International Journal of Parallel Programming
https://doi.org/10.1007/s10766-021-00704-3

SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters

August Ernstsson[1] · **Johan Ahlqvist**[1] · **Stavroula Zouzoula**[1] · **Christoph Kessler**[1]

Received: 16 November 2020 / Accepted: 2 March 2021
© The Author(s) 2021

Abstract
We present the third generation of the C++-based open-source skeleton programming framework SkePU. Its main new features include new skeletons, new data container types, support for returning multiple objects from skeleton instances and user functions, support for specifying alternative platform-specific user functions to exploit e.g. custom SIMD instructions, generalized scheduling variants for the multicore CPU backends, and a new cluster-backend targeting the custom MPI interface provided by the StarPU task-based runtime system. We have also revised the smart data containers' memory consistency model for automatic data sharing between main and device memory. The new features are the result of a two-year co-design effort collecting feedback from HPC application partners in the EU H2020 project EXA2PRO, and target especially the HPC application domain and HPC platforms. We evaluate the performance effects of the new features on high-end multicore CPU and GPU systems and on HPC clusters.

Keywords High-level parallel programming · Heterogeneous computing · Skeleton programming · Co-design approach · Cluster computing

1 Introduction

The recently observed slowdown of Moore's Law implies, for the foreseeable future, that further performance growth in high-performance computing (HPC) critically depends on efficiently utilizing hardware resources, leveraging even more heterogeneity in the form of accelerators such as GPUs and scaling up to even higher degrees of cluster-level parallelism. This leads to programmability and portability issues on

✉ August Ernstsson
august.ernstsson@liu.se

[1] PELAB, Department of Computer and Information Science, Linköping University, Linköping , Sweden

Published online: 19 May 2021

the software side. High-level programming using algorithmic skeletons is a promising approach to bridge this widening gap.

SkePU is a C++-based open-source skeleton programming framework for heterogeneous parallel systems. Based on an already modern and type-safe programming interface, we have redesigned SkePU to especially target the HPC application domain. The resulting third generation of SkePU presented in this paper is the result of a two-year co-design effort taking into account the feedback from HPC application partners in the EU H2020 project EXA2PRO, and aims at striking a good balance between improved programmability for HPC applications and decent performance and scalability on HPC platforms while keeping the strict portability approach of SkePU. The main new features include new skeletons, new data container types for multi-dimensional data and scalable data movement at distributed execution, support for returning multiple objects from skeleton instances and user functions, support for specifying optional, also platform-specific variants of user functions to exploit e.g. custom SIMD instructions, generalized scheduling variants for the multicore CPU backends, and a new cluster-backend targeting the custom MPI interface provided by the StarPU task-based runtime system. We have also revised the smart data containers' memory consistency model for automatic data sharing between main and device memory. We evaluate the performance effects of the new features on high-end multicore CPU and GPU systems and on HPC clusters.

The remainder of this article is organized as follows. Section 2 briefly revisits fundamental concepts of skeleton programming. Section 3 introduces the main concepts in SkePU (as already available in SkePU 2). Section 4 surveys the extensions in SkePU 3. Section 5 presents new skeletons as well as new features and modernized interfaces of existing ones. Section 6 presents new container types. Section 7 explains the new coherence model of SkePU 3, and Sect. 8 presents the principles for execution on clusters. Section 9 reports on experimental results. Related work is discussed in Sects. 10, and 11 concludes.

2 Skeleton Programming Fundamentals

(Algorithmic) Skeletons [5,6] are generic high-level programming constructs based on higher-order functions such as *map*, *reduce*, *scan* etc. that can be instantiated by plugging in sequential problem-specific code parameters, so-called *user functions* and that implement a frequently occurring, often domain-specific, characteristic *pattern* of control and data dependence for a possibly parallel, distributed or heterogeneous target platform. Especially in the context of heterogeneous systems, it is common that a skeleton comes with multiple implementations (called *backends*) for different target platforms, such as backends for sequential, multi-threaded, message-passing or accelerator execution. Skeletons can be realized as libraries or as (often, embedded) domain-specific languages (DSLs) atop a sequential programming language, where C++ is most common today, see also Sect. 10 for an overview of further skeleton programming environments.

Skeleton instances are thus the result of composing multiple software artifacts (a skeleton and one or several user functions), but can be used (invoked) in the same way as traditional hand-written functions.

While skeletons are high-level abstractions of *computation*, they usually also work on high-level abstractions for collections of operand *data*, often in the form of STL-like *data containers* that encapsulate and transparently manage their elements internally [9].

The foremost objective of skeleton programming is improved *programmer productivity* compared to explicit parallel programming, i.e., to make writing programs for parallel, distributed and heterogeneous systems as easy as well-structured sequential programming—where the available set of skeletons fits. Portability at a high level of abstraction is also important, especially in the context of heterogeneous computing systems, where an obvious tuning possibility is the automated selection of the fastest backend depending on the execution context [8]. The cost of abstraction might be a certain loss in efficiency compared to explicitly parallel code written by system experts, however the abstraction might even lead to higher performance where the better structuring and the knowledge of dependence patterns can enable automated optimizations.

3 A Short History of SkePU

SkePU (1) was introduced in 2010 [11] as a skeleton programming library for heterogeneous single-node but multi-accelerator systems, from the beginning designed for portability to include single- and multi-GPU backends for the C-based OpenCL and for CUDA (which then only partly supported C++), and had thus been technically based on C++03 and on C preprocessor macros as the interface to user functions.

SkePU 2, introduced in 2016 [16], was a major revision of the SkePU [11] library, ushering in modern C++ to the skeleton programming landscape. Rebuilding the interface from the ground up, the skeleton set was updated to be variadic, leaving the old fixed-arity skeletons from SkePU 1 behind. Variadic skeleton signatures was the first main motivator of SkePU 2: *flexible* skeleton programming.

This rewrite also took the opportunity to integrate patched-on functionality in SkePU 1 into the core design of the programming model. One such example is the absorption of SkePU 1 `MapArray` into the basic SkePU 2 `Map`. `MapArray` was a dedicated skeleton in SkePU 1 created as a clone of `Map` with the ability to accept an auxiliary, random-accessible array operand into the user function, allowing deviations from the strictly functional map-style patterns when demanded by the target application. This was one of the first lessons from practical experience [23] that skeleton patterns are not always perfectly suited to algorithms in real-world application code.

SkePU 2 also introduced the *pre-compiler*, lifting SkePU from its origins as a pure template include-library into a full-fledged compiler framework.

Table 1 gives a synopsis of the different features of SkePU versions.

Table 1 Overview of SkePU Features

	SkePU 1 (2010) [11]	**SkePU 2** (2016) [16]	**SkePU 3** (2020)
API based on	C, C++ (pre-2011), C preprocessor	C++11, Precompiler	C++11, Precompiler
Skeletons	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapArray`, `MapOverlap`, `Generate`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `Call`	`Map`, `Reduce`, `Scan`, `MapReduce`, `MapOverlap`, `MapPairs`/–`Reduce`, `Call`
User functions as	C preprocessor macros	Restricted C++ functions	Restr. C++ functions, plus multi-variant user functions
Function signature	Not type-safe	Type-safe, variadic	Type-safe, variadic
User types	N/A	User structs	User structs + tuples
Containers	`Vector`, `Matrix`	`Vector`, `Matrix`	`Vector`, `Matrix`, `Tensor3`, `Tensor4`, `MatRow`, `MatCol`, `Region`
Platforms supported	CPU (C, OpenMP), GPU (CUDA, OpenCL)	CPU (C++, OpenMP), GPU (CUDA, OpenCL)	CPU, GPU, hybrid CPU/GPU StarPU-MPI, ...
Scheduling (OpenMP)	Static	Static	Static, Dynamic
Memory model	Sequential consistency	Sequential consistency	Weak consistency (default), sequential consistency

4 SkePU 3 Overview

SkePU provides data-parallel *skeletons*, all of which are of arbitrary arity and polymorphic in the operand container shape and element type. Skeleton instances accept operands that are statically grouped (by a template parameter) into element-wise accessible and random-accessible parameters [16]. The `Map` skeleton computes every result container element by element-wise application of a user function f to the corresponding (for element-wise accessed operands) or possibly any (for other operands) elements; `MapOverlap` applies a stencil function in one or several dimensions of an element-wise accessed operand that can also access elements in a limited neighborhood of the corresponding operand container element(s); `Reduce` applies reduction for a binary associative user function; `MapReduce` provides a combination of `Map` and `Reduce`; `Scan` computes generic prefix-sums for the provided binary associative user function; and `Call` simply calls the user function for each position of the output container ; in combination with multi-variant user functions it also provides a portable escape mechanism to invoke explicitly parallelized code where the available skeletons do not fit (well) [15]. The new skeletons added or generalized in SkePU 3 (cf. Table 1) are described in Sect. 5.

SkePU allows to select the backend for each skeleton instance call statically or dynamically. A tuning mechanism allows for automated backend selection depending on a call's operand sizes and locations. Hence, it might be statically unknown where a skeleton call will execute.

Operands to skeleton instances are to be passed in *data containers*, which are STL-like, generic collection abstract data types like `Vector` and `Matrix` that encapsulate C++ array-type data. We call them *smart* containers [9] because they transparently perform data transfer and memory management for their elements in (heterogeneous) systems with distributed memory, as well as global optimizations for data locality [14]. Using C++ iterators, skeleton instance calls may also operate on a proper subset of a container's elements only. New containers in SkePU 3 (see Table 1) are described in Sect. 6.

5 Skeleton Set and Interface Extensions

This section covers most of the major changes to the skeleton API in SkePU 3, with a notable exception in the multi-variant user function feature which is already detailed in [15]. We begin with `MapPairs` and `MapPairsReduce`, which are variants of `Map` resp. `MapReduce` on a 2D domain that are explicit about the access pattern of lower-dimensional (1D) operands (Fig. 1).

5.1 MapPairs

The `MapPairs` skeleton, added as an additional top-level skeleton in SkePU 3, applies a Cartesian product-style pattern from *two* `Vector<T>` sets (note that the templated type may differ across these vectors). Each vector set may contain an arbi-

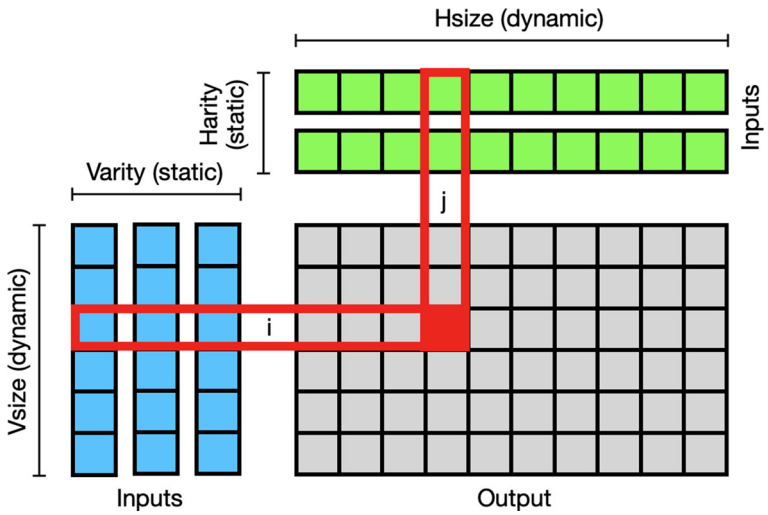

Fig. 1 `MapPairs` computes a Cartesian mapping of 1D vectors into a 2D space

Listing 1 Using the MapPairs skeleton.

```
1  // MapPairs user function
2  int mult(int a, int b) { return a * b }
3
4  // MapPairs skeleton instantiation and usage
5  // size_t Vsize, Hsize defined here
6  auto outerproduct = skepu::MapPairs(mult);
7
8  skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
9  skepu::Matrix<int> res(Vsize, Hsize);
10 outerproduct(res, v1, h1);
```

trary number of vector containers, similar to the variadicity of `Map`. All of the vectors in a set are expected to be of the same size. Each Cartesian combination of vector set indices generates one user function invocation, the result of which is an element in a `Matrix`. As in `Map`, there is an optional `Index2D` parameter in the user function signature to access this index. An example is shown in Listing 1.

Advanced and more flexible use of `MapPairs` can be carried out similarly to other SkePU skeletons. For instance, it retains flexibility of `Map` with regard to variadicity (5-way variadic, compared to `Map` being 4-way variadic): (1) resulting outputs (see Sect. 5.3), (2) element-wise-V (”vertical”, column-aligned) input arguments, (3) element-wise-H (”horizontal”, row-aligned) input arguments, (4) random-access input arguments, and (5) uniform input arguments.

A MapPairs instance of higher order would look like in the example below:

```
auto pairs = skepu::MapPairs<3, 2>(...);
```

This instance will accept three vertical and two horizontal input vectors.

Listing 2 N-body simulation code using Map.

```
auto nbody_init = skepu::Map<0>(init);
auto nbody_simulate_step = skepu::Map<1>(move);

nbody_init(particles, np);

for (size_t i = 0; i < iterations; i += 2)
{
        nbody_simulate_step(doublebuffer, particles, particles);
        nbody_simulate_step(particles, doublebuffer, doublebuffer);
}
```

Listing 3 N-body simulation code using `MapPairsReduce` in SkePU 3.

```
auto nbody_init = skepu::Map<0>(init);
auto nbody_influence = skepu::MapPairsReduce<1, 1>(influence, sum);
auto nbody_update = skepu::Map<2>(update);

nbody_init(particles, np);

for (size_t i = 0; i < iterations; ++i)
{
        nbody_influence(accel, particles, particles);
        nbody_update(particles, particles, accel);
}
```

5.2 MapPairsReduce

`MapPairsReduce` is the combination of a `MapPairs` followed by a row-wise or column-wise reduction over the generated matrix elements. Like `MapPairs` it supports arbitrary arities of the vertical and horizontal input `Vector` groups (`<0,0>` and up). It returns a `Vector` containing the row-wise or column-wise sums, where the summing dimension is specified as in 2D `Reduce`.

As a small example, we use the force calculation phase of N-body simulation using this new skeleton. In SkePU 2, using the more general `Map` skeleton, it could be written as shown in Listing 2.

In SkePU 3 (Listing 3), the use of `MapPairsReduce` eliminates the explicit loop required in the user function (`move`, omitted here for space). Moreover, this formulation intrinsically avoids the double-buffering requirement for the existing approach, and hence memory pressure is reduced.

The total number of user-function calls is now quadratic in the number of particles `np` (an inherent property of MapPairs), compared to linear before, so good performance is even more reliant on compiler optimization, in particular, user function inlining.

5.3 Multi-Valued Return in Map Skeletons

SkePU 3 introduces tuple-like return functionality for cases where a single skeleton instance requires multiple (element-wise) output containers. This way, multiple return values can be computed by the same user function, operating on the inputs in

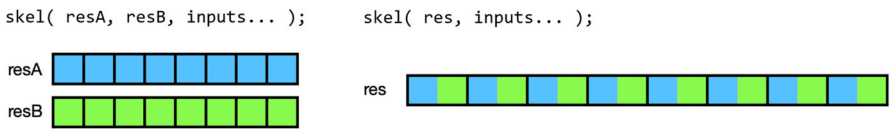

Fig. 2 Difference in return value storage between using multi-valued return (left) and single-value (by manually managed array-of-struct) return (right)

Listing 4 User function with multi-valued return.

```
skepu::multiple<int, float>
multi_f(skepu::Index1D index, int a, int b, skepu::Vec<float> c,

  int d)
{
    return skepu::ret(a * b, (float)a / b);
}
```

Listing 5 Using multi-valued return with Map in SkePU 3.

```
skepu::Vector<int> v1(size), v2(size), r1(size);
skepu::Vector<float> e(1);

auto multi = skepu::Map<2>(multi_f);

multi(r1, r2, v1, v2, e, 10);
```

one sequence, potentially improving data locality compared to two separate skeleton invocations after each other. Although the values are returned in a tuple-like manner, the output containers are completely separate objects (see Fig. 2). This distinguishes this new feature from the existing use of custom `structs` as (inputs or) return values, as those are stored in array-of-records format. To use this feature, we specify the return type in the user function signature as `skepu::multiple<[basic_type, ...]>`, i.e., analogous to `std::tuple`. Then, at the site of the `return` statement, we construct this compound object by `skepu::ret([expression, ...])`. Listing 4 shows an example of a user function utilizing the multi-valued return feature.

The skeleton instance declaration and invocation follow the syntax of ordinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order. Listing 5 gives an example.

Multi-valued return statements are available in the skeletons `Map`, `MapPairs`, `MapOverlap`, `MapReduce`, and `MapPairsReduce`.

5.4 Dynamic Scheduling with OpenMP Backends

In SkePU 2, all skeletons, in particular the `Map` based skeletons, assumed an equal load distribution of the user function executions over the entire range of input container elements. Some applications may however exhibit an irregular workload distribution

Listing 6 A MapOverlap user function in SkePU 2.

```
1  float over_2d( int oi, int oj, size_t stride, const float *r,
2                 const skepu::Mat<float> stencil )
3  {
4      float res = 0;
5      for (int i = -oi; i <= oi; ++i)
6          for (int j = -oj; j <= oj; ++j)
7              res += r[y*(int)stride+x] * stencil.data[(i+oi)*ox +
8                  (j+oj)];
9      return res;
10 }
```

instead, especially in CPU-affine computations and sometimes even in combination with very short input vectors.

For these cases, SkePU 3 adds in all skeletons (except `Scan` and `Call`) an option for dynamic scheduling in the OpenMP backend.

```
spec.setSchedulingMode( skepu::Backend::Scheduling::
Dynamic );
```

Other supported `Scheduling` modes in the OpenMP backends are `::Guided` (for guided self-scheduling), `::Auto` (for auto-tuned scheduling as implemented in the used OpenMP compiler), and of course `::Static` which remains the default scheduling mode.

In addition, a chunk size locally overriding the default chunk size (defined for each scheduling mode as in OpenMP) can be set:

```
spec.setCPUChunkSize(8);
```

Performance evaluation results for three load-balancing / nondeterministic-time benchmarks are given in Sect. 9 and show in each case some improvement over the static SkePU 2, in spite of some overhead for the dynamic scheduling modes; speedups for unbalanced workloads up to 60 % (for Mandelbrot-set computation) have been observed.

5.5 Revised Syntax for MapOverlap

Experiences from SkePU users, and in particular the application of SkePU in teaching, has showed that the syntax for `MapOverlap` user functions is one of the more challenging aspects of SkePU. In SkePU 2, a stencil operator was specified as in Listing 6 (here, for a rectangular $2oi + 1 \times 2oj + 1$ stencil implemented by 2 nested loops).

For SkePU 3 we have redesigned and simplified the programming interface for specifying stencil operators. The stencil computation from Listing 6 is shown with with SkePU 3 syntax in Listing 7.

6 New Data-Containers

The availability of smart containers, previously restricted to vector and matrix types, has a significant effect on the usability of a skeleton programming framework. Even

Listing 7 New syntax for MapOverlap user function in SkePU 3.

```
float over_2d( skepu::Region2D<float> r, const skepu::Mat<float>
stencil )
{
    float res = 0;
    for (int i = -r.oi; i <= r.oi; ++i)
        for (int j = -r.oj; j <= r.oj; ++j)
            res += r(i, j) * stencil(i + r.oi, j + r.oj);
    return res;
}
```

though a basic one-dimensional data set can be used to emulate more complex data representations, doing so at a framework level rather than on the user level provides more information to the implementation about access patterns, thus bringing increasing opportunities for optimizing communication and memory access patterns; while also providing a more intuitive user interface and reduced application code size for users.

In SkePU 3 this is recognized on two levels: new multi-dimensional *tensor* containers, as well as new "proxy" containers in user functions for accessing a single row or column from a matrix. (`RegionND<T>` is another new set of proxy container used in the new `MapOverlap` syntax.)

6.1 Tensors

The SkePU container set is extended with *tensors*, which are higher-dimensionality containers. In SkePU 3 there are tensors of three and four dimensions, complementing the existing 1D (vector) and 2D (matrix) formats. The interfaces for these containers are virtually identical to those of the other containers, differing in the obvious ways of naming and element access as detailed below. The full set of smart containers in SkePU 3 now covers up to four-dimensional structures; see Listing 8 for their definitions.

Listing 8 Smart container set in SkePU 3.

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
skepu::Tensor3<float> t3(dim1, dim2, dim3);
skepu::Tensor4<float> t4(dim1, dim2, dim3, dim4);
```

The set of `Index` object types in SkePU, usable in e.g. user function signatures to identify the index of the element being operated on, is likewise extended with 3D and 4D equivalents (see Listing 9).

Listing 9 Index types corresponding to each smart container.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

Tensors are available in the skeleton API as element-wise inputs to `Map`, `Reduce`, `MapReduce`, `Scan`, and `MapOverlap`. They are also accessible freely in user functions as proxy objects, where applicable. In some skeleton configurations the

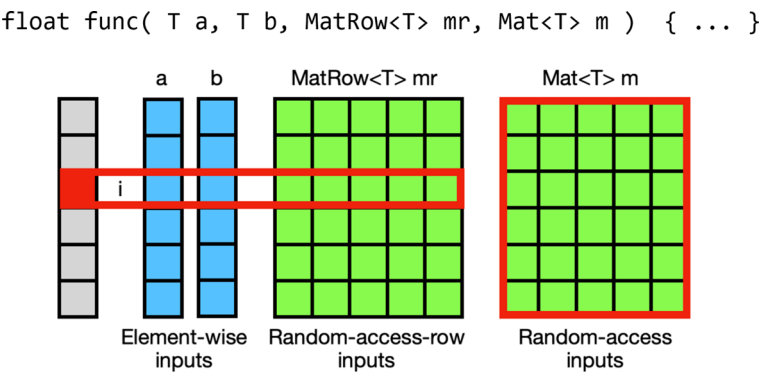

Fig. 3 Element accessibility for `MatRow` vs. `Mat` parameters in a user function

Listing 10 Matrix-vector multiply using MatRow in SkePU 3.

```
1 template<typename T>
2 T arr(const skepu::MatRow<T> mr, const skepu::Vec<T> v)
3 {
4     T res = 0;
5     for (size_t i = 0; i < v.size; ++i)
6         res += mr.data[i] * v.data[i];
7     return res;
8 }
```

dimensionality of element-wise inputs is irrelevant by design, though in `Map`-based skeletons it can be accessed by using `Index` parameters.

6.2 MatRow Container Proxy

SkePU has since version 2 offered flexible parameter lists for user functions, including *random-access* containers (implemented as lightweight *proxy objects*) in addition to the default element-wise inputs. While this allows for powerful expressivity, very little about the access patterns of these random-access containers is known to SkePU, and performance may thus not always be ideal.

One very common pattern when using `Matrix` as a random-access container parameter is that each user function invocation is only interested in one row of the matrix. This pattern is seen in matrix-vector multiplication and similar multi-reduction-style computations. To improve SkePU performance in these cases, SkePU 3 introduces a new proxy object, `MatRow<T>`. Bridging the gap between element-wise mapped and random-access container arguments, this proxy type when used in a `Map` skeleton instance[1] that maps over vectors (i.e., the result container(s) of the skeleton are `Vector`), makes available one single row of the argument matrix container to the user function, see Fig. 3.

[1] Matrix proxy arguments are available in user functions for `Map`, `MapReduce`, and `MapOverlap`.

Listing 11 Matrix-vector multiply in the SkePU 2 style.

```
template<typename T>
T arr(skepu::Index1D row, const skepu::Mat<T> m, const skepu::
 Vec<T> v)
{
    T res = 0;
    for (size_t i = 0; i < v.size; ++i)
        res += m.data[row.i * m.cols + i] * v.data[i];
    return res;
}
```

As an example, matrix-vector multiplication using `MatRow<T>` may be implemented as in Listing 10. Compared to the closest corresponding SkePU 2 implementation below (still valid in SkePU 3), which only offers the more generic `Mat<T>` proxy container, the code is more succinct and there is more information about the access pattern available to SkePU.

There is no change in syntax of skeleton instantiation or skeleton invocation needed for this feature to apply.

The performance benefit of using `MatRow<T>` (where applicable) instead of the more general `Mat<T>` container proxy comes from significantly reduced operand data transfer volume when executing over distributed memory scenarios, both in multi-GPU execution and in cluster execution: the communication pattern with `MatRow<T>` is a scatter operation, while with `Mat<T>` it is a broadcast.

7 Consistency Model

Experiences from users of SkePU 2 demonstrated that the dual-mode model of SkePU can be a bit challenging to adapt to. As in GPU programming models, SkePU programs execute code in one of two modes (in GPU programming parlance "host" and "kernel" mode). In SkePU, these are represented by being either outside or inside the *dynamic scope* of a skeleton user function. While syntactically similar, the capabilities in each mode are different. Host SkePU code is effectively like any C++ environment, as it is the goal of the framework to be possible to embed in existing C++ applications. This means that the programmer can use any C++ constructs and idioms such as classes, dynamically allocated structures, etc. Inside a user function, however, the environment is effectively a single-threaded, no-side-effects, C-like land.[2]

These differences also mean that the memory (coherency) models are different in the two views. SkePU handles memory consistency at the boundary—during entry and exit of a skeleton invocation and the user function evaluation. Inside the user function, side effects are not allowed and therefore random memory reads are disabled, and the coherency model is straightforward.

[2] The reason for this is to preserve compatibility with as many accelerator environments as possible, such as OpenCL C or even FPGAs.

Listing 12 Examples of using the flush operation.

```
skepu::Vector<int> v1(n), v2(n);
skepu::Matrix<int> m1(n, n), m2(n, n);

v1.flush(); // FlushMode::Default
m1.flush(); // FlushMode::Default

skepu::flush(v2, m2); // FlushMode::Default

v1.flush(skepu::FlushMode::Dealloc);
m1.flush(skepu::FlushMode::Dealloc);
skepu::flush<skepu::FlushMode::Dealloc>(v2, m2);
```

SkePU 3 *deprecates* the angle bracket `[ ]`-notation for smart container element read/write access *outside* user functions.[3] This is part of a simplification of the coherency mechanisms for manual element access from the host (CPU) side. Instead, the programmer should *flush* the whole container instead before doing single-element accesses of user function data, see below.

Instead of angle brackets, the parentheses `( )`-notation is extended to higher dimensionality. This syntax accepts one index argument for each dimension of the underlying container. The indices count must equal container dimensionality, otherwise there is a compile-time error. Formally, the access syntax is `container(i, [, j, [,k, [, l]]])` `[=` *value*];

Hence, there is no longer a coherency-satisfying single-element access mechanism to SkePU smart containers except inside user function proxy objects (`Vec<T>`, `Mat<T>`, etc). However, optional runtime checks outside user functions can be (re-)activated for parenthesis accesses by setting a compiler flag, e.g., for debugging purposes or for backwards compatibility with code written for SkePU 2.

A common pattern in SkePU applications is that smart containers are used for a computationally intensive part of the application, and the data is then either handed over to a non-SkePUized section, or serialized e.g. to disk. To accommodate this pattern, it is important that there is a way to ensure consistency of the local container contents. SkePU 3 provides this through the `flush` operation to complete the new consistency model.

Flushing smart container data can be performed on smart container instances or collectively by a variadic free function. Either approach accepts a flush mode enum argument providing options, e.g. if the remote data buffers should be cleaned up or not, as seen in Listing 12.

The `flush` (member) functions are known symbols to the precompiler, so the presence or absence of flush operations in SkePU source code is subject to static analysis and optimization.

[3] In SkePU 2 (and SkePU 1), the bracket operator is a protected container access, which outside user functions checks for the accessed element's state in the data container's metadata (updated or invalid) and, if necessary, triggers a (bulk) data movement to update the container's copy in host memory from a currently valid device copy. All bracket accesses thus incur runtime overhead for the check.

8 Cluster Backend

SkePU 3 provides two different modes of using cluster resources:

- *Outer MPI mode*: the application code already contains explicit MPI code for cluster-level parallel execution, using SkePU only locally on each node for execution of skeletons on multicore CPU and/or accelerators.
- *Inner MPI mode*: The application does not contain any MPI (nor other parallelization) code. If an environment for MPI parallel execution is available (usually, multiple nodes on a cluster), then skeletons can transparently execute in parallel across these nodes if selecting the MPI backend.

Outer and Inner MPI mode are mutually exclusive, i.e., for applications that are pre-parallelized using explicit MPI code the MPI backends of all skeletons are disabled.

The implementation of inner MPI parallelism is technically based on generating StarPU task code using the MPI interface of the StarPU runtime system [3], which detaches each node's generated send and receive operations into special CPU "codelets" that are exposed to StarPU as separate tasks for dynamic scheduling [2]. Distributed variants of the smart data-containers (`Vector`, `Matrix` etc.) with the same interface as the node-local counterparts come with default distributions, and each cluster node runs one copy of the SkePU executable atop a local instance of StarPU in SPMD style. Execution over distributed container operands follows the "owner computes rule", stating that each node only executes those operations that calculate (write) elements it owns (i.e., are part of its local partition of the result container).

For use of inner MPI parallelism, no syntactic changes in SkePU code are required, thus following SkePU's strict portability principle. The illusion of a single SkePU process performing all the work on a single node even with the MPI backend is maintained by implementing the `Reduce` skeleton by an MPI Allreduce operation so that the reduction result is available on each of the SPMD processes. The weak memory consistency model of SkePU 3 (cf. Sect. 7) applies also to distributed containers: the programmer must explicitly `flush` (i.e., gather) them back to the master (i.e., the rank 0 process) before the most recent values of elements of remote partitions can be accessed by a read access on the master, or after a write access by the master.

The only remaining issue in SPMD execution is that I/O operations need be protected from being executed everywhere. To make sure that such code is executed only by the SPMD master process (and distributed data to be output is automatically flushed and gathered/scattered to/from before/after the access, respectively), such code should be guarded by the new construct

```
skepu::external (
  [ skepu::read(rdcontlist),] [&]() {
    ...
  } [,  skepu::write(wrcontlist)]
);
```

where the optional arguments `skepu::read()` and `skepu::write()` list container objects that may be read from resp. written to main memory in the framed code block (`...`). This semi-automatic solution with an explicit framing construct allows

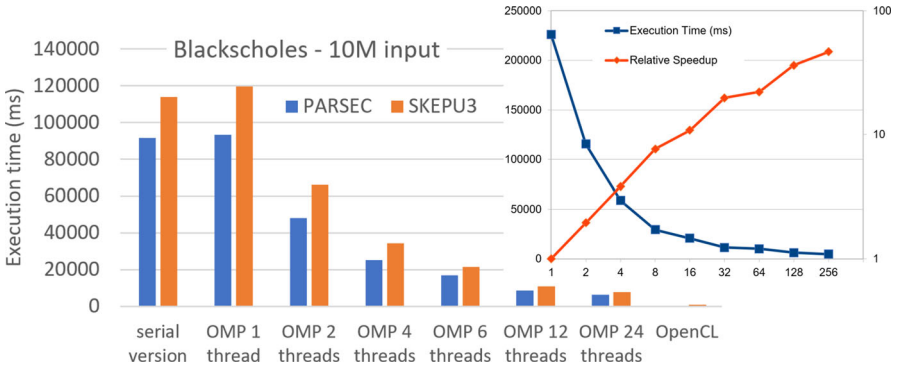

Fig. 4 Execution time (ms) of the SkePU 3 port of the PARSEC benchmark Blackscholes on its largest input set. Left: Time on the local GPU server with serial, OpenMP, OpenCL backends in SkePU and for manually multithreaded code in PARSEC. Right: Time and speedup on the Tetralith cluster with the StarPU-MPI backend

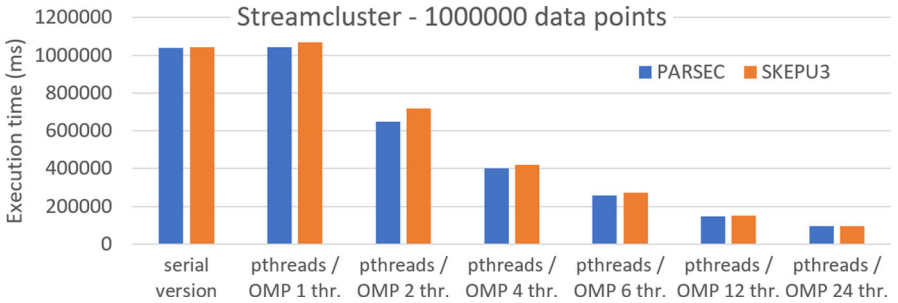

Fig. 5 Execution times of the SkePU 3 port of the PARSEC benchmark Streamcluster on 10^6 data points, on the local server

to not depend on static analysis by the precompiler, which may not be feasible in the context of separate compilation and using libraries.

9 Performance evaluation

This section presents some performance evaluation of new SkePU 3 features; further performance results can be found in Ernstsson's licentiate thesis [13], such as an embedded ODE solver from the *Libsolve* library [18], solving the Brusselator 2D-MIX problem with 7 stage vectors, performing up to 124,532 calls to skeleton instances in total for the largest problem size, which we have omitted here due to lack of space.

We use two different systems for performance evaluation: a local GPU server with 12 cores (two Xeon E5-2630L CPUs with two-way hardware multi-threading), a K20c GPU, and 64 GiB of main memory; and the *Tetralith* supercomputer cluster located at NSC Linköping. Each Tetralith node has two Xeon Gold 6130 processors with 16 cores each, and a total of 96 GiB of memory per node.

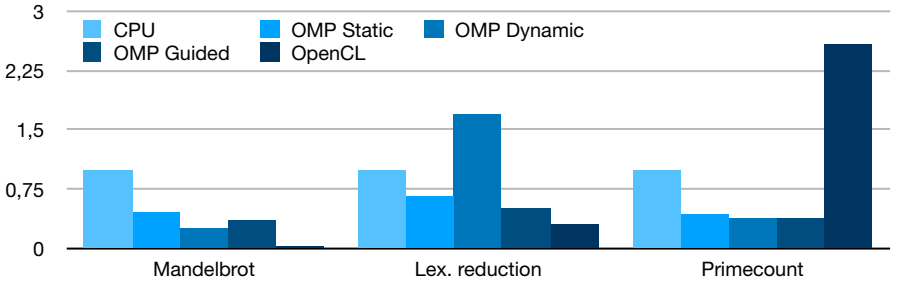

Fig. 6 Execution time (normalized to the sequential CPU backend time) for three irregular-load benchmarks on the local server

Execution time results for SkePU 3 ports of PARSEC benchmarks Blackscholes and Streamcluster can be found in Figs. 4 and 5. The results show that the SkePU abstraction overhead compared to the hand-multithreaded PARSEC code is small (Blackscholes) or very small (Streamcluster), and that SkePU provides further targets for free (here, OpenCL for Blackscholes). The Streamcluster benchmark also exhibits a common problem encountered in SkePU-izing legacy C/C++ code: arrays containing a pointer-based data structure (e.g., a directed graph), if packaged e.g. in a `Vector` container, work very well with the OpenMP backend but are not portable to execution on e.g. a GPU with a different address space, as host addresses are not portable to device memory. For such cases, more advanced container types (e.g., directed graphs) would be required, which is left for future work.

For the local server, Fig. 6 shows the positive performance effect of using dynamic scheduling in three data-parallel benchmarks with irregular workload, in spite of the runtime overhead of dynamic scheduling: (1) Generating a 1024×1024 Mandelbrot image using the SkePU 3 `Map` OpenMP backend with different scheduling modes. Dynamic scheduling (chunksize 16) outperforms the static default mode. (2) Lexicographic reduction finding the maximum among 10^8 date/time tuples. `Guided` dynamic scheduling (chunksize 8) outperforms the static default mode. (3) Counting prime numbers using `MapReduce` where dynamic scheduling performs best. Results for the sequential CPU and OpenCL backends are provided as reference.

Figure 7 shows the scaling behavior of the SkePU 3 port of a brain simulation mini-application [21] performing 200 time steps with 90000 neurons and dense synapse connectivity using up to 32 nodes on Tetralith. The version that uses the `MatRow` container proxy benefits from more scalable communication compared to using the default `Mat` container. For comparison, the diagram also shows an *Outer MPI* SkePU implementation where the communication structure corresponds to that of the `MatRow` version. While the SkePU version of `MatRow` scales and performs best for larger numbers of MPI ranks, we also see that there is an execution time overhead of using SkePU with the StarPU-MPI based backend of up to a factor of 2 as long as running on a single cluster node ($\leq$ 32 MPI ranks).

Figure 8 shows performance results for the N-body scenario of Sect. 5.2 using both the OpenMP and cluster backend, measured on the Tetralith cluster. There is a slight increase in execution time for the `MapPairsReduce` variant on a single

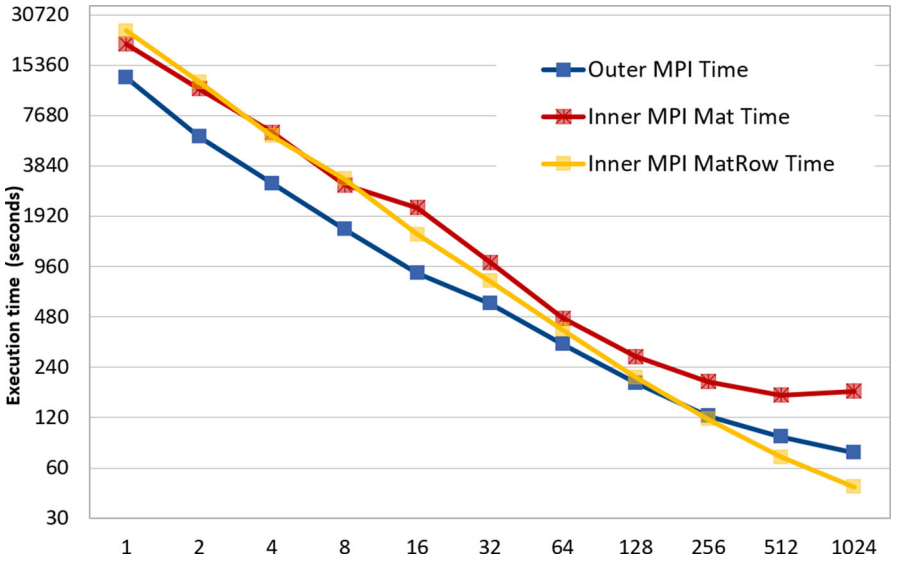

Fig. 7 Execution time of a brain simulation application performing 200 time steps with 90000 neurons on Tetralith. "Outer MPI" refers to a manual MPI parallelization using (node-local) SkePU and plain MPI, the two "Inner-MPI" versions use SkePU's StarPU-MPI backend instead

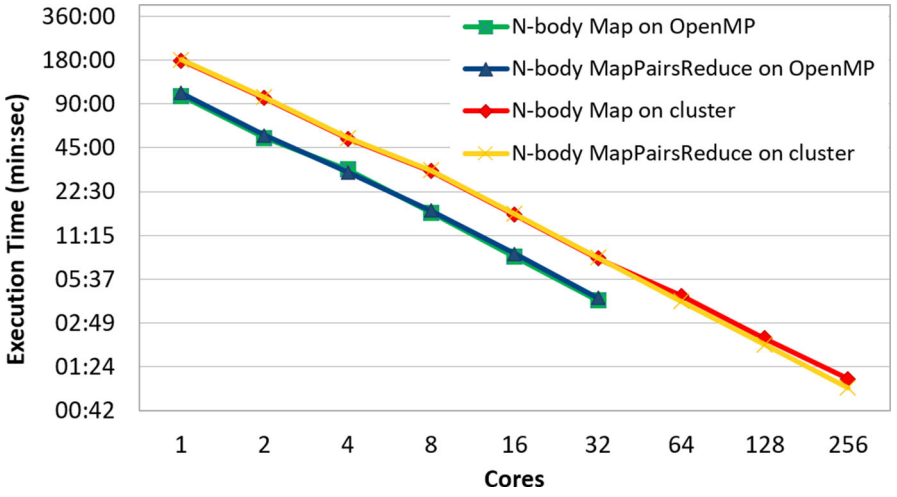

Fig. 8 Execution time for two variants of N-body on the Tetralith cluster: the `Map` variant (Listing 2) and `MapPairsReduce` variant (Listing 3), both executed with the OpenMP and cluster backends

node, although too small to account for an inlining issue (discussed in Sect. 5.2). A likely explanation for the slowdown is due to the change in memory access pattern. Depending on the environment, the more significant improvement in memory footprint might be enough to prefer the `MapPairsReduce` variant, as it uses only 70% of the memory compared to the `Map` variant. On multi-node experiments, the variant

Table 2 Microbenchmark results of vector initialization; execution time in seconds on the local GPU server

	With GPU backends	Without GPU backends
Sequential consistency `v[i]`	0.899	0.308
Weak consistency `v(i)`	0.313	0.310

advantage is flipped in favour of the `MapPairsReduce` variant, which is about 15% faster on 8 nodes (256 cores); we ascribe that to the reduced communication volume.

To illustrate the motivation behind the change of consistency model for SkePU smart containers (Sect. 7), we have measured the execution time through a microbenchmark. Allocating and initializing the elements of a SkePU vector using a simple for-loop results in the numbers in Table 2. If the SkePU application is compiled without either GPU or CUDA backends there is no appreciable overhead, but as soon as those device copies are present it is approximately 3x faster to use non-managed access operators.

10 Related work

The skeleton approach to high-level programming of parallel systems has been introduced by Cole in 1989 [5,6]. Since then, many academic skeleton programming frameworks have been presented, and the concept also increasingly found its way into commercial/industrial-strength programming environments, such as Intel TBB for multicore CPU parallelism, Nvidia Thrust or Khronos SYCL for GPU parallelism, or Google MapReduce and Apache Spark for cluster-level parallelism over huge data sets in distributed files.

While early skeleton programming environments attempted to define and implement their own programming language, library-based and DSL-based approaches have, by and large, been more successful, due to fewer dependencies and lower implementation effort. Frameworks for skeleton programming became practically most effective in combination with (modern) C++ as base language. Moreover, the approach was fueled by the increasing diversity of processing hardware with upcoming multicore and heterogeneous parallelism since about 2005.

Among the C++-based skeleton programming environments for heterogeneous systems, we find mostly library-based ones, e.g. SkePU 1 [11], SkeTo [20], SkelCL [25], GrPPI [10] or pre-compiler based, such as SkePU 2 [16] and Musket [22]. FastFlow [7], originally designed for multicore CPU execution, added support for GPU and distributed execution [1] later. SkelCL targeted OpenCL for single- and multi-GPU systems with explicit data distribution. Muesli [4] initially supported MPI cluster execution and added support for GPU execution later [12]. MPI execution of skeletons is also supported e.g. in Musket.

The Allpairs skeleton [24] in SkelCL can be considered as a variant of `MapPairs` that accepts matrix operands only; any reduction needs be implemented as part of the user function in Allpairs (i.e., by nesting), while we provide the combination `MapPairsReduce` (i.e., chaining). `MapPairs` and `MapPairsReduce` specifi-

cally support *multiple* separate *1D* vector operands in both dimensions, as requested for use with the *MetalWalls* [19] application by EXA2PRO project partner CNRS.

Multiple return values of skeletons and user functions is inspired by Python and was also found useful in SkePU-izing *MetalWalls* to avoid duplicated work resulting from using multiple skeleton calls for different result containers. We are not aware of any other skeleton programming framework supporting multiple return values from skeletons and user functions where parameters are passed explicitly. Musket [22], which also uses a precompiler, requires (except for the `this` object) using global container variables in user functions.

11 Conclusions and future work

We have presented the design of the third generation of the SkePU skeleton programming framework for heterogeneous systems and HPC clusters. The new features are the result of a co-design effort together with HPC application partners in the EXA2PRO project, and are geared towards improving programmability, flexibility, better performance, or several of these aspects, while keeping SkePU source code strictly portable and compatible with sequential C++11. We provide a sample of results from performance evaluation experiments; further results can be found in Ernstsson's licentiate thesis [13], which also goes into more detail on the design and implementation approach behind SkePU 3.

Future work will, beyond performance improvements in the implementation, conduct performance studies on the four main EXA2PRO applications being ported to SkePU 3, and further extend the set of SkePU 3 example programs. A survey of how SkePU 3 embeds in the EXA2PRO high-level programming model can be found in [17]. SkePU 3 has already been made interoperable with multi-variant functions ("components") [17], which provide a flexible escape mechanism for expressing parallelizations of computations where no skeleton fits (well) or for using accelerator types for which no appropriate SkePU backend is available (yet). Moreover, the new alternative pre-compiler being developed for SkePU 3 (which is technically based on the BSC Mercurium compiler) will allow for static transformations of skeleton groups, which is for now only supported to a limited degree as a runtime optimization for special skeleton sequences [14].

The SkePU 3 source code (with the clang-based precompiler) is publically available under a modified 4-clause BSD license at https://skepu.github.io.

Acknowledgements This work has been partly funded by EU H2020 project EXA2PRO (801015) and by the Swedish National Graduate School in Computer Science (CUGS). We thank all project partners in EXA2PRO for feedback that led to the design of SkePU 3. We also thank the National Supercomputing Centre (NSC) and SNIC for access to their HPC computing resources (SNIC 2016/5-6). Open access funding has been provided by Linköping University.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If

material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in FastFlow. In: Euro-Par 2012: Parallel Processing Workshops. LNCS 7640, pp. 47–56. Springer, Berlin (2013)
2. Augonnet C, Aumage O, Furmento N, Namyst R, Thibault S: StarPU-MPI: Task Programming Over Clusters of Machines Enhanced with Accelerators. In: Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, Recent Advances in the Message Passing Interface, pages 298–299. Springer, Berlin (2012)
3. Augonnet, Cédric., Thibault, Samuel, Namyst, Raymond, Wacrenier, Pierre-André.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
4. Ciechanowicz P, Poldner M, Kuchen H: The Münster Skeleton Library Muesli - a Comprehensive Overview. ERCIS Working Paper No. 7 (2009)
5. Cole, Murray: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parall. Comput **30**(3), 389–406 (2004)
6. Cole, Murray I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman and MIT Press, Cambridge, Mass (1989)
7. Danelutto, Marco: Torquati, Massimo: Structured Parallel Programming with "core" FastFlow. Central European Functional Programming School. volume 8606 of LNCS, pp. 29–75. Springer, Berlin (2015)
8. Dastgeer, Usman, Enmyren, Johan, Kessler, Christoph W.: Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In *Proc. 4th International Workshop on Multicore Software Engineering*, pages 25–32. ACM, (2011)
9. Dastgeer, U., Kessler, C.: Smart containers and skeleton programming for GPU-based systems. Int J Parall Programm **44**(3), 506–530 (2016)
10. del Rio Astorga, David, Dolz, Manuel F., Fernández, Javier, Daniel García, J.: A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, (2017)
11. Enmyren, Johan, Kessler, Christoph W.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, (2010)
12. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int J of High Perf Comput Netw **7**, 129–138 (2012)
13. Ernstsson, August: Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems. LinkÖping University Electronic Press, Sweden (2020)
14. Ernstsson, A., Kessler, C.: Extending smart containers for data locality-aware skeleton programming. Concurrency and Computation: Practice and Experience **31**(5), e5003 (2019)
15. Ernstsson, August, Kessler, Christoph: Multi-variant user functions for platform-aware skeleton programming. In *Proc. of ParCo-2019 conference, Prague, Sep. 2019, In: I. Foster et al. (Eds.), Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36, IOS press*, pages 475–484, (2020)
16. Ernstsson, August, Li, Lu, Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Int J Parall Program **46**, 1–19 (2017)
17. Kessler, Christoph, Ernstsson, August, Memeti, Suejb, Ahlqvist, Johan: Embracing heterogeneity for exascale computing: The EXA2PRO high-level programming model. Proc. Work-in-progress session at PDP'20 conference, Västerås, Sweden, Report SEA-SR-55-4, Johannes-Kepler Univ. Linz, Austria, (2020) ISBN 978-3-902457-55-4
18. Korch, Matthias, Rauber, Thomas: Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining. J. Parall Distribut Comput **66**(3), 444–468 (2006)
19. Marin-Laflèche, Abel, Haefele, Matthieu, Scalfi, Laura, Coretti, Alessandro, Dufils, Thomas, Jeanmairet, Guillaume, Reed, Stewart K., Serva, Alessandra, Berthin, Roxanne, Bacon, Camille, Bonella,

Sara, Rotenberg, Benjamin, Madden, Paul A., Salanne, Mathieu: Metalwalls: A classical molecular dynamics software dedicated to the simulation of electrochemical systems. Journal of Open Source Software **5**(53), 2373 (2020)
20. Matsuzaki, Kiminori, Emoto, Kento: Implementing fusion-equipped parallel skeletons by expression templates. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages*, pages 72–89. Springer, Sweden (2010)
21. Panagiotou, Sotirios, Ernstsson, August, Ahlqvist, Johan, Papadopoulos, Lazaros, Kessler, Christoph, Soudris, Dimitrios: Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU. In *Proc. SCOPES'20*. ACM, (2020)
22. Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proc. Symposium on Applied Computing (SAC'19)*, pages 1534–1543. ACM, (2019)
23. Oskar Sjöström, Soon-Heum Ko, Usman Dastgeer, Lu Li, and Christoph Kessler. Portable parallelization of the EDGE CFD application for GPU-based systems using the SkePU skeleton programming library. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer, editors, *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proc. of ParCo-2015 conference, Edinburgh, UK, Sep. 2015.*, pages 135–144. IOS Press, (2016)
24. Steuwer, Michel: Friese, Malte, Albers, Sebastian, Gorlatch, Sergei: Introducing and implementing the AllPairs skeleton for programming multi-GPU systems. Int J Parall Program **42**(4), 601–618 (2013)
25. Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL–a portable skeleton library for high-level GPU programming. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, (2011)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.