Using Semantics-Aware Composition and Weaving for Multi-Variant Progressive Parallelization

Johannes Mey¹, Sven Karol², Uwe Aßmann¹, Immo Huismann³, Jörg Stiller³, and Jochen Fröhlich³

¹ Software Technology Group, Faculty of Computer Science, TU Dresden, Dresden, Germany
{johannes.mey|uwe.assmann}@tu-dresden.de

² Chair for Compiler Construction, cfaed, TU Dresden, Dresden, Germany
sven.karol@tu-dresden.de

³ Chair of Fluid Mechanics, Faculty of Mechanical Science and Engineering, TU Dresden, Dresden, Germany
{immo.huismann|joerg.stiller|jochen.froehlich}@tu-dresden.de

Abstract
When writing parallel software for high performance computing, a common practice is to start from a sequential variant of a program that is consecutively enriched with parallelization directives. This process – progressive parallelization – has the advantage that, at every point in time, a correct version of the program exists. However, progressive parallelization leads to an entanglement of concerns, especially, if different variants of the same functional code have to be maintained and evolved concurrently. We propose orchestration style sheets (OSS) as a novel approach to separate parallelization concerns from problem-specific code by placing them in reusable style sheets, so that concerns for different platforms are always separated, and never lead to entanglement. A weaving process automatically generates platform-specific code for required target platforms, taking semantic properties of the source code into account. Based on a scientific-computing case study for fluid mechanics, we show that OSS are an adequate way to improve maintainability and reuse of Fortran code parallelized for several different platforms.

Keywords: Progressive Parallelization, DSL, Invasive Software Composition

1 Introduction

Writing efficient code for scientific high-performance computing (HPC) is one of the most challenging tasks in software development. Scientific high-performance programming is required, when a physical problem is described as a set of partial differential equations that do not have an analytical solution. In general, this implies the application of brute-force solvers generating approximative solutions, e.g., based on the spectral element method (SEM) [12]. Because of their high demands on computational power, scientific programs that use such numerical methods are
typically deployed and executed on HPC clusters with several thousand compute nodes, which can achieve a peak performance of several petaflops [21] using heterogeneous hardware, e.g., normal CPUs combined with modern accelerators such as GPGPUs. Hence, the main objective of the scientific programmer is to leverage as much from this peak performance as possible.

A frequently chosen approach to write parallel programs is to start out with a sequential program that efficiently solves the problem on a single core and then progressively parallelize it using standardized technologies [3] such as MPI [22], OpenMP [18] and OpenACC [17]. In this process, potentially parallelizable pieces of the program need to be identified and adapted according to the best-suitable kind of parallelism (e.g., task-level parallelism). This is done in many refinement-testing iterations yielding a refined and potentially better parallelized version of the program. Frequently, a larger number of different variants is developed, e.g., optimized for a specific GPU or CPU architecture, or to assure performance portability, if the underlying hardware or the compiler changes that need to be tested, profiled and maintained individually. Since the development of scientific code can take several years and generations of scientists, and often results in very large programs with more than 100.000 lines of code [23], maintainability and reusability of the variants is important – software costs more and typically lives longer than the underlying hardware [8].

We have developed orchestration style sheets (OSS) as a novel tool to support scientific programmers in progressive parallelization. In our approach, a single sequential version of the code is step-wise refined with different style sheets, deriving variants of the program automatically and orchestrating the underlying hardware appropriately. As in web cascading style sheets, whose goal is to separate the concern of layout, OSS encapsulate and separate the parallelization concern from the core program: sequential core programs are free of entangled parallel concerns. The single-source principle is enabled for a family of platform-specific parallelized variants of a program. Functional improvements of the program, bug fixes, or maintenance do no longer take place in a specific parallel variant, but always in the sequential program, from which all parallel variants are re-generated. This is enabled by separating the base code from parallelization directives, so that they can easily be exchanged and varied (separation of parallelization concerns).

The concept is illustrated in a case study based on a scientific Fortran code from the domain of computational fluid dynamics (CFD) developed by domain experts. The approach combines several techniques from invasive software composition (ISC) [1, 13], reference attribute grammars (RAGs) [9], and aspects [15].

The paper is structured as follows. In Section 2, we introduce a CFD use case and its conventionally parallelized version to identify problems of the manual process. Section 3 introduces the basic concepts of orchestration style sheets based on ISC and shows how RAGs are utilized to define parallel styles and derive context-sensitive properties from the scientific code. Section 4 elaborates the benefits of OSS and evaluates its application to the use case. Finally, Section 5 discusses related work and Section 6 summarizes and concludes the paper.

2 Programming with Directive Languages

In this section, we first introduce our use case from the fluid mechanics domain. Afterwards, its implementation and parallelization in Fortran is discussed, and corresponding problems are analyzed.
Figure 1: Simulation of a flame front for the case of $\beta = 20$ and $\alpha = 0.8$ at different times.

2.1 Use Case: Fluid Mechanics Simulation (FMS)

We are concerned with the combustion of a premixed gas in one dimension in the domain $(0, x_{\text{end}}]$ for the times $t \in (0, t_{\text{end}}]$, a test case from numerical combustion research [19]. The problem can be written as

$$\partial_t T = \Delta T + q(T),$$

where $T$ is the temperature distribution one wants to compute, $t$ the time, $\Delta$ the laplace operator and $q$ the reaction heat due to the combustion process, which uses the model constant $\alpha$ and $\beta$, specifying the reaction. In the above formulation, the temperature is normalized to the interval $[0, 1]$ where a “0” implies the unburned state and a “1” the combusted one. A derivation of (1) is given in [20].

A stable reaction front for the case of $\beta \to \infty$ serves as initial condition

$$T(t = 0, x) = \begin{cases} 1 & x < x_{\text{rf}} \\ \exp(x_{\text{rf}} - x) & x \geq x_{\text{rf}} \end{cases},$$

and the boundary conditions of the problem are

$$T(t, x = 0) = 1$$
$$\partial_x T(t, x = x_{\text{end}}) = 0.$$
The main idea of the SEM lies in the decomposition of the domain into \( n_e \) disjunct elements \( \Omega_e \), as Fig. 2 illustrates. On each element, multiple degrees of freedom constitute the solution, and all but one operation in a time step are local to the respective element. E.g. for the test problem (1) the main computational part is the calculation of the right-hand side using an approximation \( T^n = 2T^n - T^{n-1} \approx T^{n+1} \). Fig. 3 shows a simple sequential Fortran implementation of the calculation of the time derivative, incorporating a matrix multiplication, a function call and multiple nested loops. The loop iterates over all elements, but only data from the current element is used. The intermediate result is afterwards joined across element boundaries – an exchange of one value to the left and one to the right, the only data exchange between elements in the whole algorithm. For an in-depth explanation of the algorithm, we refer the reader to [10].

2.2 Parallelizing HPC Code

The progressive parallelization of HPC code is typically a two-step process, where each step addresses a different problem. Starting from a working sequential code, the first step introduces a message-passing layer, leading to a parallelized version that enables large-scale simulations on HPC clusters. The second step consists of the usage of specific hardware, e.g., multi-core processors or accelerators, leading to less communication or more compute power and, hence, better performance. We will illustrate the process on the code example in Fig. 3. The locality of the operations leads to the first coarse level of parallelization called domain decomposition: The elements can be grouped into subdomains which can be treated in separate processes. As most operations are local to the elements, the code in Fig. 3 is unchanged by the decomposition, though the loop bounds change. Only the data exchange between elements needs to be adapted for inter-process communication, typically with message-passing via MPI [22]. Hence, the first step is concluded without changing the code of the computationally intensive parts.

With multidimensional problems, using more processes leads to more communication between them, while the amount of computation per process declines. This results in a communication bottleneck, lowering the parallel efficiency of the code. To circumvent this limitation, less communication, i.e., treating more elements per subdomain, is key. One approach to this end is the utilization of threads inside a multi-core instead of separate processes, while another one is the usage of accelerator cards. Usually, OpenMP is applied for the former [18], while CUDA [16] or OpenACC [17] are employed for the latter. Both approaches employ compiler directives, or pragmas, to state the parallelization. Both are based on threads, though on a quite different scale, and both utilize a similar syntax, as OpenACC is derived from OpenMP.

With OpenMP, the `omp do` statement distributes the loop in Fig. 3 among the threads of the process, leading to the code in Fig. 4, where the private clause is utilized to avoid race conditions in the variables \( k, i, j \) and \( s \). These two additional lines accelerate the code example
do k = 1, n_e
 do i = 0, p
   ! Set to contribution from reaction term
   q(i,k) = ReactionHeat(alpha, beta, T_star(i,k))
   f(i,k) = h/2 * M(i) * q(i,k)
   ! Laplace term
   s = 0
   do j = 0, p
     s = s + 2/h * L(i, j) * T_star(j,k)
   end do
   ! Subtract Laplace term
   f(i,k) = f(i,k) - s
 end do
end do

Figure 3: Main computational loop of the solution process, computing a weighted time derivative f from an approximation to the temperature at the end of time step T^n + 1. The number of elements in the subdomain is n_e, and the number of degrees of freedom per element is p + 1.

Figure 4: Variant of the code in Fig. 3 including OpenMP and OpenACC pragmas.

for multi-cores, though the threads need to be instantiated first, which is done outside the example with two further lines by an OpenMP parallel region. In contrast, OpenACC requires more space. The data region (Line 2) notifies the compiler that the working set is present on the accelerator, thus no data copy operations are required that are limiting computational throughput to the PCIe bandwidth. The parallel region (Line 3) designates its content to be executed on the accelerator; coarse and fine acceleration levels are accomplished by stating that the outer and inner loop are to be executed on the accelerator.

2.3 Problem Analysis

The process of acquiring parallel program variants with progressive parallelization has some severe issues. Parallelized code usually is much longer than its sequential counterpart. This can be seen when comparing Fig. 3 and 4: while a simple OpenMP loop parallelization extends the code by a mere 2 lines of code, the OpenACC version accounts for eight more lines of code. This problem is referred to as the code size problem P.1.

Furthermore, the algorithm is mixed with several different parallelization directives, making either concern less readable, less understandable and therefore less debuggable. This is the entanglement of concerns problem P.2.

A third issue is the scattering of concerns problem P.3, which can be addressed by aspect-oriented programming (AOP) [15]. In terms of our use case, scattering means that the dependency on a specific platform leads to many platform-specific pragmas scattered all over a base program. Whenever the platform should be changed, the scattering impedes the exchange of the pragmas. Therefore, scattering impedes not only comprehensibility, but also variability.

Another set of potential problems arises from the interaction of base program and directives, which must obey the context-sensitive rules of static semantics. Here, we will focus on the consistency problem P.4, which is related to the problem of aspect fragility [2]: directives depend on the code they annotate. Because they are secondary specifications relying on a base program, their applicability and meaning relate to it; they obey specific rules of static semantics that
Figure 5: Main computational loop from Fig. 3 written in Fortran 2008 with the do concurrent construct.

Figure 6: OpenMP style sheet for do concurrent loops with a slot for private variables.

connect them to the base program. For instance, they may relate directly to declarations in the code, e.g., which variables should be treated as private. These dependencies require to maintain the consistency between directives and the base program. Very similar looking pragmas spread over many locations in the code need to be evolved consistently. Frequently, this seduces programmers to copy-and-paste them, which, however, does not ensure the necessary context-specific adaption of the pragmas.

These problems can be remedied by OSS, which are introduced in the following sections.

3 Orchestration Style Sheets (OSS)

OSS borrow concepts from cascading style sheets (CSS)\(^1\) and aspect-oriented programming to address the problems of progressive parallelization. Like CSS annotate XML code with presentation information, OSS annotate sequential code with parallelization hints. Similar to aspects, OSS separate the parallelization concern from the base program. Subsequently, we first discuss the basic model of OSS as well as their processor and workflow. Afterwards, OSS are applied to the example introduced in Section 2.1.

3.1 Artifacts and Workflow

The OSS processor is a source-to-source compiler that converts sequential code into parallel code. The parallelization of the core code, i.e., a set of files, is controlled with style sheets and composition recipes. A style sheet contains parametrizable, generic fragments (code templates) for parallelization and addressing expressions specifying how and where they are applied to the core code. The addressing expressions specify how the sheet is woven into the core code, enriching it with parallelization directives. Finally, a set of composition recipes specifies which styles should be applied to the core. Each recipe results in a set of output files.

We explain the composition process for the example of Fig. 5, from which the style sheet shown in Fig. 6 creates variants parallelized with OpenMP. Alternatively, another style can be used to parallelize the code with OpenACC; the corresponding style sheet is shown in Sect. 3.3. To identify the set of nested loops that could benefit from parallelization, we use a language construct introduced in the Fortran 2008 standard [11]: do concurrent, that specifies that the

\(^1\)https://www.w3.org/Style/CSS/
iteration steps over its loop variables are independent of each other and thus can be parallelized.\(^2\) Fig. 7 shows the workflow of the OSS processor and its artifacts, which are described here:

The **sequential source code** is fully compliant to the Fortran specification to allow the programmer to use standard tooling.

**Style Sheets** are collected in files with the file ending `.oss`. They contain named styles of a type which indicates its purpose. These contain code fragments with addressing expressions that address *static join points* in the core and code fragments of the core that will be augmented. The expression points to a set of manifestations of a specific language construct (e.g., a specific form of a loop) which may be further restricted by the construct’s properties or position. In our example, we apply a style sheet that transforms the `do concurrent` loop, thus the addressing expression in Line 2 of Fig. 6 matches all `do concurrent` in the core. The code in the advice fragment adheres to two languages: the core language and the directive language of the parallelization platform (e.g., OpenMP). The fragment code may contain slots, named placeholders for other fragments (Line 7 of Fig. 6). These slots are filled during the weaving process. The named slots of OSS can be used for flexible control of composition. Slot types can be defined by users, by specifying a simple correspondence between the slot name and the attribute filling it (cf. Fig. 6, line 4). This process is described in the following section.

**Composition recipes** contain a selection of the available styles and define the manner and order in which they are applied to the core code. Each recipe generates a composed code variant. In our example,

```plaintext
recipe openmp { do:loops openmp:parallelization }
```

specifies that the two styles `do:loops` and `OpenMP:parallelization` should be applied; the former transforms the `do concurrent` loop into two nested regular `do` loops, while the latter adds OpenMP directives (see Sect. 3.3).

The **composition process** applies styles to the core as configured in the recipes. To perform the weaving of the style fragments, the abstract syntax tree of the core code is traversed and elements are identified by addressing expressions. Next, these elements are enriched with all applicable styles by wrapping the styles’ code fragments around the core elements. After adorning all abstract syntax trees with the style fragments, these are unparsed and stored in a user-defined directory for compilation or further processing.

In our example, `do concurrent` loops mark parallelizable code sections. A useful aspect of `do concurrent` is that it can contain multiple control variables (Line 1 of Fig. 5). The Fortran 2008 standard specifies that the nesting order of the corresponding loops is unspecified and

\(^2\) Even though this language feature is currently available in the major Fortran compilers, its concurrent execution is not supported – the code is executed sequentially. In this case, OSS turn out to be very useful to provide the concurrent execution of such loops and allow developers to configure the parallelized code.
can be determined by the compiler [11, Section 8.1.6.6.2 (3)]. Thus, the generated do loops may be reordered to gain speed. The do loops in the generated code resemble the original ones from Fig. 3. If the sheet do:loops specifies a different order, the loops can be interchanged to influence performance. Thus, alternative sheets with different orders can be used for profiling.

3.2 Using Attribution in Style Sheets

The OSS processor is designed and implemented using reference attribute grammars (RAGs) [9]. RAGs do not only provide a simple way to access elements of the core code’s abstract syntax tree (AST) with reference attributes, but can also be used to perform program analysis and context-sensitive computations. Attributes and their corresponding semantic rules can be specified for all non-terminals of a grammar. In OSS, we take advantage of this by specifying attributes for non-terminals designated for style application. In practice, the semantic rules of attributes are specified with side-effect free methods taking a non-terminal as an input. Furthermore, if attributes are directly used in the computation of a stylable fragment’s property, they are required to evaluate to an AST node, that can in turn be inserted into a style fragment’s slot. The use of RAGs provides the following two use cases:

**Computation of Fragments** Styles must know many things about the core loops they adorn. Pragmas as well as other fragment code depend on information from the core. For instance, the attribution of the Fortran RAG can be used to check the legality of private loop variables in a pragma. However, when private loop variables are explicitly specified in the style, its reuse is restricted – a new style has to be written for every loop. With attribution rules, the set of private variables of a loop can be computed and reused in the style. In this case, the private variables of the style fragment’s loop $i$, $j$, $k$, and $s$ are computed by attributes. The newly defined attribute **private Variables** can now be referred to from styles, e.g., to compute OpenMP pragma parameters as shown in Fig. 6. Another more complex example for OpenACC is shown in Fig. 8. The attribute **getLower** finds a loop’s lower bound. It can be accessed in slot #LBOUND#. Similarly, upper bounds and step size can be accessed.

**Fragment composition with core attributes** Core attributes can also compute entire Fortran fragments, i.e., abstract syntax tree fragments that may be inserted into fragments contained in styles. Consider the slot #INNER# in Fig. 6. This slots refers to an core attribute **inner** which is defined in the RAG for every stylable Fortran element. $^3$ **inner** returns an AST node $A$ which can be used to parameterize the slot #INNER#.

3.3 Architecture of OSS

This section describes the Orchestration Style Sheet DSL and how the OSS processor works.

**The OSS DSL** Style sheets and recipes are part of a single DSL; they can be spread over many files, facilitating extension. Examples of style definitions are shown in Fig. 6 and 8.

Styles contain **fragments** which include source code with slots hedged within <code/> tags. A fragment’s addressing expression is a reference to a Fortran non-terminal and a selector to pick a subset or all occurrences of it. Finally, fragments can have **slot** parameters to link slots in the code to attributes and possibly parameters to control the weaving. For example, Fig. 8 contains iterateOver: this parameter expects an attribute that returns a list of AST nodes. In the weaving process, the fragment is woven into the code once for every item in the list.

$^3$The **inner** attribute returns the stylable Fortran element by default. In the case of do loops, this behavior is overridden to return only the loop’s body.
The syntax for recipes is very easy: it is a named list of pairs of styles and style types, where every style type may appear at most once.

**The OSS Processor** Our prototype\(^4\) of the OSS processor is implemented in the SkAT composition framework \([14, 13]\), which allows to create preprocessors and aspect weavers for arbitrary languages using the method of invasive software composition (ISC) \([1]\).

SkAT uses JastAdd RAGs \([9, 7]\) as a specification language and provides an attribute grammar-based implementation of ISC. Due to attributes, SkAT-based tools can reuse and extend static semantics of the target language, e.g., for computing context-sensitive information or to check constraints.

### 3.4 Extending the OSS processor

SkAT and JastAdd toolkit are easily extensible and the OSS processor inherits this property. Therefore, the process of customizing OSS is relatively simple and straightforward. If a parser and unparsers are available that are integratable into OSS, even other languages can be styled.\(^5\)

Even simpler is the modification of OSS for different target platforms or concerns to be woven into the code: all that has to be done is to provide the required analysis attributes for the particular use case. This is done by writing JastAdd attributes, side-effect free java methods defined for AST elements. No further knowledge of the underlying system is required.

### 4 Discussion of the Benefits of OSS

With the use of attribution and RAGs, OSS help to solve the problems stated in Sec. 2.3.

First, the core code stays slim, just 12 lines with the additional style files. As OSS scatter their directives over all join points of the base program, they reduce the overall code size: the replication required for the scattering is done by the preprocessor, not by the programmer. Therefore, we presume that the code size problem P.1 can be addressed with OSS. Because parallelization directives are separated from base programs and from those of other platforms,

\(^4\)The OSS prototype is available at https://git-st.inf.tu-dresden.de/johannes.mey/orchestration-style-sheets

\(^5\)A Java variant already exists https://git-st.inf.tu-dresden.de/johannes.mey/orchestration-style-sheets
the concerns for the parallelization of different platforms are clearly separated and no longer entangled (problem P.2). Additionally, OSS support the single-source principle; the sequential version of the base program is the single source of functional development, and style sheets separate parallelization directives, thus reducing the scattering of concerns (problem P.3).

The main benefit of OSS is that they respect context-sensitive static semantics, both of the base language as well as of the directive languages. Using the underlying RAG, context-dependent parameters, names, and code fragments can be computed as attributes and used to parameterize the style sheets. Therefore, manual parameter specification in styles, as well as manual parameter extraction from the base program are avoided, the specification of the styles gets simpler and style reusability increases. Thus, the amount of potentially inconsistent directive definitions is reduced (problem P.4).

4.1 Evaluation of the Use Case

As already mentioned in the last section, OSS are flexible and extensible, the system provides a very lightweight approach that can easily be adopted to other use cases with special, user-defined analysis attributes and even to other languages.

Furthermore, it can be shown that for our use case, OSS significantly shortens the Fortran core code. After the removal of the directive code, the code is much shorter and contains almost only the core concern. The additional effort necessary to write the style sheets is also beneficial, because now directives are parametrized by attributes and thus can be reused, reducing the total number of directives.

Both the straightforward creation of an OSS processor tailored for a particular use case and its easy and efficient application make OSS a well-suited candidate for the automated progressive parallelization in arbitrary languages.

5 Related and Future Work

There are different approaches assisting the generation of parallel code OSS can be compared to.

Aspect-oriented programming (AOP) [15] separates cross-cutting concerns from the application core by putting them into separate aspect components that are composed during compilation. However, the join points in standard model of AOP do not offer enough information to the aspects to safely and efficiently parallelize code fragments. Furthermore, AOP has only limited support for the composition of advices, which is an important feature of directive languages. In particular, conflicts between advice definitions are not handled. This deficiency is mainly due to the lack of support of composition programs in AOP approaches [1], which would allow for reasoning about conflicts and dependencies. Despite these problems, AOP can be used to parallelize some loops with static analysis to ensure correctness of loop parallelization [6]. However, this approach is limited in its application because of strict requirements for parallelizable loops and a limited set of generatable variants. In particular, much of the information required for distributed memory parallelization is not available.

OmpSs [4] is a directive-based parallel programming model extending the OpenMP language. It supports a heterogeneous programming model for CPUs and GPUs with its own execution and memory model. Like our approach, OmpSs is implemented using source-to-source transformations and supports C, C++ and Fortran code. However, OmpSs targets a specific runtime environment.

---

6In order to remove all OpenMP and OpenACC pragmas, few new pragmas have to be introduced, e.g., to demarcate single regions and to control data transfer, this is done with a stylable !$oss directive.
and has no extension concept. In contrast, OSS are a lightweight and extensible approach that can be adopted to any target runtime on any HPC cluster.

OSS share similarities with programming macros such as C preprocessor macros or C++ templates. The code in a style’s fragment can be compared to a macro definition (syntax checks left aside) or a template (with syntax checks). However, there are major differences. OSS are based on invasive software composition and provide a customizable fragment component model so that they can be adopted by users for new parallelization use cases and syntactic constructs. Additionally, OSS may use information from static analysis to derive parameters automatically while in macros and templates theses would need to be specified by the user. In future work, we intend to also use this information to provide consistency checks during the processing phase.

Finally, it may be an opportunity to use general term-based transformation systems such as TXL [5] for style sheet based composition. However, we see at least two obstacles. First, the OSS weaving and fragment composition operations would need to be modeled accordingly. Second, we use reference attribute grammars as a mechanism for program analysis in the OSS processor, e.g., for extracting definitions and types from the context. Thus, the transformation system would need to provide an analog approach to declarative analysis specification or integrate with an attribute grammar tool.

6 Conclusion

We introduced orchestration style sheets, a novel approach to perform progressive parallelization. Using a fluid mechanics case study, we analyzed the process of parallelizing a loop for multiple target platforms and identified several potential problems with manual parallelization. With RAGs and ISC, powerful methods are used to generate a very flexible yet easy to use approach. OSS do not only offer improved reuse and maintainability through a separation of concerns, but can also eliminate error sources by replacing otherwise manual steps in the parallelization process with automatized semantic analysis. Furthermore, the approach is lightweight and scalable with little overhead. Finally, the separation of core code and styles and therefore the possible generation of different variants offers new opportunities for improved programming workflows, potentially facilitating processes like testing, benchmarking or auto tuning.

Acknowledgment: This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence “Center for Advancing Electronics Dresden”.

References


