Towards Generic Parallel Programming in Computer Science Education with Kokkos

Jan Ciesko
Computer Science Reseach Institute
Sandia National Laboratories
Albuquerque, NM, USA
jciesko@sandia.gov

David Poliakoff
Computer Science Reseach Institute
Sandia National Laboratories
Albuquerque, NM, USA
dzpolia@sandia.gov

David S. Hollman
Computer Science Reseach Institute
Sandia National Laboratories
Livermore, CA, USA
dshallm@sandia.gov

Christian C. Trott
Computer Science Reseach Institute
Sandia National Laboratories
Albuquerque, NM, USA
ctrott@sandia.gov

Damien Lebrun-Grandié
Computational Sciences and Engineering
Oak Ridge National Laboratory
Oak Ridge, TN, USA
lebrungrandt@ornl.gov

Abstract— **Parallel patterns, views, and spaces are promising abstractions to capture the programmer's intent as well as the contextual information that can be used by an underlying runtime to efficiently map software to parallel hardware. These abstractions can be valuable in cases where an algorithm must accommodate requirements of code and performance portability across hardware architectures and vendor programming models. Kokkos is a parallel programming model for host- and accelerator architectures that relies on these abstractions and targets these requirements. It consists of a pure C++ interface, a specification, and a programming library. The programming library exposes patterns and types and maps them to an underlying abstract machine model. The abstract machine model offers a generic view of parallel hardware. While Kokkos is gaining popularity in large-scale HPC applications at some DOE laboratories, we believe that the implemented concepts are of interest to a broader audience including academia as they may contribute to parallel programming productivity. In this work, we give an insight into design considerations of a representative programming model, list important abstractions, document feedback received during workshops, and give pointers to resources that the reader may consider valuable for a lecture on generic parallel programming for students with preexisting knowledge on this matter.**

Index Terms—**parallel programming, C++, performance portability, programming model**

Parallel and distributed computing is a focus at universities and teaching institutions. The interest is two-fold. Firstly, given their role, educators adapt to trends in research and industry and include parallel computing in their curricula, and secondly, parallel computing is an attractive field for department research and industry collaborations.

While industry-driven collaborations often predefine technological frameworks such as programming environments and targeted architectures, academic teaching leaves this mostly undefined. In this case, educators can select from a variety of solutions, approaches, and trends in parallel computing. Which particular set to choose for teaching in class is not trivial. Complexity is further amplified by the breath of related topics ranging from parallel hardware architectures, programming models, algorithms as well as productivity in parallel software development. However, it is the productivity concern that is gaining importance as the number of hardware solutions, their complexity as well as demands for a quick time-to-solution increase.

While vendor programming models maintain their position in curricula today, we think that the inclusion of parallel programming models such as Kokkos is worth consideration. They offer an easy to understand set of concepts for a vendor- and architecture-independent, performance-portable expression of concurrency which significantly contributes to parallel programming productivity.

Kokkos[1] [1] is a performance-portable, parallel programming model for C++ developed collaboratively by several national laboratories as part of the US Department of Energy Exascale Project [2]. It consists of a programming model specification and a library. The specification defines the execution and memory model and an application programming interface (library API). The runtime library exposes programming primitives, namely execution patterns and types, including views and execution spaces and maps them to the underlying abstract machine model. The machine model creates a generic view of hardware and memories. Finally, the Kokkos library implements the mapping of the programming model to vendor-specific programming models, libraries, and memory layouts. Kokkos has been successfully used in many large-scale projects at Sandia National Laboratories, Oak-Ridge National Laboratories, Los Alamos National Laboratories, University of Utah, and others [3]. We list its key properties as follows.

- Targets an abstract machine model
- Implements patterns, views, and spaces
- Vendor- and architecture-independent and open-source
- Aligns with C++ standardization efforts for parallel programming

[1]from Greek for grain or seed

- Includes exercises and teaching material

The objective of this paper is to present Kokkos from three perspectives. Firstly, we would like to present our thoughts that accompanied the design and development of the Kokkos parallel programming model. These insights may be useful in parallel programming classes that introduce programming models, reason about design choices, and cover productivity in parallel software development.

Further, we would like to promote Kokkos to students and educators as a platform that helps them to understand and experiment with parallel patterns in a way that is generic, vendor-agnostic, and declarative. Abstractions allow students to learn about concurrency without being exposed to many architectural concerns initially and help to create awareness of their productivity benefits in terms of code- and performance portability on current and upcoming hardware architectures.

Finally, we would like to present Kokkos to the educational community as one representative parallel programming model for a trend towards portability abstractions in parallel programming models. We show a set of interfaces that implement patterns, views, and spaces, and capture enough semantic information to produce performance-portable, parallel code on current architectures. Further, we highlight its alignment and contributions to standardization efforts towards supporting parallel programming in C++. For teaching insights, we document feedback received during on-line lectures.

The rest of this paper is structured as follows. The next chapter gives an overview of parallel programming concepts and presents the notion of *semantic capture*. Chapter II presents the abstract machine model used to derive the required semantic information for performance portability in Kokkos. Chapters III and IV present the Kokkos programming model and an example application. In Chapter V we provide insight into a Kokkos back-end implementation. Chapters VI and **??** discuss related work, tutorials and student feedback from tutorials. Lastly Chapter VII concludes this work.

I. Towards Generic Parallel Programming

A programming model exposes constrained semantics that the programmer uses to express intent. Absent a programming model providing additional constraints, the semantics of C-like languages can lead to sequential thinking and execution. Parallel programming models expose parallel semantics to enable developers to create performance-portable code. This is commonly achieved by adding annotations to the base programming language, often through *pragma* clauses or similar language extensions. Other methods expose such semantics through library calls or new languages. All three approaches are conceptually suitable to represent programming paradigms such as tasking, data parallelism, and other patterns. Some representatives are the OpenMP programming model [4], threading libraries, or languages like Erlang [5] in which parallelism is a first-class citizen. In general, it is up to the parallel programming model implementer to expose language constructs or library interfaces (APIs) that combine convenience to the programmer with a sufficiently constrained semantics to map the program execution to the parallel computer hardware efficiently and correctly. Questions of which semantic information is needed, what paradigm to present, and by what syntactic means to do so span an exciting design space that is well worth exploring.

Figure 1 shows the four properties that constitute a parallel programming model and its design. We call it the semantic capture.

```
Constrained Semantics: memory allocations,
memory movement, execution parameters
Parallel Paradigms: patterns, tasks
Expression: language
Control Paradigm: prescriptive, descriptive
```

Fig. 1: Semantic capture: a set of constrained semantics, parallel paradigms, expression and concern define key properties of a parallel programming model.

Constrained Semantics provided by the programmer to the parallel programming model can be grouped into information to express intent or correctness and properties that control critical performance aspects. They address the question of *what*, *where* and *how*. In the context of parallel programming, these correspond to defining which code portion to parallelize, where to run the code, where to access the data, and how to run that parallel code. Synchronization primitives may be considered part of the *what*, and information on the execution properties such as data placement or memory access type may be considered as part of the *how*. Application logic is expressed following a *paradigm* implemented in a *programming language*.

A *Parallel Paradigm* is an abstract representation to facilitate the programmer's understanding of programming rules and program behavior. Which programming paradigm to choose depends on several considerations. Parallel patterns allow the programmer to permit concurrent execution for commonly occurring programming patterns such as loop constructs. Tasking is a paradigm that supports the expression of interdependent work items needed for irregular algorithms. Distributed and correctness-oriented programming models may implement actor-based programming, where each unit of execution represents an actor who communicates over predefined communication channels. This paradigm eliminates access to a shared state and aligns well with message passing programming (MPI). The execution model and memory model aspects of a programming model define its behavior. That is, it defines the relationship between abstract concepts and program execution on the given architecture.

A language defines the syntax of the parallel semantics. Annotations-based languages add semantic information to the base language through pragmas while embedded programming models rely on the base language and offer additional programming interfaces to capture information. Their semantics is defined in the API specification.

Lastly, a programming model must divide responsibility between the developer and the model. We refer to this as

```
#pragma model parallel loop
for ( size_t i = 0; i < N; ++i) {
 /* loop body */
}
```

Fig. 2: Annotations-based programming models extend the semantic of the base language through pragma annotations.

```
model::parallel_for (N, [=] ( const size_t i) {
  /* loop body */
});
```

Fig. 3: Embedded parallel programming uses the base language and relies on programming interfaces to capture information.

defining the *Control Paradigm*. In practice, this is a choice between defining a descriptive and prescriptive semantics. Descriptive semantics requires a developer to express intent. The implementation is the responsibility of the programming model and of the underlying toolchain. Prescriptive semantics requires the developer to specify the mechanisms by which their intent will be implemented. In a prescriptive model, developer success is tied to their knowledge of implementation mechanisms. In a descriptive model that success is tied to the developer's ability to accurately describe their requirements and the toolchain's ability to turn that into implementation.

Figures 2 and 3 show two code examples using the aforementioned parallel programming syntaxes. While both programming models implement the same programming paradigm (parallel patterns), Figures 2 extends the base programming language through pragma annotations. Both, pragma annotations and an API call as shown in Figure 3 can have a descriptive or prescriptive semantics of describing a parallel loop construct. It is up to the programming model specification to make further definitions regarding the separation of concern.

Both examples provide the semantic information on what to parallelize, however, they do not expose enough abstraction primitives that would allow them to map parallel execution on modern architectures efficiently. To define what further information is needed to express concurrency on modern computer architectures, it is important to define an *abstract machine representation* first. In the following chapters we use the words *representation* and *model* interchangeably. In this work, the abstract machine model does not describe a computer in terms of automata theory but in terms of physical or functional components.

II. The Kokkos Machine Model

The Kokkos machine model defines abstractions that represent hardware capabilities for processing and data access. The Kokkos parallel programming model exposes these abstractions to the programmer through C++ and the Kokkos API. By abstracting from physical hardware, the machine model ensures that applications written in programming models targeting this machine model are generic and performance-portable on current and future hardware. The Kokkos parallel programming model is one particular instance of a programming model that builds on top of that machine model. We would like to point out that it is this conceptual differentiation that allows scenarios where the underlying machine model is instantiated in other languages beyond C++, yet the algorithmic specification as well as performance characteristics and portability remain the same.

The Kokkos abstract machine representation targets a design of a future shared-memory computing architecture. The design is shown in Figure 4. It is characterized by multiple latency-optimized execution units and off-die bandwidth-optimized accelerators. Both compute device types can have disjoint memory address spaces with unique performance properties. In such an architecture, execution units might be hierarchically organized with multiple levels of achievable parallelism with different memory access characteristics and coherence properties of caches. In order to ensure performance portability on such a wide range of configurations, an abstraction of compute-resources and available memories is required. For this purpose, we introduce the abstract concept called *space*.

An instance of an *execution space* is an abstraction over an execution resource to which a programmer can target parallel work. For example, an execution space can be used to describe a multi-core processor. In this case, the execution space contains several homogeneous cores organized into arbitrary logical groups. In a parallel programming model that implements this machine model, an instance of such an execution space would be made available to the programmer to run kernels. Adding more logical groups or accelerators simply increases the number of available execution spaces to the programmer.

Memory and memory types are exposed through *memory spaces*. Each memory space provides storage capacity at which data structures can be allocated and accessed. Different memory space types have different characteristics with respect to accessibility from execution spaces and performance.

An instance of a memory space provides a concrete method for the application programmer to request data storage allocations. Following the architecture as discussed earlier, the multi-core processor may contain multiple memory spaces that abstract on-package memory, slower DRAM, and non-volatile memories. Accelerators can provide an additional memory space through its local on-package memory. This is also highlighted in Figure 4. The programmer is free to decide where each data structure is allocated by requesting the corresponding memory space. Programmatically this is achieved by instantiating the according memory space. Kokkos provides the appropriate abstraction of memory allocation and data management. We believe that an abstract representation of execution- and memory devices is a key property towards performance-portable parallel programming.

How a machine model is exposed to the programmer and what the design considerations are towards defining an appropriate semantic capture are discussed in the next section.

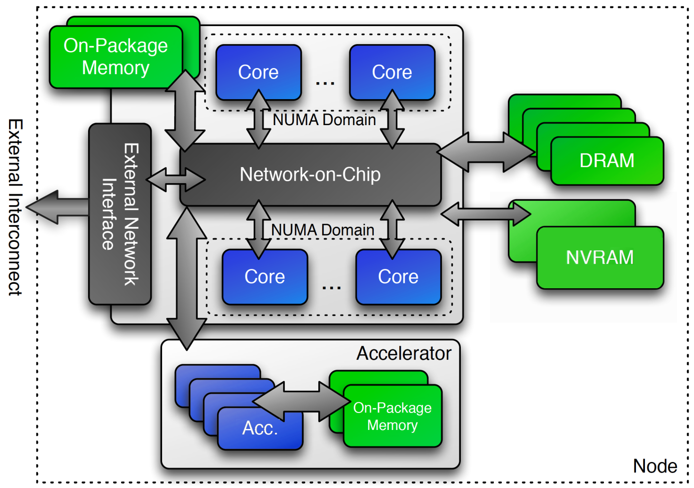

Fig. 4: Spaces represent conceptual building blocks of the abstract machine model in Kokkos.

III. The Kokkos Programming Model

The Kokkos programming model specifies a programming paradigm and an API. The API exposes the machine representation to the developer and defines a set of patterns and types that capture intents and properties of execution. This completes the semantic capture, discussed in Chapter I, We detail all three components as follows.

Kokkos defines a pure interface for C++ and uses C++ as a programming language. This has multiple reasons. It is our experience that application developers of scientific and high-performance computing applications in the high-performance computing community and in the industry in general express the demand for C++ and a generic parallel programming model to support their codes. Further, following their estimates, porting efforts towards adapting to vendors, programming models, APIs, and software releases represents a burden worth addressing. Lastly, the possibility to maintain pure C++ codes is appealing from the code development and debugging perspective.

From the programming model perspective, C++ offers template metaprogramming which is well suited to implement generic APIs and libraries. In this case, class specialization and templating allow for compile-time generated types and optimizations for a given hardware architecture.

Using the programming language, a parallel programming paradigm can be applied to simplify the process of thought. Kokkos supports parallel patterns and tasking. Parallel patterns covers *for*, *scan* and *reduction*. This allows the expression of concurrency over iterative, for-loop computable algorithms. To cover the class of while-loop computable algorithms (irregular algorithms), Kokkos implements the tasking paradigm. Tasks encapsulate work into units that may be executed in parallel to other tasks or sections of the program. Equipped with a language and set of paradigms, a particular set of abstractions can be defined to expose the abstract machine model and capture the intent of the programmer.

To capture semantic information, that is the *what*, the *where* and the *how*, Kokkos introduces six abstractions: *execution spaces*, *execution patterns*, *execution policies*, *memory spaces*, *memory layout* and *memory traits*. These abstractions specify semantic information that enables to capture the programmer's intent and allows the runtime to efficiently map the program to any underlying hardware architecture. We list them as follows:

- An execution space is a place where code can be executed. On current hardware architectures, this corresponds to accelerators and CPUs and can include other compute devices in the future. This abstraction supports remote compute devices in distributed memory scenarios as remote execution spaces.
- Execution patterns expose the parallel programming paradigm. Supported patterns are *parallel_for* loop that executes the loop body in any order a specified amount of times, the *parallel_reduce* which combines a parallel_for with a reduction operation, *parallel_scan* which combines a parallel_for operation with a prefix or postfix scan, and *task* which executes a single function potentially in parallel in respect to other tasks or code sections.
- Execution policies shape the iteration space of a loop pattern. A simple execution policy is a range policy. It specifies that the loop body is executed once for each element in a range.
- Memory spaces are the places where data resides. They specify the physical locations of data as well as access characteristics. Different physical locations correspond to different device types such as high bandwidth memories, on-die scratch memories, or non-volatile bulk storage. Different logical memory spaces allow for concepts such as memory in the CUDA programming model, which is accessible from the host and the CUDA accelerator. Memory spaces, similarly to execution spaces, conceptually support remote memory locations in distributed-memory scenarios. Furthermore, they encapsulate functionality such as consistency control and persistence scopes.
- Layouts express the mapping from array indices to address offsets. By adopting appropriate layouts for memory structures, an application can optimize data access patterns in a given algorithm. If an implementation provides polymorphic layouts (i.e. a data structure can be instantiated at compile or runtime with different layouts), architecture-dependent optimizations can be performed.
- Memory traits specify how a data structure is accessed. Traits express usage scenarios such as atomic access, random access, and streaming loads or stores. This allows the programming model to optimize load and store operations.

Figure 6 shows one definition of the Kokkos *parallel_for* interface. This interface accepts two template parameters and three function arguments. *ExecPolicy* shapes the iteration space for a particular execution space. Examples of exe-

```
Constrained Semantics: Patterns (intent),
Spaces, Layouts, Policies and Traits
Parallel Paradigms: Parallel patterns and
tasking
Expression: Embedded, C++
Control Paradigm: descriptive
```

Fig. 5: Semantic capture defined by the Kokkos programming model.

```
template <class ExecPolicy, class FunctorType>
Kokkos::parallel_for(const std::string& name,
 const ExecPolicy& policy,
 const FunctorType& functor);
```

Fig. 6: The Kokkos *parallel_for* class shows how template metaprogramming allows to specialize types. Its descriptive semantic offers the freedom to optimize execution on modern architectures with multiple degrees of concurrency where an execution policy (*ExecPolicy*) shapes the iteration space accordingly.

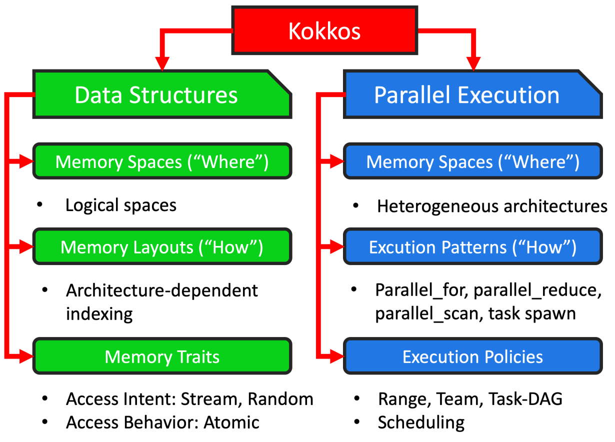

Fig. 7: Overview of abstractions that define the constrained semantic of the Kokkos parallel programming model.

cution policies are the *RangePolicy*, *MDRangePolicy* and the *TeamPolicy*. As the name suggests, these policies express iteration spaces. For example, a range policy defines a one-dimensional iteration range. The MDRangePolicy represents a multi-dimensional iteration space and is used to express tightly nested loop patterns. A team policy defines a 1D iteration range, each of which is assigned to a team of threads. This policy allows the expression of hierarchical parallelism. The second template parameter, *FunctorType*, represent a functor that implements the *()-operator* with a matching signature for the given execution policy. The functor can be defined using a C++ class, struct or lambda expression. We would like to highlight that the return semantic of the parallel_for is defined as *potentially asynchronous*. To guarantee a kernel has finished, a developer should call the fence of the execution space on which the kernel is being executed. Otherwise, it depends on the execution space where the loop executes and whether this execution space implements a barrier.

The semantic of the parallel_for class in Kokkos demonstrates the descriptive nature of the programming model. Programming with patterns and abstractions allows the compiler to generate optimized transformations. Generally, parallel patterns in Kokkos do not guarantee iteration ordering nor the degree of concurrency during their execution. This gives the freedom to the underlying abstraction layers to implement different mapping patterns on different hardware such as the assignment of iterations to threads or vector lanes. We believe that these abstractions and their descriptive semantic represent key programming primitives and the way forward for generic support for parallel programming. In conclusion, Figure 5 shows the semantic capture defined by the Kokkos programming model. Figure 7 summarizes patterns and abstractions and structures them by information type in the semantic capture.

To the interested reader, the complete programming model description can be accessed on-line [6]. The next section shows an example of the Kokkos parallel_for and explains how it is mapped to the underlying back-end programming model.

IV. Example

In this section we provide an example of an application written in the Kokkos programming model. The code in Figure 8 shows a parallel matrix-vector multiplication implemented as two nested loops. The outer loop, that is the outer *parallel_for*, iterates over the rows of the input matrix and of the input vector while the inner loop iterates over elements in each row. Since the inner loop potentially executes in parallel, the update operation of the aggregating variable *y_tmp* must be protected against data races. Using the *parallel_reduction* pattern, the programming model provides a local variable to that loop iteration. Once the computation finishes, the final result of the reduction operation is stored in the output vector *y* at the row index *e*. As a first step, a Kokkos application must call *Kokkos::initialize*. Initialization and finalization are required and serve the purpose of setting up the underlying runtime. Further, the example code shows the use of the *Kokkos::View* abstraction.

Views are abstractions that represent data. Naturally, data can reside in host memory where it is accessible to the host execution space or on device memory where it is accessible to the device execution space. In this example, copies of the views are created to make them accessible by the host process. This is accomplished by calling the *create_mirror_view* function.

If the original data is already accessible by the host process, this function returns a *View* referencing the same allocation. In real applications, a host-accessible allocation is necessary to allow I/O operations during filesystem access. To move data to the respective memory space that is accessible by the parallel execution resources, the function *deep_copy* has to be invoked. In case two view objects are aliased, this operation results in a *no-op*.

Consequently, the simple example does not explicitly specify a memory space where the data lives in. However, Kokkos ensures that the defaults for parallel execution and data allocations match.

Taking a closer look at the loop step shows the hierarchical expression of concurrency. This is accomplished by using the *Kokkos::TeamPolicy*. For each row of *A* a team of threads is launched, where the size of the team is not further specified. When using a *TeamPolicy* the operator of the lambda expression does not receive an index, but rather a handle to the team of threads. This handle provides the team identifier *e*, and is subsequently handed to the nested parallel execution policies. The nested reduction requires the operator of the nested lambda to take a reference to the thread-local reduction variable (called here *y_tmp*) in addition to the loop index. It then writes the result to the correct position of the output vector. The parallel loops are followed by a call to *Kokkos::fence*. This ensures that the execution of the parallel code is completed before copying the results back to the host memory space. This is implemented through the *Kokkos::deep_copy* call. Lastly, the Kokkos application calls *Kokkos::finalize*.

While this example can only provide a brief insight into Kokkos, we recommend to the interested reader to access further resources. Resources include example applications for individual Kokkos features, tutorials with multiple days worth of lectures and exercises, a programming guide, and API documentation. Source code examples as well as the tutorial are located in the Kokkos repository [7]. The tutorial is intended for students with minimal or even no prior knowledge of parallel programming. It introduces concepts of parallel programming using Kokkos through a series of lectures and hands-on exercises that build upon each other and gradually introduce new concepts. The programming guide is available on-line [6].

```
#include<Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
  Kokkos::initialize(argc,argv);
  {
    Kokkos::View<double*> x("x", M);
    Kokkos::View<double*> y("y", N);
    Kokkos::View<double**,Kokkos::LayoutRight>
                              A("A", N, M);
    auto x_h = Kokkos::create_mirror_view(x);
    auto y_h = Kokkos::create_mirror_view(y);
    auto A_h = Kokkos::create_mirror_view(A);
    Kokkos::deep_copy(x,x_h);
    Kokkos::deep_copy(A,A_h);

    Kokkos::parallel_for("outer",
      TeamPolicy<>(N,AUTO),
      [=](const member_type &team_handle) {
        const int e = team_handle.league_rank();
        Kokkos::parallel_reduce(
          TeamThreadRange(team_handle, M),
          [=](const int & i, double & y_tmp) {
            y_tmp += A(e, i) * x(i);
          },
          y(e));
    });
    Kokkos::fence();
    Kokkos::deep_copy(y_h,y);
    //output_result_on_host(y_h);
  }
  Kokkos::finalize();
}
```

Fig. 8: This example application written in Kokkos shows the use of a *view*-type and a parallel for loop using a *TeamPolicy* and a nested parallel reduction.

V. Back-end Support

In this section we peek inside the Kokkos library and show how a generic parallel for loop is mapped to the OpenMP backend programming model. As a template metaprogramming library, Kokkos makes use of partial specialization. In particular the *parallel_for* or *parallel_reduce* functional calls as shown in Figure 8 are mapped to partially specialized classes. Such classes correspond to a particular back-end where for each back-end, concepts with partial specialization must be provided. Today, Kokkos supports the CUDA, OpenMP, HIP as well as one-sided MPI, SHMEM and NVSHMEM back-ends. For each new back-end, partially implemented classes that implement necessary concepts must be provided.

A simplified version of a partially specialized class for a parallel for loop implementation is shown in Figure9. The *parallel_for* function instantiates a partial specialization of the *ParallelFor* class and calls its *execute* function. That function is then responsible to implement the Kokkos parallel pattern. In this back-end, the parallel implementation is generated by using OpenMP pragma annotations and a by invoking a compatible compiler when the library is compiled.

VI. Related Work

C++ increasingly supports parallel constructs. It is becoming increasingly possible to teach parallelism in C++ and relying on modern C++ compilers. However, at the time of publication, C++ does not support accelerators or many of the complexities of parallel computing related to accelerator programming. In courses that do not target accelerators, this can be a relevant option.

Over the past decade the Kokkos programming model has contributed to a number of current and near-future ISO-C++ features, including *atomic_ref* [8], *mdspan* [9], and executors. In particular, the design of C++ executors, which are similar to Kokkos execution spaces, were influenced by this programming model. As the demand to extract performance from increasingly deep and increasingly asynchronous software stacks across a wide variety of domains increases, many C++ experts expect executors to become a central abstraction in any performance-oriented software stack.

RAJA [10] is a C++ abstraction library with similarities to Kokkos. The important difference however is that Kokkos is descriptive, while RAJA is prescriptive. In Kokkos, the programming models determine how an application is mapped

```
template <class FunctorType>
class ParallelFor<FunctorType,
                  RangePolicy<OpenMP>>
{
  const FunctorType functor;
  const RangePolicy<OpenMP> policy;
public:
  void execute() const {
    #pragma omp parallel for
    for(int i=policy.begin();
            i<policy.end();
            ++i)
      functor(i);
  }
};
template <class FunctorType,class PolicyType>
void parallel_for(std::string label,
    const PolicyType& policy,
    const FunctorType& functor) {
  // Call profiling tool hooks with label
  ParallelFor<FunctorType,PolicyType>
    pf{functor,policy};
  pf.execute();
}
```

Fig. 9: This code shows the back-end implementation of a parallel loop in the Kokkos programming model. It shows a particular implementation of a generic pattern as a templated class to the OpenMP execution space. Also, it shows the use of OpenMP pragma annotations that put platform compilers in charge to generate parallel code for the underlying hardware architecture.

to the underlying hardware. RAJA provides functionality that exposes hardware details and relies on the developer to try different strategies to map kernels to architectures. RAJA is a viable option for class education in which the educator would like to expose students to the semantics of different programming models without exposing them to the detailed syntax of those models.

DPC++ [11] is a model developed by Intel® for expressing parallelism across Intel architectures. In addition to sharing a descriptive philosophy with Kokkos, it also shares parallel patterns. Unlike Kokkos, it targets Intel architectures only and is not vendor-agnostic. A course targeting Intel architectures, DPC++ could be a valid option.

VII. Conclusion

In this work we gave an introduction to the Kokkos parallel programming model and presented guiding thoughts behind its design. We discussed a machine representation of a parallel hardware architecture that is characterized by a variety of processing units, memories, and their hierarchical organization. This machine model allowed us to define a constrained semantic represented though different types and patterns. Further, we listed supported paradigms and considerations that influenced the design of the semantic capture.

An example application gave insight into parallel programming with patterns and into the back-end implementation. We used them to show the generic programming interface and type specialization of the implementing classes to support a particular vendor programming model.

To the educational community, we presented Kokkos as a representative programming model of a trend towards the support of parallel programming in base languages and libraries. We believe in the importance of embracing this trend in computer science education as such parallel programming models can contribute to the generic and relevant education of parallel programming and prepare students for their programming careers.

Finally, we would like to express gratitude to educators for taking on this important role. We are open to provide further material or support and propose the interested reader to reach out to the development team on Slack [12].

References

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202 – 3216, 2014.
[2] Exascale Computing Project. https://exascaleproject.org.
[3] "Apps using Kokkos," https://github.com/kokkos/kokkos/issues/1950, 2019.
[4] OpenMP 5.0 Specifications. https://www.openmp.org/specifications/.
[5] F. Cesarini and S. Thompson, *ERLANG Programming*. O'Reilly Media, Inc., 2009.
[6] The Kokkos Programming Model Wiki. https://github.com/kokkos/kokkos/wiki.
[7] Kokkos Programming Model GitHub Repository. https://github.com/kokkos/.
[8] D. Sunderland, H. C. Edwards, H. Boehm, O. Giroux, M. Hoemmen, D. Hollman, B. A. Lelbach, and J. Maurer, "P0019: Atomic Ref," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep., 2018, https://wg21.link/P0019.
[9] D. Hollman, B. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, and C. R. Trott, "mdspan in c++: A case study in the integration of performance portable features into international language standards," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 60–70.
[10] R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin, C. Vaughan, J. Cook, R. Hoekstra, B. Bergen, J. Payne, and G. Womeldorff, "ASC Tri-lab co-design level 2 milestone report 2015," 9 2015.
[11] Intel® oneAPI Base Toolkit. https://software.intel.com/en-us/oneapi/base-kit#specifications/.
[12] Kokkos Slack Channel. https://kokkosteam.slack.com.