Combining Both a Component Model and a Task-based Model for HPC Applications: a Feasibility Study on GYSELA

Olivier Aumage, Julien Bigot, Hélène Coullon, Christian Pérez, Jérôme Richard

Combining Both a Component Model and a Task-based Model for HPC Applications: a Feasibility Study on GYSELA

Olivier Aumage*, Julien Bigot†, Hélène Coullon‡, Christian Pérez‡ and Jérôme Richard‡
* *Inria, LaBRI*
Email: olivier.aumage@inria.fr
† *Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Université Paris-Saclay*
Email: julien.bigot@cea.fr
‡ *Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP*
Emails: {helene.coullon,christian.perez,jerome.richard}@inria.fr

***Abstract*—This paper studies the feasibility of efficiently combining both a software component model and a task-based model. Task based models are known to enable efficient executions on recent HPC computing nodes while component models ease the separation of concerns of application and thus improve their modularity and adaptability. This paper describes a prototype version of the COMET programming model combining concepts of task-based and component models, and a preliminary version of the COMET runtime built on top of StarPU and L2C. Evaluations of the approach have been conducted on a real-world use-case analysis of a subpart of the production application GYSELA. Results show that the approach is feasible and that it enables easy composition of independent software codes without introducing overheads. Performance results are equivalent to those obtained with a plain OpenMP based implementation.**

***Keywords*-HPC; Software Component Model; Task-Based Model; Task scheduling; Multi-cores; Shared-memory;**

I. INTRODUCTION

High-performance architectures are now massively multi-core or even many-core. They may also feature heterogeneous technologies such as those with GPGPU accelerators in additions to the CPUs. Efficient use of these platforms requires hand-tailoring algorithms for the hardware. It requires a good understanding of both the target platform and the application itself. Moreover, codes must be continuously tweaked to adapt to new architectures. In numerical simulation domain, another important source of code variation stems from using improved mathematical schemes or taking into account new discoveries in the simulated domain. This process has a major cost in term of development time during the lifespan of the application. Without special attention to software engineering aspects, this cost can explode, for example, due to duplication of efforts, multiplication of concerns in the same code, etc. Developers therefore have to take maintainability into account when designing a code. Otherwise, the code could become obsolete, in the worse case even before it becomes feature-rich enough to be useful. Maintainability concerns do however often conflict with performance requirements. HPC code developers are therefore faced with a fundamental dilemma between maintainability and performance, and have to make a difficult choice regarding the best compromise.

A solution on the software engineering side of the dilemma is brought by component-based software engineering (CBSE) [1], [2]. CBSE proposes to build applications by assembling independent software building blocks (components) with well-defined interfaces. Components are instantiated and their interfaces connected to form an assembly. This enables easy reuse of (potentially third-party) components and architectural-level modifications of applications through their assembly. Thus CBSE highly improves application maintainability and adaptability. Component models with low overheads have been proposed for high-performance computing, such as CCA [3] or L^2C [4].

Existing HPC component models do however not solve the fundamental trade-off between maintainability and performance. Indeed, while splitting an algorithm into two independent components improves code adaptability, it typically constraints the way their execution can be interleaved and can hinder performance at runtime.

HPC task-based scheduling runtime systems [5], [6], [7], [8] have been designed to ease reaching high performance on complex hardware such as those with many-cores or GPUs (by constantly feeding processing units while hiding synchronizations and data transfers) as well as performance portability (by separating the description of the work from how it is performed on available resources). In this approach applications are described as graphs of tasks with ordering constraints, called dependencies. Different runtime systems exist proposing different scheduling policies and techniques. Task granularity should be small enough such that the runtime scheduling algorithm can leverage this flexibility to make efficient choices for the available hardware. This does however mean that tasks are not the right model for handling multiple coarse-grain algorithmic variants, since a whole subset of tasks (*i.e.,* a subgraph) may need to be modified to replace a single algorithm.

Combining both approaches appears as a compelling way

to improve the development of HPC applications. Software components ease algorithm composition and replacement, while tasks and runtime scheduling ease performance portability over a wide range of architectures. However, to our knowledge, existing HPC models do not unify component and task models in a coherent way.

This paper presents a feasibility study of COMET, an approach aiming to combine both models. COMET is composed of two elements. First, the COMET *programming model* extends existing component model concepts with a new way of interacting between components specified by dataflows. We demonstrate that this enables the separation of concerns in distinct components while retaining high-performance thanks to the COMET underlying execution model.

Second, the COMET *runtime* maps the programming model concepts to an execution model, while retaining extensibility for new concepts in the programming model. This feasibility study targets the shared-memory intra-node scale since this is the level at which fine-grain execution interleaving is critical. Inter-node interactions are left for future work. We apply and evaluate the COMET approach on a use-case from the large-scale production code GYSELA.

Section II presents the GYSELA application. Section III deals with related works. Section IV describes the COMET model and runtime and how the use-case has been implemented using it. Section V evaluates the approach both in terms of software-engineering capabilities and performance. Section VI concludes the paper and gives some perspectives.

II. THE 2D (r, θ) ADVECTION IN GYSELA

A. GYSELA *Overview*

GYSELA is an application that simulates the electrostatic branch of the ion temperature gradient turbulence in tokamak plasmas [9], [10]. The code consists of around 60,000 source lines of code, 95% Fortran and 5% C. It uses a hybrid MPI and OpenMP parallelization and has run on up to 1.8 M threads on the Juqueen Blue Gene/Q (Jülich, Germany) with 91% relative efficiency in weak scaling [11]. Two solvers are self-consistently coupled at the heart of GYSELA. A Vlasov solver computes the particles advection while a Poisson solver computes the magnetic fields.

The main data manipulated in the Vlasov solver is a 5D particle distribution function in phase space $(r, \theta, \varphi, v_{\|}, \mu)$, where (r, θ, φ) are the space coordinates and $(v_{\|}, \mu)$ the velocity coordinates. The advection part consists in moving the particles in this 5D space according to the electromagnetic field provided by the Poisson solver. In GYSELA, this is computed thanks to a semi-lagrangian scheme with a Strang splitting. As a result, the 5D problem is implemented by successive 1D and 2D advections.

This paper focuses on the 2D advection in the (r, θ) dimension that is the most computational intensive in GYSELA. This 2D-advection solves an independent problem for each (r, θ) plane of the 5D field. It takes two fields as input: a 2D plane of the 5D field corresponding to the particles density and another 2D field corresponding the electromagnetic field in (r, θ). The result (wherein the particles have been moved) is computed in-place in the particle density 5D array.

B. 2D Advection Overview

The 2D advection can be split into two distinct parts: the computation of four displacement fields from the electromagnetic field and the actual semi-lagrangian advection computing the new particle density. The computation of each displacement field is independent and can be done in parallel. The semi-lagrangian scheme then uses these displacement fields to determine for each point of the density function the origin of the displacement where the density value was the same at the previous time-step. Since this point was most likely not a grid point, an interpolation is used to evaluate the value there based on surrounding grid points. Two interpolation schemes often used in the community are splines and Lagrange polynomials. According to the chosen method, the dataflow of the algorithm differs slightly.

The actual semi-lagrangian advection is composed of three sequential sub-steps, whatever the chosen interpolation scheme. First, data buffers (such as the input buffer or the displacement buffers) are copied into larger ones with boundary values, where each cell can be computed independently (except for the boundary values of the spline interpolation). Second, field derivatives are computed, using different ways according to the chosen interpolation method. The spline interpolation first computes 2D spline coefficients as two successive 1D computations: independent computations on each line, followed by independent computations on each column of the previous sub-step results. It then uses a stencil where the computation of each output cell accesses neighbor cells in the input buffers. The Lagrange interpolation only uses a sequence of multiple stencils. Third, the actual interpolation is computed, where each output cell can be computed independently, but requires the entire two-dimensional input sub-domain due to a value-driven indirection.

The whole algorithm is applied for each 2D plane independently, providing coarse-grain parallelism (data parallelism). Finer-grained parallelism inside each 2D plane could also be used. However, this would require a task-parallel model since some kernels require the full 2D plane. As a result, this is not implemented in the reference GYSELA code and thus not evaluated here.

C. Analysis

The current state of the 2D advection code is a monolithic package wherein the two main algorithms (displacement field computation and interpolation) are tightly and manually coupled using nested functions. Tightly coupling of algorithms is required at runtime to be efficient: it enables better

temporal and data locality as well as higher computational intensity. However, coupling algorithms using nested functions is difficult due to the increasing number of concerns in the same code as well as conflicting requirements from software engineering (maintainability) and performance. Indeed, replacing or providing alternative implementations of the interpolation scheme while keeping an efficient code is hard because it requires changing low-level functions in the heart of GYSELA. This is especially so when the granularity or the access pattern of alternative algorithms differs, requiring significant changes. Since alternative algorithms may be devised to provide more accurate or efficient methods, maintaining the application while keeping high-performance is an important concern.

The 2D advection uses multiple kinds of optimizations to achieve high-performance. These are hidden deep in the heart of the code and thus difficult to tell apart from the actual semantic. They are also specialized for a specific use and guided by the best practices regarding hardware generally accepted at the time of implementation. Since hardware evolves continuously, the code has to be adapted to handle new architectures. Moreover, high-level optimizations such as the parallelization method and low-level optimization such as tiling and vectorization are mixed together in the same code. Consequently, understanding and adapting the code for a new specific use (*e.g.,* adapting the computational method, supporting new architectures) requires advanced skills from the programmer.

Based on this example, we can identify five properties a programming model should provide to improve the situation: *i)* algorithmic substitution should be made easy, it should be possible to reuse as much application code as possible, and it should not require advanced skills from the programmer; *ii)* the programming model should map to an efficient execution model leveraging available resources (with a high scalability), and avoiding unnecessary overheads; *iii)* expression of both task and data parallelism should be possible (without manually setting any arbitrary optimization choice first); *iv)* optimizations should be described separately or even be done automatically; *v)* finally, high-level architectural level optimizations (parallelization method tuning, computational method tuning, etc.) and low-level optimizations (tiling, vectorization, etc.) should be clearly separated.

III. RELATED-WORK

Amongst component models, the Common Component Architecture (CCA) [3] and the Low-Level Component (L^2C) [4] both target to improve maintainability and adaptability for HPC applications with low overhead and support for MPI. In these two models, components can expose services (*i.e.,* a set of functions) using *provide ports* and can require services provided by other components through *use ports*. An assembly is a set of component instances whose use ports are connected to provide ports, enabling the first one to calls functions of the second one. Such composition aims to significantly ease code reuse as well as code coupling. However, being able to simultaneously express task and data parallelism without hindering efficiency using these two models is not currently supported. Indeed, when an algorithm is split into multiple independent components, L^2C and CCA provide no simple way to efficiently compose, or interleave, the parallel execution of the different parts. In fact, this burden is up to application developers.

On the other hand, task-based approaches aim to enable this interleaving of algorithm execution without requiring developers to specify the exact execution order. Task-based runtime systems such as Parsec [6], Legion [12], OmpSs [7], XKaapi [13], StarPU [8] and Peppher [14] enable to efficiently execute computational parts of HPC applications. This is achieved by scheduling computation units called tasks over the available resources. Tasks usually require data as input and produce output data. In most models, task outputs are connected to task inputs using data dependencies (temporal dependencies), so as to form a directed acyclic graph (DAG). In a such DAG, the nodes are tasks and the edges represent (data) dependencies between tasks. The DAG is then scheduled by a runtime, targeting an efficient execution over a wide range of hardware. However, being runtimes, these tools only focus on execution-time matters. The software engineering aspects such as code composition or maintainability depend on the programming model or API used to describe the task-graph.

Approaches such as Regent [5], Swift [15] and OpenMP [16] offer models to describe applications task-graphs using either a dedicated language or annotations respectively. Regent [5] and Swift [15] are programming languages for implicit dataflow parallelism. A high-level task-based imperative language offers end-users a way to implicitly (*i.e.,* in a transparent way) build their task graph using constructs such as loops, conditionals, etc. Actually, function calls or specific code constructs are transparently replaced during execution by submitting tasks to a task-based runtime. While proposing high-level abstraction to end-users, low-level data-based composition can naturally be expressed using Regent thanks to two given abstractions: tasks, and logical regions (*i.e.,* collections of structured objects that can recursively be partitioned). However, high-level coupling of independent code is not handled in Regent. Moreover, the use of an imperative language to implicitly create the task DAG does not improve application structure understanding, separation of concerns into independent software parts, or ease of replacement and maintainability like component models do through the concept of assembly.

OpenMP [16] is a well-known framework that supports shared memory multiprocessing programming. It provides means to easily incorporate parallelism into sequential applications, at a relatively high-level through the use of code annotations. It supports both task and data parallelism.

While OpenMP helps to easily parallelize HPC applications, it does not change the underlying programming paradigm of the annotated language (*i.e.* imperative programming). Supporting such an approach for component models, where parallelization aspects could be specified by a dedicated language in the assembly, would be a very interesting feature. However, as of now, the authors are not aware of any such work. OpenMP can be used in C, C++ or Fortran component implementations but is not available at the assembly level. The approach proposed in this paper can be seen as a first step into this direction.

Data-driven workflows such as Gwendia [17] and dataflow-based models such as FlowVR [18], FastFlow [19], or the model proposed by Lau and al. [20], emphasize easing the composition of software codes and also provide a higher abstraction than task-based runtime systems. All of them share the ability to compose algorithms through data-based composition where components of the models expose data ports which, once connected, enable components to produce data consumed by other ones. The use of such composition is a convenient way to describe many HPC applications and could be useful to describe the 2D advection. However, these models are mainly designed for heavy-grained task-based composition. As a result, coupling HPC codes with fine-grain task parallelism using these models would be detrimental to the maintainability, because it would require codes to be split into many independent parts, reducing the cohesion of coupled codes.

Finally, the Spatio-Temporal Component Model [21] (STCM) unifies features from both component models and data-driven workflow models. The model provides composition units called component-tasks (merging tasks and components). Such units can be composed into assemblies through both use-provide and data connections. However, the model does no target fine-grain intra-node parallelism for HPC applications. Components and tasks are merged together, which would result in high overhead if used for fine-grained task decomposition of HPC applications, due to component instantiation and connection overheads.

To summarize, some existing approaches improve software engineering while others support task-based high-performance computing. However, no single model combines both advantages. Thus, end-users have to face a trade-off between maintainability and performance on modern HPC architectures. None of the proposed models are satisfactory as they do not enable the writing of independent modules efficiently coupled at runtime. The next section presents the COMET model, a proof of concept aiming to handle both component programming models and task-based high-performance runtimes.

IV. THE COMET MODEL

This section gives an overview of the COMET programming model as well as the COMET runtime. The COMET programming model is based on the L^{2}C model, since it is a minimalist HPC-oriented component model [4]. COMET extends it with a dataflow-based form of interactions between components enabling the creation of a task graph. It also enables component connections with fine-grain execution to interleave in a controlled manner, as described in the assembly. The COMET compiler and runtime map these concepts to plain L^{2}C components that make use of the StarPU task graph scheduling [8]. The COMET runtime distinguishes three types of components: a) components written by the user in the programming model, b) components generated during the compilation phase, and c) components written by experts that make the runtime easily extensible to add more flexibility in the programming model.

A. The COMET Programming Model

Let us note that, in this work, only a subset of the COMET programming model has fully been defined. The choices have been guided to support the 2D advection use-case. In particular, this paper only deals with intra-node assemblies and local C++ function calls between components.

The two main elements of the COMET programming models are *components* and *dataflow sections*. These are instantiated and connected in an assembly describing the application architecture. COMET components are L^{2}C components extended with new data ports that enable components to interact with dataflow sections.

Dataflow sections contain *metatasks* connected to *data buffers* with dataflow-based interactions. The *data* partitioning function determines data fragments that are the minimal unit used by a task. For example, the current implementation supports block partitioning of multidimensional arrays. Hence, given a 2D array $A(N, M)$, a 2D block partition of size (n, m) generates $\lceil \frac{N}{n} \rceil * \lceil \frac{M}{m} \rceil$ fragments.

A metatask represents a set of tasks that is created at runtime. They expose a list of data ports defining the memory buffers used by the tasks. Each task has dependencies over fragments of the data buffers connected to data ports. The dependencies are defined by an expression in the assembly specifying alignment between data buffers. Currently, the proposed alignment language is very simple but enough for the 2D advection use-case: only perfect alignment between arrays can be defined, *i.e.,* for two arrays A and B of the same size, $a(i, j)$ and $b(i, j)$ are guaranteed to be proceeded by the same task receiving fragments of A and B as inputs.

The task implementation is provided by a use port of the metatask that is connected to a component instance outside any dataflow section. Hence, metatasks do not contain user-level code; they only enable the implicit description of task sets; the actual implementation is delegated to components.

Four types of data ports are defined in the model, depending on whether the entity (component or metatask) exposing the port generates information through the port and whether it uses the information accessible through it.

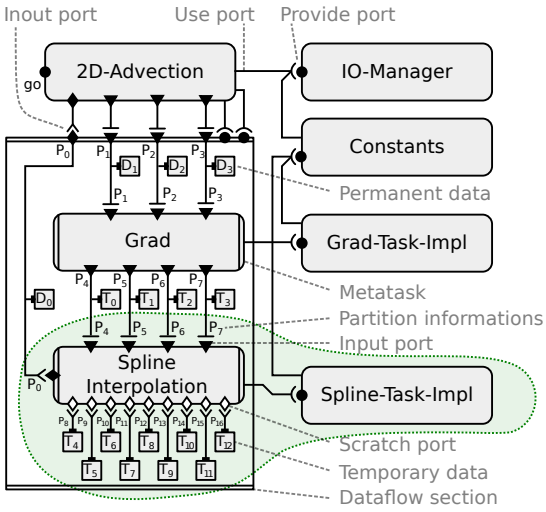

Figure 1: The 2D advection assembly using COMET.

Out ports are only used when the entity writes data, *in ports* mean it only reads it, *inout ports* mean it does both, and *scratch* represent temporary read/write memory buffers, whose content is neither provided from outside the task nor meaningful after its execution. Each port is associated with a partition type and a data type.

Data buffers are instances of multidimensional arrays typed by a rank (array dimension), a size for each dimension and the type of the contained data. There are two kinds of data buffers: permanent data buffers that aim to hold a user reference to pre-allocated data, and temporary data buffers that control data life-cycle and allocate them if necessary. Each data buffer instance is associated with a list of *partition instances*. As already stated, the current implementation only supports block partitioning.

The application is described in an assembly containing component instances, dataflow sections, and connections between them. Component instances can be connected through use-provide connections that enable the instance exposing the use ports to call functions implemented by the instance exposing the provide port. Component instances can also interact through dataflow sections by connecting the data ports of the components to those of the dataflow sections. Dataflow sections are started and managed through a management oriented provide port exposed by each section.

B. The COMET Programming Model on 2D Advection

1) Representation: Figure 1 proposes a description of the 2D advection use-case using the COMET programming model. Component instances (*2D-Advection*, *IO-Manager*, etc.) are represented as simple edged rounded boxes and the dataflow section as a large double edged rectangle. Metatask instances (*Grad* and *Spline Interpolation*) are displayed as double edged rounded boxes, and data instances as squares (D_i for permanent data buffer instances and T_i for temporary data buffer instances). Partition instances that are exposed by their associated data buffer instance are displayed as P_i annotations on data ports. The figure only contains instance names. For clarity, other information (*i.e.,* instance attributes) such as data size for data buffer instances or block size for block partition instances are not represented.

Provide ports are represented as a full dark circle. They are connecting to use ports without any specific representation. Data connectors are lines connecting data ports and data buffer instances. They are used to describe the flow of data in the dataflow section.

2) Description: The *2D-Advection* component controls the use-case execution: first, it obtains data from the *IO-Manager* component that deals with files in our case. Second, it sends this data to inout ports of the dataflow section to compute the result of one 2D-advection step. Third, it starts the dataflow section computation using the management dataflow section port. The current model does not deal with the automatic start of the dataflow section, since it requires defining a precise semantics when multiple components submit data. This is left for future work if the current model is proved to be able to achieve high-performance.

The *Constants* component, initialized by *IO-Manager*, deals with providing some constant data to the dataflow.

The two distinct parts of the dataflow section of the 2D advection computation described in Section II (*i.e.,* displacement fields and interpolation) are expressed by two interconnected metatasks: the *Grad* metatask (for displacement fields) and *Interpolation* metatask (for the actual interpolation).

Metatasks delegate the actual computation on data fragments using their use port connected to the provide port of the respectively *Grad-Task-Impl* and *Spline-Task-Impl* component instances. Because the interpolation metatask requires temporary data buffers to compute data fragments, it is connected to many temporary data instances.

The actual dataflow is quite straightforward: data are sent from the component *2D-Advection* to the dataflow section, where they are partitioned into fragments using block partitioning and perfect alignment of fragments of the various arrays. Those fragments are used to generate tasks from the metatask *Grad*; the output fragments of this metatask are used for the tasks of the metatask *Spline*. Then, fragments are gathered to form non-partitioned data sent back to the component *2D-Advection*.

3) From spline to Lagrange Interpolation: Switching the interpolation method from spline to Lagrange has required replacing the *Interpolation* metatask and its used task implementation (green area with dotted edges on Figure 1). Moreover, as the *Spline Interpolation* and *Lagrange Interpolation* metatasks have different temporary data needs, the number of scratch ports and their type have also been modified.

C. The COMET Runtime

The COMET runtime deals with low-level components and tasks. It is built on top of both L^{2}C and StarPU [8]. Components have use and provide ports as well as attributes. StarPU is used to submit and manage tasks.

The COMET runtime is a L^{2}C assembly with three types of components: user level components (coming from the COMET programming model), expert components, and glue components. This L^{2}C assembly is made of all components (including their use-provide connections) but the dataflow section of a COMET assembly. An assembly pattern (*cf.* further) is used to transform all COMET dataflow sections into subsets of this L^{2}C assembly (called task sections).

The management related to tasks (such as task submission to the scheduler, data management, data partitioning) is done inside specialized components. Such components are divided into two main categories: expert components and glue components. Expert components are expected to be written by experts; an example of such a component is the runtime component that deals with block partitioning. Glue components are components generated by the compiler presented in Section IV-D, typically to deal with the management of task sections. Only such components expose StarPU-specific interfaces so the COMET runtime may use another compatible tasking runtime without involving users.

Task sections conform to the assembly pattern shown in Figure 2. Components can be instantiated more than once (*e.g.,* the TaskGen component is instantiated as many times as there are metatasks in a dataflow section). Components may expose a variable number of use or provide ports. Cardinalities are used to indicate the number of ports: A cardinality of 1 means that there is a single port, while a cardinality of many ($*$ in Figure 2) means that this generic port is instantiated as multiple ports. Cardinalities do not give any information about the actual connections (*e.g.,* multiple use ports connected to a provide port). Let us detail a little more these components.

TaskGen: Using StarPU, this glue component is responsible for submitting a group of independent tasks that work on a set of partitioned data buffers. The task submission is done when a call is received from the *go* provide port. When tasks are executed, they perform a call on the *compute* use port. Because the task submission requires defining the input/output data fragment of each task, this component needs to access the actual dependencies using the *mapping* use port and to retrieve the actual data fragment through the *partInfoDF* use port.

Task: This user component contains the code of a task. This code is provided through the *compute* port.

DataflowGen: This glue component coordinates the submission of task groups (through the use port *go*), the data partitioning (through the use port *partition*), and the interactions with external components (those providing or requiring data via the *control*, *state*, *inData* and *outData* provide ports) by doing calls on other component interfaces of the task section. It also manages the interactions between this section and the user components outside the section: input data are gathered using use ports (using *inData* ports); output data are retrieved using provide ports (when the computation ends, using *outData* provide ports); the port *control* enables the submission of all the kernel tasks; the *state* provide port enables access to the section running state.

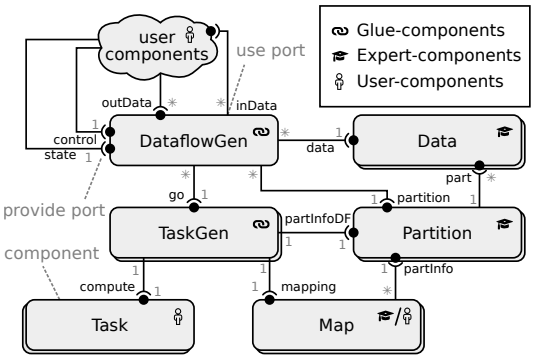

Figure 2: Generic COMET runtime assembly associated to a task section.

Partition: This expert component is responsible for data partitioning. Because tasks are submitted on data fragments, and data can have several partitioning during their life cycle, this component is responsible for the partitioning and the unpartitioning. This is controlled via its *partition* provide port. This component also exposes the *partInfo* and *partInfoDF* provide ports to get information about data partitioning (*e.g.,* the number of data fragments) or specific data fragment information (*e.g.,* type, shape, location inside a data partition, etc.). Currently, there is only a block partitioning component implemented.

Data: This expert component manages the life cycle of data inside the task section (when data comes in/out of the task section, or whenever a data buffer is partitioned). In particular, it ensures the compliance of task dependencies over multiple partitions of a given data buffer (*i.e.,* sequential consistency), preventing tasks to work on invalid data fragments. This component also defines the type of data buffers. Those services are provided via its *data* and *part* ports.

Map: This expert component provides the set of dependencies between the tasks and the data fragments of a given task group (*i.e.,* alignment) through its *mapping* provide port. Because alignment often requires information about involved data partitions, this component retrieves them via its multiple *partInfo* use ports. Currently, there is just a simple map component that only supports a perfect alignment.

D. The COMET Compiler

The COMET compiler is currently a straightforward Python code: from a COMET assembly, it generates the

runtime L^{2}C assemly by copying COMET component instances into L^{2}C component instances and by translating metatasks and dataflow sections of the COMET programming model into L^{2}C components and assemblies according to the pattern described in Figure 2. COMET data ports are transformed in use-provide ports into the runtime assembly.

The partition and mapping information are partially taken into account: for the partitioning, the block size (per dimension) is used whereas nothing is used from the alignment but its type as only a perfect alignment (without any parameter) is currently supported. Nevertheless, we believe it is not currently critical as our goal is to study the feasibility and the performance of mixing components and tasks. For both cases, the corresponding L^{2}C expert component is selected by the compiler and inserted in the runtime assembly.

V. EVALUATION

This section evaluates the COMET approach (*i.e.,* both the programming model and the runtime) on the 2D advection use-case. It first studies the software engineering properties of the programming model by analyzing COMET assemblies of the 2D advection use-case; then it focuses on the performance of the runtime by evaluating the performance of generated assemblies.

For comparison, the 2D advection spline and Lagrange variants have been implemented in two versions each. The reference implementation is directly extracted from GYSELA, but rewritten as a standalone C++ prototype. It is a pure OpenMP fork-join version using a *parallel for* to iterate over plans of the five-dimensional data buffers. The COMET implementation makes use of both components and metatasks of the COMET model.

A. Software Engineering Evaluation

The software engineering properties of the programming model are analyzed with respect to the expected features defined at the end of Section II-C.

1) Algorithmic substitution: The COMET programming model enables algorithmic substitution through components (including metatasks) and their composition. Indeed, component interfaces are well defined and thus component (*i.e.,* subparts of the application algorithm) can be easily replaced by other components providing (at least) the same interfaces. In the case of the 2D advection, Figure 1 shows that the *interpolation* metatask can be easily replaced by another metatask in order to substitute the interpolation method.

Expression of algorithms is made possible through the two kinds of component composition (*i.e.,* both data and service sharing based composition) brought by the COMET programming model. Data dependency based composition enables to compose metatasks inside the dataflow, while service sharing based composition provides a way to share code between metatasks of the dataflow, thanks to a use port connected to a component that sets the task implementation. This enables reusing code between the implementation of metatasks as well as relatively fine-grained composition in task implementation [22]. Indeed, from the 3,246 lines of code of all the user components of both the spline and the Lagrange variants (excluding expert components that contain altogether 798 lines of code shared between the two variants), 1,576 are shared between the both variants, 1,182 are specific to the spline variant and 488 are specific to the Lagrange variant (49% of all the user code can be fully reused). By comparison, in GYSELA, the two variants are currently developed in separated branches.

2) Expression of both task and data parallelism: The expression of task parallelism is achieved through to the composition of metatasks of dataflow sections, while data parallelism is expressed inside metatask using data partitioning and alignment expressions. In the case of the 2D advection, the two main metatasks are composed together to complete one time-step. At runtime, metatasks produce fine-grained tasks to deal with data parallelism according to the data alignment rule (defined in both metatasks). Consequently, the implementation of the two main parts of the 2D advection can be defined separately and executed simultaneously (data streaming), without any artificial synchronizations between the two units, since task can be scheduled regardless of the submission order. Finer-grained decomposition could even be achieved (especially for the independent field computations of the *Grad* metatask), but was not applied to remain close to the reference version of GYSELA to make comparison meaningful. Thus, expression of both task and data parallelism is possible without manually setting any arbitrary optimization choice, only granularity has to be tuned to get high-performance.

Parallelism expressiveness of the current COMET approach is limited to the fine-grained composition of multiple interdependent bags of tasks (produced by metatasks). Parallel patterns such as reduction or scan for example are not yet supported. This is due to the fact that the current set of features has been chosen to validate the approach on the 2D advection use-case. The addition of such features is left for future work that, in addition, could study the extensible properties of the COMET programming model and its runtime.

3) Optimizations: Optimizations of the 2D advection can be described separately in the COMET programming model, thanks to components and assemblies. Indeed, optimizing partitions and their parameters (*e.g.,* block size and shape) is just a matter of replacing partition components and tuning component attributes. Adapting metatask granularity can be mostly done by tuning assemblies, whenever possible: multiple metatasks can be aggregated to reduce the size of the scheduled DAG and the use of temporary data buffers, then task composition can be replaced by new simple components to make static calls on the finer-grained task implementation components. Moreover, new task scheduling strategies can

be implemented to schedule more efficiently the task DAG produced by dataflow sections over the available resources.

4) High Level architecture: The COMET programming model ensures the separation between high-level architectural level optimizations and low-level optimizations by setting the first one inside assemblies and the second one inside components. Indeed, in the 2D advection, high-level architectural level optimizations such as the parallelization method are only described in the assembly while low level-optimizations only lie inside the code of components.

B. Performance Evaluation

This section deals with the performance evaluation of COMET on the 2D advection use case, including the spline and Lagrange variants.

Performance and scalability are evaluated up to 16 cores on a shared memory node of the MdlS-Poincare cluster (IDRIS, France). Each node contains 2 sockets of 8 cores Intel Xeon E5-2670 (2.60GHz).

Each experiment has been done 20 times and the median is displayed. Most of the figure use logarithmic scales. Error bars on plots correspond to the first and last quartile. The compiler used is Intel ICC 15.0.0 (the latest version available on the machine) and the OpenMP runtime used is the one proposed by the compiler. Performance results are only those of the computation kernel. They do not include the initialization phase that includes loading data from disk.

1) Strong Scaling Performance: Figures 3 and 4 display the completion time of the strong scalability of the reference and COMET implementation on the respective spline and Lagrange variants. They also show the completion time of an intermediate version to understand the gap between the reference and the COMET version. This intermediate version is very close to the COMET version in term of dataflow but it uses a static OpenMP fork-join scheduling instead of a dynamic task-based one. This intermediate version is completely monolithic and sacrifices maintainability for performances as it does neither split code using subroutines as in the reference case nor using COMET features (it uses a sequences of nested loops calling single functions that work are the finest granularity to expose data dependencies between computational parts).

Sequential completion time of the reference implementation is 16% slower than the COMET version on the spline variant and 12% faster than COMET on the Lagrange variant. On 16 cores, the COMET version is respectively 18% faster and 21% slower than the reference implementation on the spline and Lagrange variants. Completion time of both the intermediate and COMET version are close (less than 1% in sequential and up to 6% 8% on 16 core) on the both variants. All versions scale relatively well until 16 cores both on the spline and the Lagrange variants: they are from 12.7 to 13.8 times faster than the sequential completion time on the both the spline and Lagrange variants.

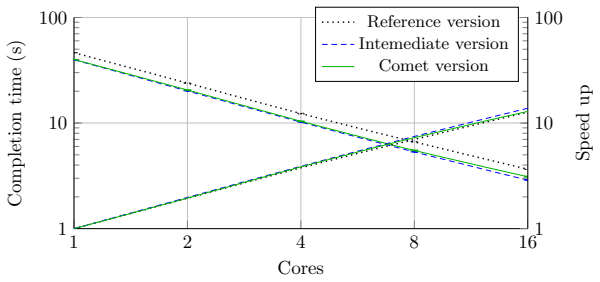

Figure 3: Completion time (left) and strong scalability (right) of the three versions of the spline variant.

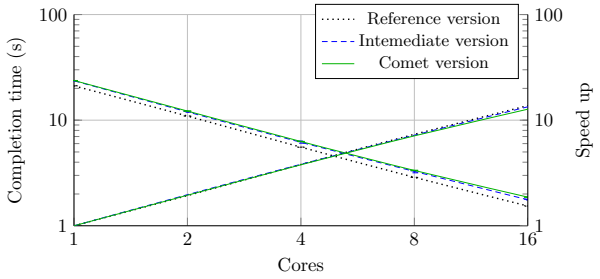

Figure 4: Completion time (left) and strong scalability (right) of the three versions of the Lagrange variant.

The gap between the reference and the intermediate version is huge compared to the gap between the intermediate and the COMET version. This is due to the change of programming paradigm: by focusing more on the flow of data in the code, compiler optimizations differ providing a code more efficient on the spline variant but less efficient on the Lagrange variant (due to pointer aliasing). However, the gap comes mainly from sequential optimizations, which are not the concern of this paper. Let us focus on the differences between the intermediate and the COMET implementation.

2) Overhead Analysis: Figure 5 shows the relative completion time of the COMET version over the intermediate version in function of the number of cores (relative strong scalability) for both the spline and the Lagrange variants.

The COMET implementation is almost as fast as the intermediate version in the sequential case for the two

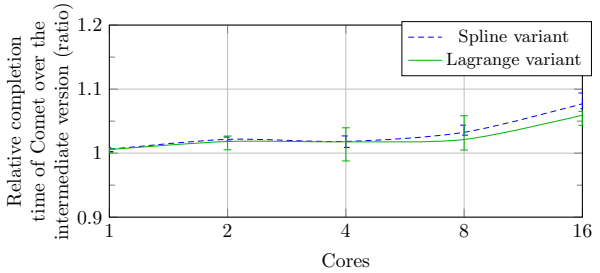

Figure 5: COMET relative performance over the intermediate implementation on the spline and Lagrange variants.

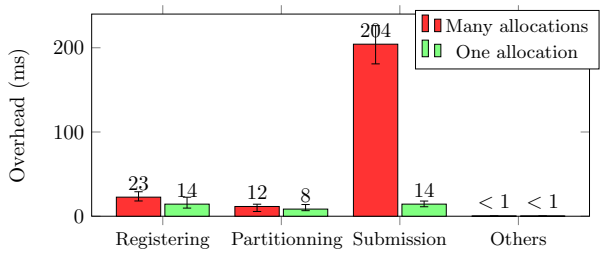

Figure 6: Comet overhead source on the Lagrange variant.

variants (less than 1% slower). But, as the number of cores grows, the overhead of the COMET version is becoming higher, reaching a peak of 8% on 16 cores for the spline variant and 6% for the Lagrange version. As this overhead is not negligible, its source has been deeply analyzed on the Lagrange variant.

The source of this overhead lies in the tasking thread of the COMET runtime, responsible for the registration of data to StarPU, the data partitioning as well as the submission of tasks to the scheduler. This thread is not bound to a specific core, resulting in a higher overhead when it overloads too much the cores used to complete the scheduled tasks.

Figure 6 breaks down the amount of time taken by each concern of the tasking thread of the COMET runtime on 16 cores. This thread takes 240 ms to complete his work, which is much compared to the 1.76 s taken to complete the whole 2D advection computation. The major part of the overhead is due to the abnormally slow task submission. It is caused by the many temporary data fragment allocations done during task submissions. These are used to handle data transfers between the *Grad* and the *Interpolation* metatasks.

However, this overhead can be reduced by allocating the whole data buffers rather than doing per fragment allocations. Figure 6 also displays the effect of the two allocation policies (per fragment allocations vs per buffer allocations) over the time taken by all the concerns of the tasking thread. The time taken by the task submission falls down to 14 ms when one allocation per temporary buffer is used, causing the tasking thread completion time to drop from 240 ms to 37 ms.

3) Temporary Data Allocation Overhead: In COMET, temporary buffers are used to ensure the communication between the bags of tasks generated from metatasks at runtime: when a bag of tasks writes into a data buffer read by another bag of tasks, data fragments have to be temporary stored, waiting to be read, according to the chosen task scheduling policy. The use of such temporary buffers comes from task scheduling. Indeed, task scheduling is performed at runtime and the scheduler can choose either to execute first all the tasks of the writing bag and then the other tasks or to interleave the execution of the tasks of the two bags.

Figure 7 shows the relative completion time of the COMET version over the intermediate one over the number of cores

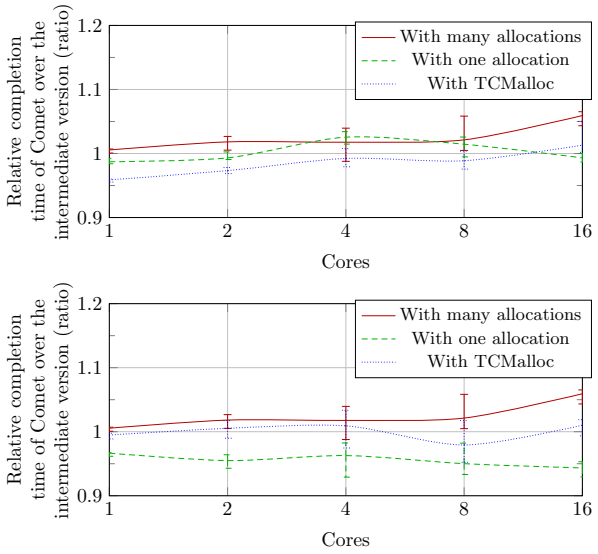

Figure 7: COMET relative performance over the intermediate version on the spline variant (top) and on the Lagrange variant (bottom) using various allocation policies.

using three different allocation policies in the cases of the spline and Lagrange variants. The new policy called *TCMalloc* reuses exactly the code of the *many allocation* policy, but it replaces the default GNU C library malloc implementation by the *Google Performance Tools TCMalloc* library[1] one at runtime.

The *one allocation* policy outperforms the *many allocation* policy whatever the number of core used on the both spline and Lagrange variants. Moreover, it does not suffer from the scalability problems of the *many allocation* policy. However, since the whole temporary data buffers are quite big (around 1 GB), it causes a high memory consumption that may be unacceptable.

The *TCMalloc* policy solves this by allocating chunks of fragments on demand and by reusing them. On 16 cores, the COMET version with this policy is only about 1% slower than the intermediate version on both the two variants. Finally, when using 16 cores, the COMET version using the TCMalloc allocation policy is 28% faster than the reference implementation on the spline variant and 16% slower on the Lagrange variant due to the performance gap between the reference version and the intermediate one, mainly coming from sequential compiler optimizations.

The *TCMalloc* policy is a good trade off between the two other policies as it provides performance similar to the intermediate version while reducing memory consumption. Thus, the use of the COMET model does not add a significant overhead compared to the intermediate version when using an adequate temporary buffer allocation policy.

[1] *cf.* https://github.com/gperftools/gperftools

VI. Conclusion and Future Work

To harness today supercomputer computing power while easing maintainability of HPC applications, this paper has studied the feasibility of efficiently combining both a software component model and a task-based model. Through the COMET approach, this paper has described how the models can be combined into a programming model and a runtime model. The evaluation has been made on a real-world use-case extracted from the GYSELA application.

Evaluations demonstrate the applicability of the approach. Experimental results show that independent software codes can be easily composed and replaced while being efficiently executed at runtime: performance results are equivalent as those obtain with the reference implementation. However, specific care must be taken for the temporary data management to achieve high-performance.

Future works include the support of more complex partitioning and alignment functions, and the support of parallel patterns such as scan or reduction. The support of accelerators such as GPGPU needs also to be investigated as well as the integration of parallel languages, including the tasking model of OpenMP.

Acknowledgment

The authors wish to thank Guillaume Latu for his contribution to explaining us the 2D advection part of Gysela. This work has partially been supported by the PIA ELCI project of the French FSN.

References

[1] M. D. McIlroy, "Mass-produced Software Components," *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] B. A. Allan and al., "A Component Architecture for High-Performance Scientific Computing," *International Journal of High Performance Computing Applications*, 2006.

[4] J. Bigot, Z. Hou, C. Pérez, and V. Pichon, "A low level component model easing performance portability of HPC applications," *Computing*, 2013.

[5] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for hpc with logical regions," in *Intl Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[6] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *International Parallel and Distributed Processing Symposium*. IEEE, 2015.

[7] J. Bueno, J. Planas, A. Duran, X. Martorell, E. Ayguadé, R. M. Badia, and J. Labarta, "Productive programming of gpu clusters with ompss," in *International Parallel and Distributed Processing Symposium*. IEEE, 2012.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2011.

[9] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, C. Ehrlacher, D. Esteve, X. Garbet, P. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso, "A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations," *Computer Physics Communications*, 2016.

[10] J. Bigot, V. Grandgirard, G. Latu, C. Passeron, F. Rozar, and O. Thomine, "Scaling GYSELA code beyond 32K-cores on Blue Gene/Q," *ESAIM: Proceedings*, 2013.

[11] V. Grandgirard, "High-Q club: Highest scaling codes on JUQUEEN – GYSELA: GYrokinetic SEmi-LAgrangian code for plasma turbulence simulations," March 2015. [Online]. Available: http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/Gysela/_node.html

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Intl Conf. on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[13] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *International Symposium on Parallel and Distributed Processing*. IEEE, 2013.

[14] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, A. Richards, R. Namyst, B. Bachmayer, C. Kessler, D. Moloner, and P. Sanders, "The PEPPHER approach to programmability and performance portability for heterogeneous many-core architectures," in *ParCo*. IOS press, 2011.

[15] A. Espinosa, P. Beckman, M. Hategan, Z. Zhang, M. Wilde, K. Iskra, I. Foster, B. Clifford, and I. Raicu, "Parallel scripting for applications at the petascale and beyond," *Computer*, 2009.

[16] OpenMP Architecture Review Board, "OpenMP Application Programming Interface Version 4.5," 2015. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf

[17] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. B. Fornarino, "A data-driven workflow language for grids based on array programming principles," in *Workshop on Workflows in Support of Large-Scale Science*. ACM, 2009.

[18] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert, "FlowVR: a middleware for large scale virtual reality applications," in *International Euro-Par Conference on Parallel Processing*. Springer, 2004.

[19] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, 2014.

[20] K.-K. Lau, L. Safie, P. Stepan, and C. Tran, "A component model that is both control-driven and data-driven," in *International Symposium on Component Based Software Engineering*. ACM, 2011.

[21] H. L. Bouziane, C. Pérez, and T. Priol, "A software component model with spatial and temporal compositions for grid infrastructures," in *International Euro-Par Conference on Parallel Processing*. Springer, 2008.

[22] H. Coullon, C. Pérez, and J. Richard, "Feasibility Study of a Runtime Component-based Model Integrating Task Graph Concept on a 1D Advection Case Study," 2016. [Online]. Available: https://hal.inria.fr/hal-01348204