime) []. As numerical computations of a simulation can be considered as tasks, linked together through data dependencies for example, this parallelization technique can be used for HPC numerical simulations too.

This section discussed the various parallelization techniques which can be used for numerical simulations. As a result, it seems that, by a combination of existing techniques, modern heterogeneous parallel architectures can be fully harnessed by such applications []. In this paper is presented a component-based model to expose those three kinds of parallelism in the overall simulation, while hiding them from the user (numericians).

[1] http://www.top500.org

2.3 Dependencies and synchronizations

To define and explain the component-based implicit parallelism model presented in this paper, two additional definitions handle the relations between data read and written into computations. Those relations are responsible for dependencies and synchronizations.

Definition 1 *For two computations $c_1(R_1, w_1, D_1, e_1)$ and $c_2(R_2, w_2, D_2, e_2)$, it is said that c_2 is dependant from c_1, denoted $c_1 < c_2$, if $w_1 \cap R_2 \neq \emptyset$. In this case, c_1 has to be computed before c_2. The binary relation $c_1 < c_2$ represents a dependency.*

Dependencies lead to one computation before another, but when the program is parallelized some dependencies are also responsible for synchronizations between different processes. The opposite definition can also be given,

Definition 2 *For two computations $c_1(R_1, w_1, D_1, e_1)$ and $c_2(R_2, w_2, D_2, e_2)$, it is said that c_2 is independant from c_1, denoted $c_1 \not< c_2$ or $c_1 \parallel c_2$, if $w_1 \cap R_2 = \emptyset$.*

Considering a coarse-grain data parallelized stencil program $\mathcal{P}$,

Definition 3 *A depedency between two computations $c_1 < c_2$ is a synchronisation, denoted $c_1 \ll c_2$, if $c_2 = s_2(R_2, w_2, D_2, e_2)$, and $w_1 \cap R_2 \neq \emptyset$.*

Actually, in a parallel stencil program, synchronizations between processes are always needed for the set of data R of a stencil $s(R, w, D, e)$ because the expression e involves a neighborhood $\mathcal{N}$ which could be handled by another processor or resource.

3 Component-based parallelism

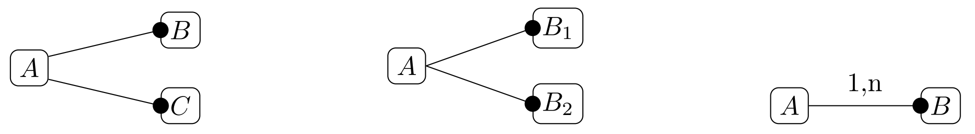
3.1 Component models

A *component* is an independent object which represents a functionnality of an application, and which is associated to a set of *interfaces* used to plug an instance of a component with other ones. The object composed of multiple component instances linked together through their interfaces is called a component assembly. As a result, a component assembly is the code of an entire application represented as a set of splitted functionnalities. Such software engineering model is known to increase *code re-use* and *productivity*, as one component can be used in different applications, but also *maintainability* and *scalability*, as functionnalities of an application are clearly splitted in different and independent components. More recently, component models have been studied to write HPC applications such as L^2C [] (Low Level Component Model) and CCA [] (Common Component Architecture).

In the rest of this paper, the component model L^2C is considered and used by the solution. A component is defined as

$$C(I, F), \tag{5}$$

where I is a set of available interfaces for the given component, and F is the set of functions to answer the functionnality of the component. A component instance, as for an object, is the actual creation of the component, and more than one instance can be created. The components of L^2C are limited to a few interfaces without the introduction of overheads. In this paper we are interested in four types of interfaces. The three first one are *use* and *provide* interfaces



(a) A uses B and A uses C

(b) A uses multiple B instances

(c) A uses multiple B instances (condensed notation)

Figure 1: L^2C use-provide and use-multiple interfaces

(Figure 1(a)) to use a functionnality provided by another component instance, and *use-multiple* interface (Figure 1(b)) to use a vector of functionnalities provided by a set of component instances. A condensed notation of a use-multiple is given in the Figure 1(c).

An instance i of a component has the same properties than the component it is instanciated from, but it is also asociated to a resource r (a core, a thread etc.) of the set of all available resources R.

$$C_i(I, F, r) \tag{6}$$

A component assembly represents the application as a set of component instances and the composition of those instances through their interfaces. A component assembly is defined as

$$\alpha(\mathcal{C}, L, R), \tag{7}$$

where $\mathcal{C}$ is the set of component instances used, L is the set of links between interfaces of component instances, and R is the set of available resources. The component assembly of L^2C offers a way to write an assembly for each MPI process, and to connect some of their components by MPI communicators. This connection is an *MPI interface*. In the rest of this paper we consider a broader interface called *synchronization*, which corresponds to the MPI interface of L^2C but which could also be a *thread synchronization* instead.

Using those four interfaces (use, use-multiple, provide and synchronization), it is possible to write an application with medium-grain components, enough fine to keep efficiency and to stay close to the execution model, and enough coarse to take advantage of components. The efficiency of L^2C has been proved, while increasing the productivity and maintainability of HPC applications, but moreover it has also been shown that the protability of such component-based HPC application is improved [].

3.2 Stencil program parallelization using components

In this section is studied the parallelization of a numerical simulation using the given definitions of a component, an assembly and four interfaces. First, the different parallelization techniques explained in the Section 2.2 are translated to component-based parallel assemblies, and an approximation of the general case is discussed for the solution presented in this paper. In the rest of this section, the following example of dependencies is used

$$A < ((B < D) \parallel C) < E \ll A' < ((B' < D') \parallel C'), \tag{8}$$

where $A, B, C, D, E, A', B', C', D'$ are numerical computations. This example, can be represented as a directed acyclic graph as in the Figure 2.

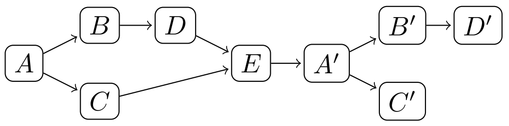


Figure 2: Dependencies example (8) represented as a DAG.

Coarse- and fine-grain data parallelism. As explained in the Section 2.2, to apply coarse-grain data parallelism on numerical simulations, a domain decomposition (or graph partitioning) has to be done on the mesh and also on the data mapped on it. When this important step is solved, the same program can be applied on each subpart of the data, on the different resources. In addition to this, a set of synchronizations are needed between the resources at each time step to compute the neighborhood $\mathcal{N}$ correctly. When using a component model, the computing part of the application, which is called for each time step, can be seen as a component assembly which is duplicated on different resources, which is possible using L^2C for example. However, it is needed to classify computations in the component assembly to identify when synchronizations are needed (denoted $\ll$ in the Section 2.3). For this reason, a *sequence* component SEQ, which uses one after the other a set of computation components, is defined as

$$SEQ(\{provide, use - multiple\}, \{sequence\}). \tag{9}$$

Thus, the *sequence* function (represented in the Algorithm (1)) is responsible for the loop which calls as much *uses* as the number of components connected to the *use-multiple.* In addition to this, a *synchronized sequence* component $SSEQ$, which uses one after the other a set of computation components, but which also proceeds synchronizations between them, is defined as

$$SSEQ(\{provide, use - multiple, synchronization\}, \{ssequence\}). \tag{10}$$

In this case, the *ssequence* function (represented in the Algorithm (2)) is responsible for a loop of two steps, first a synchronization step, second a use step. Finally, each numerical computation is represented by a component

$$K(\{provide\}, \{compute\}), \tag{11}$$

where the function *compute* (represented in the Algorithm (3)) is responsible for the sapce loop and the numerical computation on each element of the mesh.

Algorithm 1: sequence function
```
forall the component cp to use do
    use the provide interface of cp
```

Algorithm 2: ssequence function
```
forall the component cp to use do
    synchronizations
    use the provide interface of cp
```

Algorithm 3: compute function
```
forall the elements d in D do
    $w(d) = e(R(d), R(\mathcal{N}(d)))$
```

Using the three components $SSEQ$, SEQ, and K, the component assembly of the dependencies (8) is represented in the Figure 3. One can notice, in this Figure, that the parallel dependencies ($||$) are lost. However, in a SPMD parallelization, the parallelism is obtained from executing the same program on different part of the data. As a result, a correct sequence of computations is sufficient to expose coarse-grain data parallelism. The overall computation assembly of an

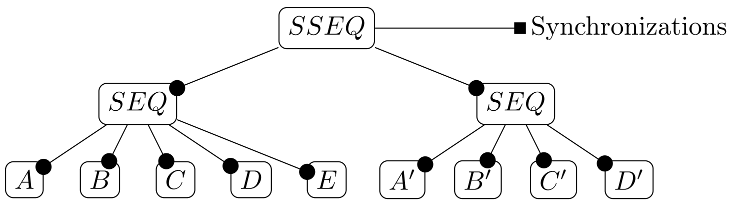


Figure 3: Example assembly for coarse- and fine-grain data parallelism

SPMD numerical simulation is composed of a duplication of the computation (or dependencies) assembly described above. Each duplication is applied on an available resource as illustrated in the Figure 4. In the rest of this paper, this overall computation assembly of an SPMD application (coarse-grain data parallelism) is represented as a single computation assembly, and the *SSEQ* component indicate where the synchronizations are needed.

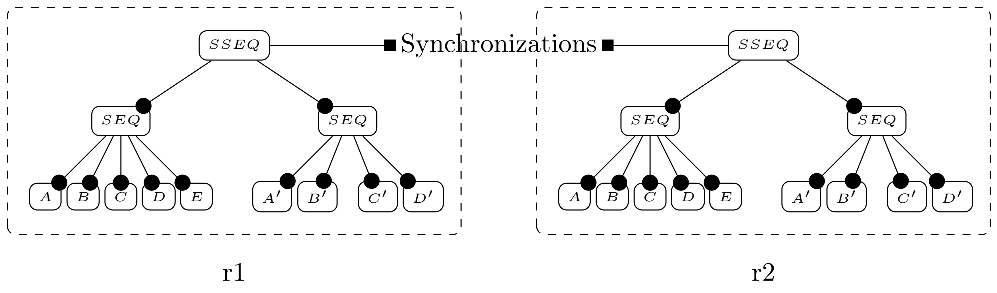


Figure 4: Overall SPMD computation assembly of the example (8) on two resources.

But the assembly obtained from the instanciation of *SEQ*, *SSEQ*, and *K* does not only expose coarse-grain data parallelism. Actually, the generic notation of a computation component *K*, which represents a numerical computation, is important to identifiy the different computations of a numerical simulation as different functionnalities of the program, but in addition to this, exposing a computation as a component, also exposes the fine-grain data parallelism level. In fact, a numerical computation is nothing more than a loop (or a nested loop) on the mesh elements and a numerical expression to compute (Algorithm (3)). As a result a computation component is ideal to introduce SIMD and SIMT parallelism.

The general case of assembly to introduce coarse- and fine-grain data parallelism in numerical simulations is represented in the Figure 5.

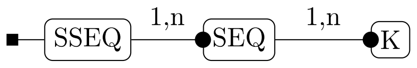


Figure 5: General assembly description for coarse- and fine-grain data parallelism

Data and task parallelism As also explained in the Section 2.2, to apply task parallelism to numerical simulations, it is needed to identify the different tasks of the application and their dependencies. With the introduction of $SSEQ$, SEQ, and K components, a part of this work is already done. Actually, each computation component K represents a task. However, to be able to express all dependencies between computations, the PAR component is defined as

$$PAR(\{provide, use - multiple\}, \{parallel\}), \tag{12}$$

and corresponds to the expression $\|$ or $\not\prec$. The function *parallel* creates a thread for each component to use and join all threads at the end.

Algorithm 4: parallel function

```
forall the component cp to use do
    create thread t t uses the provide interface of cp
join all threads
```

In addition to this, a composition of the components $SSEQ$, SEQ, PAR and K is needed to be able to represent complex dependencies. For example, the example dependencies (8) is represented in the Figure 6.

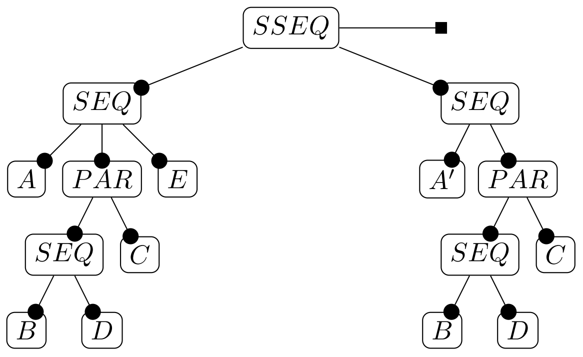


Figure 6: Example assembly for coarse- and fine-grain data parallelism

The general case of assembly to introduce data parallelism and task parallelism in numerical simulations is represented in the Figure 7.

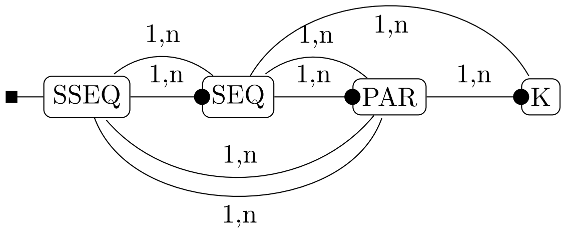


Figure 7: General assembly description for data and task parallelism

Intermediate expressivity The component assembly defined in the Figure 7 is usefull to expose data parallelism, task parallelism, and to have a complete expressivity for dependencies between computations. For the general case of scientific applications, this assembly is needed, however, this assembly has a certain number of drawbacks. First, $SSEQ$, SEQ and PAR do not represent functionnalities of the application but control of functionnalities. As a result, the role of a component is twisted. Second, in the case of a complex application, with complex dependencies, the component assembly is going to be riddled with $SSEQ$, SEQ and PAR components, making difficult the reading of the application. Thus, it decreases the maintainability bring by component models. Finally, this expressivity is as complex as a *task graph* [], and as a result, two additional difficulties get out if such an assembly is used in our solution: the automatic detection of complex dependencies from a simple description of the application, and the *scheduling* of such tasks. This last point is particularly critical and difficult. Because of the compositions of $SSEQ$, SEQ and PAR components illustrated in the Figure 7, and because of the definition of PAR which simply creates a thread for each component to use, it is possible that the work of each thread becomes unbalanced. For example, the Figure 8 represents an unbalanced example.

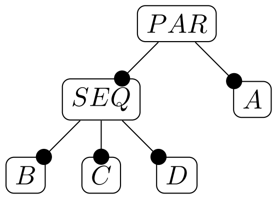


Figure 8: Unbalanced assembly example

However, in existing component models for HPC (for exemple in L^2C) only the component object is defined, and all components are created, destroyed, and called the same way. Thus, it is not possible to manage a PAR component to dynamically balance the workload on threads. For this reason, it seems that the combination of tasks and components are needed as well as a scheduler. The scheduling of a task graph is an active research domain and it could be an interesting perspective to work on the combination of L^2C and StarPU [], for example. However, this work aims to propose a light solution using component contributions, and to illustrate the advantages and drawbacks of such a solution. To only use components in our solution, and to avoid unbalanced and heavy assemblies, we propose an intermediate assembly which is an approximation of the general case of the Figure 7. We claim, in the rest of this paper, that this approximation is enough expressive and produces efficient applications for the specific case of mesh-based numerical simulations.

The proposed intermediate assembly is represented in the Figure 9. The same basic association of components $SSEQ$, SEQ, PAR and K is found, however a limitation of the possible compositions of those components is applied. It is no longer possible to use a SEQ or $SSEQ$ from a PAR component, as well as using a $SSEQ$ from a SEQ component and vice versa. The most important part of those limitations is to avoid too much imbalance. Actually, from a PAR component it is only possible to use one computation K in each created thread. In the specific case of mesh-based numerical simulations, as most numerical computations $c(R, w, D, e)$ (except boundary conditions) are applied on a complete set of mesh elements D and computes a numerical expression e, two numerical computations are approximately balanced, or at least the imbalance is limited.

Using this approximated assembly, the assembly of the example is given in the Figure 10. The limitation of such an approximation is clearer in this example: the dependencies $(B < D) \parallel C$

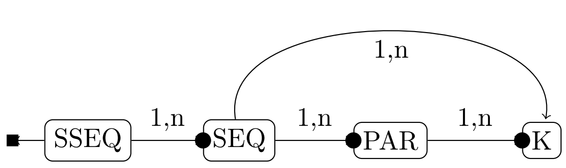


Figure 9: General assembly description approximation

and $(B' < D') \parallel C'$ can only be expressed by $B \parallel C < D$ which is a less precised dependency. But as explained before, this limitation of expressivity also guarantees a limitation of the imbalance which is due to the use of components only, and not task graphs and schedulers. Finally, the proposed approximation still exposes the three level of parallelism introduced in the Section **??**.

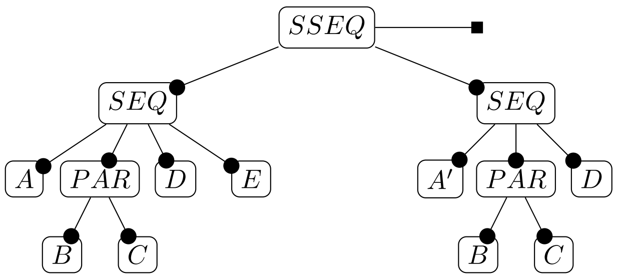


Figure 10: Example assembly approximation

4 Component Stencil Model

4.1 Phase, group and kernel

Using the formalism described in Sections 2 and 3, the following definitions are given for the solution.

Definition 4 *For a stencil program $\mathcal{P}(\mathcal{M}, \Delta, \Gamma, \mathcal{T})$, a* phase $\Phi \subset \Gamma$ *is a subset of ordered computations $\{c_0, ..., c_{n-1}\} \in \Gamma$ such that $\forall 0 \leq i, j \leq n-1, \nexists c_i, c_j \in \Phi$, which verifies $c_i \ll c_j$.*

One can notice, however, that in a phase Phi of a stencil program it is possible to have a pair $c_i, c_j \in \Phi$, which verifies $c_i < c_j$. As a result, a phase is equivalent to a sequence of computations SEQ. The set of phases of a stencil program are ordered and a synchronization is needed between two different *phases* of a stencil program $\mathcal{P}$. In other terms for two phases Φ_1 and Φ_2 $\exists c_i \in \Phi_1, c_j \in \Phi_2$ such that $c_i \ll c_j$. If m is the number of stencil computations in a phase Φ_i among n total computations, the synchronizations of Φ_i are applied on $\bigcup_{j=0}^{m-1} R_j$.

Instead of introducing an additional definition for the equivalent of $SSEQ$, and because $sequence(sequence(a, b), sequence(c, d)) = sequence(a, b, c, d)$, synchronizations needed by a new phase are commputed inside the Phase itself before computations. A phase component is defined as

$$Phase(\{provide, use - multiple, synchronization\}, \{phase\}). \quad (13)$$

The function *phase* performs the needed synchronizations before to use each component of the sequence

Algorithm 5: phase function

```
synchronizations
forall the component cp to use do
    use the provide interface of cp
```

Definition 5 *For a stencil program $\mathcal{P}(\mathcal{M}, \Delta, \Gamma, \mathcal{T})$, a* group $\mathcal{G} \subset \Gamma$ *is a subset of unordered computations $\{c_0, ..., c_{n-1}\}$ such that $\forall 0 \leq i, j \leq n-1, \nexists c_i, c_j \in \mathcal{G}$, which verifies $c_i < c_j$.*

Thus, a dependency exists between two different *groups* of a stencil program $\mathcal{P}$, but the computations inside a *group* are unordered, without dependencies. As a result, the Group component is defined as PAR

$$Group(\{provide, use-multiple\}, \{group\}), \tag{14}$$

where the function *group* creates a thread for each component to use, and join all threads at the end.

Algorithm 6: group function

```
forall the component cp to use do
    create thread t t uses the provide interface of cp
join all threads
```

Definition 6 *For a stencil program $\mathcal{P}(\mathcal{M}, \Delta, \Gamma, \mathcal{T})$, a* kernel $\mathcal{K} \subset \Gamma$ *is a subset of unordered computations $\{c_0, ..., c_{n-1}\}$ such that $\forall 0 \leq i, j \leq n-1, \nexists c_i, c_j \in \mathcal{K}$, which verifies $c_i < c_j$, and where $\forall i, j < n, D_i = D_j$.*

Thus, a *kernel* could contains n computations $\{c_0, ..., c_{n-1}\}$, but handles a single space loop in a coarser computation c' such that $c'(\bigcup_{i=0}^{n-1} R_i, \bigcup_{i=0}^{n-1} w_i, D, \{e_0, ..., e_{n-1}\})$. This definition differs from the component K described in this section as a kernel could contains more than one computation if the space domain is the same. This definition of a kernel improves performances as the memory bandwidth is better used. A kernel component if defined as

$$Kernel(\{provide\}, \{kernel\}). \tag{15}$$

The function *kernel* performs the set of numerical computations

Algorithm 7: kernel function

```
forall the elements d in D do
    $w(d) = e_1(R(d), R(\mathcal{N}(d)))$
    $w(d) = e_2(R(d), R(\mathcal{N}(d)))$
    ...
```

The component assembly of the computations of a mesh-based numerical simulation using phase, group and kernel components is represented in the Figure 11. This assembly is equivalent to the one presented in the Figure 9, but $SSEQ$ and SEQ are fusionned in *phase*, and the component *kernel* is bit different from K.

This component assembly of computations can be, by definition, extracted from an ordered list of typed computations $s(R, w, D, e, \mathcal{N})$ or $l(R, w, D, e)$. R, w and D are used to build Phases, Groups and Kernels, while e and $\mathcal{N}$ are used inside the Kernel components. As a result, we can define a function λ to build the computation assembly

$$\lambda : list_o(c(R, w, D, e)) \rightarrow \alpha(Phase, Group, Kernel, L, R) \tag{16}$$

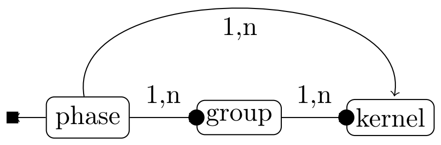


Figure 11: General assembly proposed in our solution

4.2 Overall assembly

The SIPSim model. The SIPSim model [] is a model which proposes a systematic way to create an SPMD implicit parallelism solution for mesh-based numerical simulations through four concepts. The first concept is a *Distributed Data Structure* (DDS), which represents the distributed mesh. Thus, the DDS manages a domain decomposition, or more broadly a graph partitioning problem which balances the number of elements on available resources, and which tries to minimize the number of needed synchronizations between resources. It has to be noticed that this particular task could be managed by an external graph partitionner. In addition to this, the DDS also has to offer an efficient access to an element of a mesh and to its neighborhood. The second concept is the *Distributed Property Map* (DPMap), which is responsible to map a quantity to simulate (a data) onto the distributed mesh. The third concept is an *applicator*, which is responsible for applying a set of synchronizations before a set of numerical computations written by the user (also called operations). Finally, the last concept contains the *interfaces* needed to write an operation in a sequential programming style while using DPMaps mapped onto the DDS. Those interfaces are directly linked to the type of DDS used.

CSM overall assembly. From the SIPSim model, CSM uses the concept of DDS and DPMap and transforms them in components to build the overall component assembly. The implementation of those components are not described in this paper, and it is possible to imagine an adhoc implementation in simple cases, as a Cartesian mesh, or existing external implementations (libraries) in more complex cases. This choice is an implementation choice and will be compatible with CSM. In addition to those two new components, the concept of *interface* of the SIPSim model has to be kept to write the implementation of the expressions e of the numerical computations on distributed data structures. But this concept is not needed as a component. It could be provided by the DDS itself or it could be introduced as a header file for example.

The overall assembly produced by the model is presented in the Figure 12. First, the component *DDS* and the component *Data*, which corresponds to the DPMap concept of the SIPSim model, are introduced in the assembly with the same functionnality than described above.

Second, one can notice that the assembly described in the previous Figure 11 is, of course, inserted in the overall assembly. However, even if the *Phase* component is still responsible for starting a set of synchronizations, those synchronizations are in fact performed by the *Connector* component. Actually, the *Phase* component uses the data concerned by its synchronizations, and then each data uses the *Connector*, which finally performs the synchronizations. As a result, the *Connector* component can be implemented with threads, or with MPI, without the modification of *Phase* and *Data* components. This is a separation of concerns between the data of the simulation and the parallelization strategy which increases the portability of the application. This strategy has been more described in the paper [].

The third important point to notice is the introduction of the component *Time* which is responsible for the time iteration steps of the simulation. To establish the end of time steps, most of the time a convergence function is computed on data, but it also happens that a fix number of iterations is precised. For this reason, the instanciation of the component *Convergence* is optional. The *Driver* component starts the simulation and launches the initialization of the DDS, and the *DrApp* component starts the initialization of the data and also the time loop of the simulation. Finally, the *Initializer* component is linked to the initialization of data (read a file format, default value, or numerical computation). The computation components, specific to each simulation, are *Kernel* and *Convergence*, and to perform numerical expressions e those components have to be connected to the Data of each R and w.

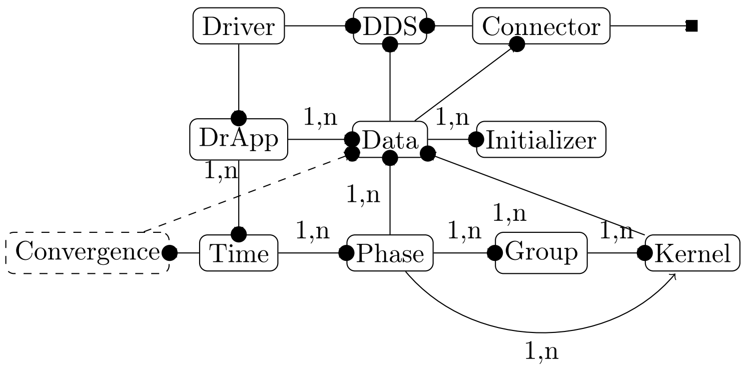


Figure 12: Overall assembly of the Component Stencil Model.

This overall assembly can be compiled from a set of sequential and descriptive information. First an unordered list of data is needed. A data is defined as

$$data(init_f, E_i, type) \tag{17}$$

, where $init_f$ is the function to initialize the data (in the Initializer component), Ei is the set of elements of the mesh on which the data is applied, and *type* is the actual type of data (*int*, *double* etc.).

The second needed information is the description of the mesh to use as

$$mesh(type, dimension, size) \tag{18}$$

, where *type* of is the topology of the mesh (for example Cartesian or k-sided unstructured meshes), *dimension* is the dimension of the mesh, and *size* is the size of the mesh.

As a result, we can define a function Λ to build the overall simulation assembly of the Figure 12 as

$$\Lambda : list_o(c(R, w, D, e)) \times list(data(init_f, E_i, type)) \times mesh(type, dimension, size) \rightarrow \alpha(\mathcal{C}, L, R) \tag{19}$$

The needed information of the function *Lambda* are typically produced by numericians who try to solve a set of partial differential equations by existing numerical methods, by a combination of methods, or by their own methods. Once the parallel assembly α is created, the engineer or a numerican developper of the lab can write the set of numerical computation components *Kernel*, the set of *Initializer* components, and eventually the *Convergence* component. All those components are written with an imperative sequential programming style using the interfaces linked to the DDS. As a result, multiple levels of separation of concerns are introduced. First the separation of concerns between the numerician and the person in charge of the implementation, and second the separation of concerns between the implementation and the parallelization of the simulation, which is generated.

5 Component Stencil Language

5.1 CSL language

5.2 Performance evaluation

6 Related work

7 Conclusion

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

ISSN 0249-6399