EXA2PRO High-Level Programming Framework User Guide

August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula,
Suejb Memeti, Christoph Kessler

Linköping University

September 7, 2021

Contents

Chapter 1

SkePU User Guide

Revision history

- 2019-09-25
  First Exa2Pro release.
- 2019-11-20
  Second Exa2Pro release. Revised installation instructions.
  Updated SkePU 3 changes.
- 2020-01-29
  Third Exa2Pro release. Updated to reflect changes in SkePU API and implementation progress.
- 2020-04-29
  Fourth Exa2Pro release. Updated with new skeleton MapPairsReduce and changes to backend selection process.
- 2020-05-20
  Separate document for public SkePU 3 user guide. Refactored changelog into the user guide proper.

1.1 Introduction

This chapter gives a high-level introduction to programming with SkePU. The version of SkePU documented here is a developement release of SkePU 3, a source-breaking update from SkePU 2. SkePU is a skeleton programming framework for multicore and multi-GPU systems with a C++11 interface. It includes data-parallel skeletons such as Map and Reduce generalized to a flexible programming interface. SkePU emphasizes and improves on flexibility, type-safety and syntactic clarity over its predecessor, while retaining efficient parallel algorithms and smart data movement for hgh-performance and energy-efficient computation.

SkePU is structured around a source-to-source translator (precompiler) built on top of Clang libraries, and thus requires the LLVM and Clang source when building the compiler driver.

All user-facing types and functions in the SkePU API are defined in the `skepu` namespace. Nested namespaces are not part of the API and should be considered implementation-specific. The `skepu::` qualifier is implicit for all symbols in this document.

1.2 License

SkePU is distributed as open source and licensed under a modified BSD 4-clause licence. The copyright belongs to the individual contributors.

1.3 Authors and Maintainers

The original SkePU was created by Johan Enmyren and Christoph Kessler [5]. A number of people has contributed to SkePU, including Usman Dastgeer.

The major revision SkePU 2 was designed by August Ernstsson, Lu Li and Christoph Kessler [6].

The major revision towards SkePU 3 was designed by August Ernstsson, Christoph Kessler, Johan Ahlqvist, and Suejb Memeti with input from partners in the EXA2PRO project. [6].

August Ernstsson[1] is the current maintainer of SkePU.

1.3.1 Acknowledgements

This work was partly funded by EU H2020 project EXA2PRO, by the EU FP7 projects PEPPHER and EXCESS, by SeRC project OpCoReS, by the Swedish national graduate school in computer science (CUGS), and by Linköping University.

We also acknowledge the National Supercomputer Centre (NSC) and SNIC for providing access to HPC cluster systems used for performance testing (SNIC 2016/5-6).

1.4 Dependencies and Requirements

SkePU is fundamentally structured around C++11 features and thus requires a mature C++11 compiler. It has been tested with relatively recent versions of Clang and GCC, and NVCC version 9.

It also uses the STL, including C++11 additions. It has been tested with `libstdc++` and `libc++`. SkePU does not depend on other libraries.

SkePU requires the LLVM and Clang source when building the source-to-source translator. The translator produces valid C++11, OpenCL and/or CUDA source code and can thus be used on a separate system than the target if necessary (”cross-precompilation”).

The StarPU MPI backend requires a recent GCC compiler, an OpenMP library, an MPI library (tested with OpenMPI version 2.1), and StarPU built from the master branch.

[1]august.ernstsson@liu.se

1.5 Example

We will introduce the SkePU syntax with an example.

Listing 1.1: Example SkePU 3 userfunction: A linear congruential generator.

```
#include <iostream>
#include <cmath>
#include <skepu>

// Unary user function
float square(float a)
{
        return a * a;
}

// Binary user function
float mult(float a, float b)
{
        return a * b;
}

// User function template
template<typename T>
T plus(T a, T b)
{
        return a + b;
}

// Function computing PPMCC
float ppmcc(skepu::Vector<float> &x, skepu::Vector<float> &y)
{
        // Instance of Reduce skeleton
        auto sum = skepu::Reduce(plus<float>);

        // Instance of MapReduce skeleton
        auto sumSquare = skepu::MapReduce<1>(square, plus<float>);

        // Instance with lambda syntax
        auto dotProduct = skepu::MapReduce<2>(
                [] (float a, float b) { return a * b; },
                [] (float a, float b) { return a + b; }
        );

        size_t N = x.size();
        float sumX = sum(x);
        float sumY = sum(y);

        return (N * dotProduct(x, y) - sumX * sumY)
                / sqrt((N * sumSquare(x) - pow(sumX, 2)) * (N * sumSquare(y) - pow(sumY, 2)))
                    ;
}

int main()
{
        const size_t size = 100;

        // Vector operands
        skepu::Vector<float> x(size), y(size);
        x.randomize(1, 3);
        y.randomize(2, 4);

        std::cout << "X:␣" << x << "\n";
        std::cout << "Y:␣" << y << "\n";

        float res = ppmcc(x, y);

        std::cout << "res:␣" << res << "\n";

        return 0;
}
```

1.6 Installation

The installation chapter is divided into two parts. In Section 1.6.1 we will look at building and installing the clang-based version of SkePU tool, and in Section 1.6.2 we will look at the build and installation process of the Mercurium-based version.

The installation and use process for SkePU is still in a prototype state.

1.6.1 Clang based skepu-tool

There are three steps to do when building skepu-tool from source:

- Getting the source
- Build skepu-tool
- Install skepu-tool

Getting the source

The source code is available at the Exa2Pro repository at `https://gitlab.seis.iti.gr/exa2pro/skepu-clang`. The repository uses submodules for it's in source dependencies on other projects. Either use `git clone --recursive` to clone the submodules at the same time as the SkePU repository, or `git submodule update --init` to fetch the submodules within an already cloned SkePU repository.

Building skepu-tool

New for skepu-tool is the move from a Makefile to a CMake based build procedure. The CMake scripts requires CMake version 3.13 or later. Best practice is to create an out of source build folder. In the following code snippet, we will use <src>/build. The following commands will build skepu-tool:

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

A couple of notable build options:

Option name	Default value	Description
SKEPU_ENABLE_TESTING	OFF (Release) ON (Debug)	Enables the test suite for skepu-tool.
SKEPU_BUILD_EXAMPLES	OFF (Release) ON (Debug)	Enables building skepu examples

For more build options, run `cmake -LAH`.

1.6.2 Mercurium based SkePU tool

Dependencies

- GCC version 6
- NVIDIA CUDA toolkit

Build dependencies

- GCC (including gfortran)
- NVIDIA CUDA toolkit
- bison
- autoconf
- automake
- autotools
- flex
- gperf
- libtool
- pkg-config
- python3
- sqlite3

Getting the source code

A tarball is available in the repo x2p_programming_framework in the exa2pro group [2].

Building and installation

Enter the directory skepu-mcxx/skepu and run the build_skepu script. The build script uses the following environment variables to control the build process:

Name	Value
GPP	The path to the GCC prerocessor
GCC	The path to the GCC C compierl
GXX	The path to the GCC C++ compiler
GFORTRAN	The path to the GCC fortran compiler
NVCC	The path to the NVIDIA CUDA compiler

When the build process is complete, there will a directory called skepu containing SkePU tool and the SkePU header files. Install the contents of the skepu folder the desired place. The last step in the install precess is to update the environment path to include <install path>/bin.

1.7 Usage

The source-to-source translator tool `skepu-tool` accepts as arguments:

- input file path: `-name <filename>`,
- output directory: `-dir <directory>`,
- output file name: `<filename>` (without file extension),

[2] `https://gitlab.seis.iti.gr/exa2pro/x2p_programming_framework`

- any combination of backends to be generated: `-cuda -opencl -openmp`.

The Mercurium based skepu-tool also supports the flag `--compile` which will generate a binary instead of a precompiled file. The host compiler for the precompiled source will be either GXX or NVCC, which can be set as environment variables to control which host compiler to use. By default the script uses nvcc for cuda codes, and gcc version 6 otherwise.

A complete list of supported flags, and further instructions, can be found by running `skepu-tool -help` on the clang based skepu-tool.

Note that code for the sequential backend is always generated.

SkePU programs (source files) are written as if a sequential implementation—without source translation—was targeted. In fact, such an implementation exists and is automatically selected if non-transformed source files are compiled directly. Make sure to `#include` header `skepu`, which contains all of the SkePU library[3].

Please look at the included example programs and Makefiles to get an idea of how everything works in practice.

Include directories

The Clang based `skepu-tool` uses Clang libraries and will perform an actual parse to be able to properly analyze and transform the source code; still, it is not a fully-featured compiler as you would get with a pre-configured package of, e.g., Clang or GCC. This has consequences when it comes to locating platform and system-specific include directories, as these have to be specified explicitly.

By adding the `--` token to the arguments list, you signal that any remaining arguments should be passed directly to the underlying Clang engine. These arguments are formatted as standard Clang arguments. The required arguments are as follows:

- `-std=c++11`;
- include path to Clang's compiler-specific C++ headers,
  `-I <path_to_skepu>/lib/SkePU/clang-headers`, where the path is the root of the Clang sources (typically in the `tools` directory in the LLVM tree);
- include path to the SkePU source tree: `-I <path_to_skepu>/include`;
- include path(s) to the C++ standard library, platform-specific;
- additional flags as necessary for the particular application, as if it was being compiled.

The Mercurium based clang-tool handles the include paths to skepu on it's own as long as the relative path from the binary to the headers remains the same as is generated by the build script (<install_path>/include).

Debugging

Standard debuggers can be used with SkePU. Per default, SkePU does not use or require exceptions, and reports internal fatal errors to `stderr` and terminates. For facilitating debugging, defining the `SKEPU_ENABLE_EXCEPTIONS` macro will instead cause SkePU to report these errors by throwing exceptions. This should *not* be used for error recovery in release builds, as the internal state of SkePU is not consistent after an error. (The types of errors reported this way are mostly related to GPU management.)

[3]Almost everything in SkePU is templates, so there is no penalty from including skeletons etc., which are not used.

1.7.1 StarPU MPI

To use the StarPU MPI backend, precompile the source with the flags `-openmp -starpu-mpi`. If only the Map skeleton is used in a program, one can also enable the cuda backend. Do not forget to add the link flags and include flags that is needed to compile StarPU codes.

When using the SkePU StarPU MPI backend, do not forget to make sure the code is safe to execute on multiple ranks at the same time. The `skepu::external` function can be used to make a region of code safe. Writing to file is one example where multiple nodes cannot execute the same region at the same time.

1.7.2 Clang skepu-tool and CMake

The clang skepu-tool offers a CMake function to automatically configure the precompilation step. The syntax is as follows:

```
skepu_add_executable(<name> [EXCLUDE_FROM_ALL]
  [[[CUDA] [OpenCL] [OpenMP]] | [MPI]]
  SKEPUSRC ssrc1 [ssrc2 ...]
  [SRC src1 [src2 ...]])
```

The function is a wrapper around `add_executable` that will generate precompilation targets for the SkePU sources listed as argument. Any include directories added via `target_include_directory` or `target_link_library` will propagate to the precompilation targets as well. The MPI backend option to `skepu_add_executable` implies OpenMP.

To be able to use the function, add `find_package(SkePU)` to the CMakeLists.txt file. If SkePU is not installed in a path that CMake is aware of, one adds the argument `HINTS <skepu_prefix>` to find_package.

1.8 Limitations

The following section details limitations as of 2020-05-20.

1.8.1 SkePU general

Known issues with the SkePU headers:

- MapOverlap (all) code generation is disabled for OpenCL (Section 1.10.5).
- MapOverlap (3D, 4D) code generation is disabled for CUDA (Section 1.10.5).
- MapPairsReduce code generation is disabled for CUDA and OpenCL (Section 1.10.7).
- Multi-valued return (Section 1.11.1) is not implemted for MapOverlap.
- Multi-valued return is disabled for CUDA and OpenCL.
- Scan is missing OpenMP backend selection parameters (thread count, scheduling mode).

1.8.2 StarPU-MPI skeleton backends

The StarPU MPI backend is not very well tested yet. The following list are known issues:

- The containers are not be fully compatible with the normal version of SkePU.
- The skeleton Call is missing.
- Only the Map, MapPairs, and Reduce skeletons has support for CUDA and applications that use other skeletons cannot have CUDA and StarPU-MPI enabled at the same time.

1.8.3 Clang SkePU tool

In addition, not all combinations of skeleton features are implemented/tested for this release.

1.8.4 Mercurium SkePU tool

This is a young implementation of SkePU tool, and it comes with the following caveats.

- Haing more than one lambda userfunction argument to a skeleton.
- No help is available for the skepu-tool script.

1.9 Definitions

Please read through this section once to familiarize yourself with the terms used in this document. It can then be used as a reference, as the terms defined here are typeset in *italics* at first mention in each section.

Skeleton A computation structure on *container*s, e.g. map or reduce. The skeletons in SkePU 2 are all data-parallel, i.e., the computation graph is directed by the structure of container parameters and not dependent on the value of individual elements in a container.

Container An object of some SkePU container class, i.e., vector or matrix. Homogenous; contains objects of a single *scalar* type. In this document, the term container refers exclusively to SkePU containers (as opposed to, e.g., raw data pointers or STL vectors).

Scalar The type of elements in a *container.* May be a fundamental type such as `float`, `double` or `int` or a compound struct type satisfying certain rules. (Note that the compound types are still refered to as scalar types when in containers.)

User function An operation performed repeatedly (perhaps in parallel) in a *skeleton instance.* A user function in SkePU should not contain side effects, with the exeption of writing to *random access arguments.*

Skeleton instance An object of some skeleton type instantiated with one or more *user functions.* May include state such as

- *backend specification*,
- *execution plan*, and
- *skeleton*-specific parameters such as the starting value for a reduction.

Skeleton invocation The process of applying a *skeleton instace* to a set of parameters. Performs some computation as specified by the instance's *skeleton* type and *user function.*

Output argument For the *skeletons* which return a *container*, this container is passed as the first argument in a *skeleton invocation.* If the skeleton instead returns a *scalar*, no argument is passed and the value is instead the evaluated value of the invocation expression (i.e., the return value).

Element-wise parameter/argument A *container* argument to a *skeleton instace*, elements of which, during *skeleton invocation*, is passed to the corresponding *user function* parameter as a scalar value. Iterators into containers can also be used for these parameters, with

Random access parameter/argument A *container* argument to a *skeleton instace*, which, during *skeleton invocation*, is passed to the corresponding *user function* parameter and av.

Uniform parameter/argument A *scalar* argument to a *skeleton invocation*, passed unaltered to each user function call.

Backend The compute units and/or programming interface to use when executing a skeleton

Backend specification An object of type `BackendSpec`. Encodes a *backend* (e.g., OpenMP) along with backend-specific parameters for execution (e.g., number of threads) for use by a *skeleton instance*. Overrides *execution plan*s when selecting backends.

Tuning The process of training a *skeleton instance* on differently sized input data to determine the optimal *backend* in each case.

Execution plan Generated during *tuning* and stored in a *skeleton instance*. Helps select the proper *backend* for a certain input size.

Source-to-source translator / precompiler (`skepu-tool`) Clang-based tool which transforms SkePU programs for parallel execution. Accepts C++11 code as input and produces C++11/CUDA/OpenCL/OpenMP code as output. Built by user from Clang sources, patched with SkePU-provided extensions.

Host compiler User-provided C++11/CUDA compiler which performs the final build of a SkePU program, producing an executable. Can also be used on raw (non-precompiled) SkePU source for a sequential executable.

1.10 Skeletons

SkePU (3) encompasses currently eight different skeletons:

- `Map`,
- `Reduce`,
- `MapReduce`,
- `Scan`,
- `MapOverlap`,
- `MapPairs`,
- `MapPairsReduce`, and
- `Call`.

Each skeleton except for `Call` encodes a computational pattern which is efficiently parallelized. In general, the skeletons are differentiated enough to make selection obvious for each use case. However, there is some overlap; for example, MapReduce is an efficient combination of Map and Reduce in sequence. This makes Reduce a special case of MapReduce.

Most of the skeletons are very flexible in how they can be used. All but Reduce and Scan are variadic, and some have different behaviours for one- and two-dimensional computations.

Skeletons in SkePU are instantated by calling factory functions named after the skeletons, returning a ready-to-use skeleton *instance*. The type of this instance is implementation-defined and can

Listing 1.2: Example usage of the Map skeleton.

```cpp
float sum(float a, float b)
{
  return a + b;
}

Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
  auto vsum = Map<2>(sum);
  Vector<float> result(v1.size());
  return vsum(result, v1, v2);
}
```

only be declared as `auto`. This has the consequence of an instance not being possible to declare before definition, passed as function arguments, etc., which is important to consider when architecting applications based on SkePU[4].

SkePU guarantees, however, that a skeleton instance supports a basic set of operations (a ”concept” in C++ parlance).

`instance(`*`args...`*`)` *Invokes* the instance with the arguments. Specific rules for the argument list applies to each skeleton.

`instance.tune()` Performs *tuning* on the instance.

`instance.setBackend(`*`backendspec`*`)` Sets a *backend specification* to be used by the instance and overrides the automatic choice.

`instance.resetBackend()` Clears a backend specification set by `setBackend`.

`instance.setExecPlan(`*`plan`*`)` Sets the execution plan manually. The plan should be heap-allocated, and ownership of it is immediately transferred to the instance and cannot be dereferenced by the caller anymore.

1.10.1 Map

The fundamental property of `Map` is that it represents a set of computations without dependencies. The amount of such computations matches the size of the *element-wise* container arguments in the *application* of a `Map` *instance*. Each such computation is a call to (application of) the user function associated with the `Map` instance, with the element-wise parameters taken from a certain position in the inputs. The return value of the user function is directed to the matching position in the output container.

`Map` can additionally accept any number of *random access* container arguments and *uniform* scalar arguments.

When *invoking* a `Map` skeleton, the output container (required) is passed as the first argument, followed by element-wise containers all of a size and format which matches the output container. After this comes all random-access container arguments in a group, and then all uniform scalars. The user function signature matches this grouping, but without a parameter for the output (this is the return value) and the element-wise parameters being scalar types instead. The return value is the output container, by reference.

An example can be found in Listing 1.2

[4]We are considering different solutions to work around this restriction, please contact the SkePU maintainers if this is important for you.

Listing 1.3: Example usage of the Reduce skeleton.

```
float min_f(float a, float b)
{
  return (a < b) ? a : b;
}

float min_element(Vector<float> &v)
{
  auto min_calc = Reduce(min_f);
  return min_calc(v);
}
```

OpenMP Scheduling Modes

The OpenMP backend in SkePU 3 has changed. It is now possible to control the scheduling mode, as the implementation uses the `runtime` option for OpenMP loop scheduling. The options are static scheduling (default), dynamic scheduling, guided dynamic scheduling or letting the OpenMP runtime decide.

```
skepu::BackendSpec spec{...};

// OpenMP
spec.setBackend(skepu::Backend::Type::OpenMP);
spec.setSchedulingMode(skepu::Backend::Scheduling::Static);
spec.setSchedulingMode(skepu::Backend::Scheduling::Dynamic);
spec.setSchedulingMode(skepu::Backend::Scheduling::Guided);
spec.setSchedulingMode(skepu::Backend::Scheduling::Auto);
spec.setCPUChunkSize(/*int*/);

// CUDA + OpenCL
spec.setBackend(skepu::Backend::Type::CUDA);
spec.setBackend(skepu::Backend::Type::OpenCL);
spec.setDevices(<int>); // number of GPUs
spec.setGPUThreads(/*int*/);
spec.setGPUBlocks(/*int*/);

// Hybrid
spec.setBackend(skepu::Backend::Type::Hybrid);
spec.setCPUPartitionRatio(/*float*/); // CPU fraction, range [0, 1]
```

1.10.2 Reduce

Reduce performs a standard reduction. Two modes are available: 1D reduction on vectors or matrices and 2D reduction on matrices only. An instance of the former type accepts a vector or a matrix, producing a scalar respectively a vector, while the latter only works on matrices. For matrix reductions, the primary direction can be controlled with a parameter on the instance.

The reduction is allowed to be implemented in a tree pattern, so the user function(s) should be associative.

`instance.setReduceMode(`*`mode`*`)` Sets the reduce mode for matrix reductions. The accepted values are `ReduceMode::RowWise` (default) or `ReduceMode::ColWise`.

`instance.setStartValue(`*`value`*`)` Sets the start value for reductions. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example can be found in Listing 1.3

Listing 1.4: Example usage of the MapReduce skeleton.

```
float add(float a, float b)
{
  return a + b;
}

float mult(float a, float b)
{
  return a * b;
}

float dot_product(Vector<float> &v1, Vector<float> &v2)
{
  auto dotprod = MapReduce<2>(mult, add);
  return dotprod(v1, v2);
}
```

Revisions to Reduce Skeleton

The `Reduce` skeleton and the reduce step of `MapReduce` is seeing some changes in SkePU 3.

Reduce modes will be revised to not always trigger data rearrangement such as transposition (sublinear extra memory complexity).

A define is available to enable the old behavoir up to a set container size,
`-DSKEPU_REDUCE2DCOL_TRANSPOSE_SIZE_MAX [n]`.

The revisions to MapReduce skeleton includes the availablilty of an additional reduce mode: not only reduction over the entire container span, but also reduction over the innmost dimension (row-wise for matrices).

1.10.3 MapReduce

`MapReduce` is a combination of `Map` and `Reduce` in sequence and offers the most features of both, for example, only 1D reductions are supported.

An instance is created from two user functions, one for mapping and one for reducing. The reduce function should be associative.

`instance.setStartValue(`*`value`*`)` Sets the start value for reduction. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example can be found in Listing 1.4

1.10.4 Scan

`Scan` performs a generalized prefix sum operation, either inclusive or exclusive.

When *invoking* a `Scan` skeleton, the output container is passed as the first argument, followed by a single input container of equal size to the first argument. The return value is the output container, by reference.

`instance.setScanMode(`*`mode`*`)` Sets the scan mode. The accepted values are `ScanMode::Inclusive` (default) or `ScanMode::Exclusive`.

`instance.setStartValue(`*`value`*`)` Sets the start value for exclusive scan. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example can be found in Listing 1.5

Listing 1.5: Example usage of the Scan skeleton.

```
float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

Vector<float> partial_max(Vector<float> &v)
{
  auto premax = Scan(max_f);
  Vector<float> result(v.size());
  return premax(result, v);
}
```

1.10.5 MapOverlap

`MapOverlap` is a stencil operation. It is similar to Map, but instead of a single element, a region of elements is available in the user function. The region is passed as a pointer, so manual pointer arithmetic is required to access the data. The pointer points to the center element.

A MapOverlap can either be one-dimensional, working on vectors or matrices (separable computations only) or two-dimensional for matrices. The type is set per-instance and deduced from the user function.

The parameter list for a user function to `MapOverlap` is important. It always starts with an `int`, which is the overlap radius in the x-direction. 2D `MapOverlap` also has another `int`, which will bind to the y-direction overlap radius. The presence of this parameter is used to deduce that an instance is for 2D. A `size_t` parameter follows, this is the stride. The next parameter is a pointer to of the contained type, pointing to the center of the overlap region. *Random-access* container and *uniform* scalar arguments follow just as in `Map` and `MapReduce`.

`instance.setOverlap(`*`radius`*`)` Sets the overlap radius for all available dimensions.

`instance.setOverlap(`*`x_radius, y_radius`*`)` For 2D `MapOverlap` only. Sets the overlap for x and y directions.

`instance.getOverlap()` Returns the overlap radius: a single value for 1D `MapOverlap`, a `std::pair` (x, y) for 2D `MapOverlap`.

`instance.setEdgeMode(`*`mode`*`)` Sets the mode to use for out-of-bounds accesses in the overlap region. Allowed values are `Edge::Pad` for a user-supplied constant value, `Edge::Cyclic` for cyclic access, or `Edge::Duplicate` (default) which duplicates the closest element.

`instance.setOverlapMode(`*`mode`*`)` For 1D MapOverlap: Sets the mode to use for operations on matrices. Allowed values are `Overlap::RowWise` (default), `Overlap::ColWise`, `Overlap::RowColWise`, or `Overlap::ColRowWise`. The latter two are for separable 2D operations, implemented as two passes of 1D MapOverlap.

`instance.setPad(`*`pad`*`)` Sets the value to use for out-of-bounds accesses in the overlap region when using `Edge::Pad` overlap mode. Defaults to a default-constructed object, which is `0` for built-in numeric types.

Major Revision to MapOverlap Skeleton

To improve upon the usability of `MapOverlap`, and also to better generalize the syntax to higher-dimensionality smart contaniners (see the addition of tensors in SkePU 3), the interface of MapOverlap user functions has changed.

Before, the signature of a MapOverlap user function was very deliberate with explicit parameters for overlap size, and the smart container data was passed by a raw pointer. This also meant that the adressing into the overlap region was left to the user, and this could be quite tricky to get right.

In SkePU 3, this is now changed by the addition of a new set of smart container proxy classes. Similar to existing `Vec`, `Mat`, and the new `MatRow` (see Section 1.15.3), the `RegionND` (where $N = 1, 2, 3, 4$) classes are proxy classes representing an overlap region from a smart container of dimensionality N.

The region object contains members for accessing the overlap size, `region.on` where $n = i, j, k, l$.

A region object allows element access by using paranthesis notation, with one argument for each dimension. **Note:** the element at position $(0, ..., 0)$ is the center element, and the notation allows access in the range $[-overlap, +overlap]$ (inclusive) for each dimension.

Below are some examples of how to use the new user function syntax. Note that the parameter list after the Section parameter is flexible like before, with any number of random-access and uniform arguments allowed.

```
float over_1d(skepu::Region1D<float> r, int scale)
{
	return (r(-2)*4 + r(-1)*2 + r(0) + r(1)*2 + r(2)*4) / scale;
}
```

```
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;
}
```

```
float over_3d(skepu::Region3D<float> r)
{
	float res = 0;
	for (int i = -r.oi; i <= r.oi; ++i)
		for (int j = -r.oj; j <= r.oj; ++j)
			for (int k = -r.ok; k <= r.ok; ++k)
				res += r(i, j, k);
	return res;
}
```

```
float over_4d(skepu::Region4D<float> r, skepu::Ten4<float> stencil)
{
	float res = 0;
	for (int i = -r.oi; i <= r.oi; ++i)
		for (int j = -r.oj; j <= r.oj; ++j)
			for (int k = -r.ok; k <= r.ok; ++k)
				for (int l = -r.ol; l <= r.ol; ++l)
					res += r(i, j, k, l) * stencil(i + r.oi, j + r.oj, k
					    + r.ok, l + r.ol);
	return res;
}
```

1.10.6 MapPairs

SkePU 3 adds an additional top-level skeleton, **MapPairs**. This skeleton applies a cartesian product-style pattern from *two* `Vector<T>` sets (note that the templated type may differ across these vectors).

Each cartesian combination of vector set indicies generates one userfunction 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.

Each vector set may contain an arbitrary 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.

MapPairs is only defined for Vector-Matrix application, there is no analog for Matrix-Tensor4 at this point.

Below is a simple example of MapPairs in use.

```
int uf(int a, int b) { return a * b }

void test1(size_t Vsize, size_t Hsize, skepu::BackendSpec spec)
{
        auto pairs = skepu::MapPairs(uf);
        pairs.setBackend(spec);

        skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
        skepu::Matrix<int> res(Vsize, Hsize);
        pairs(res, v1, h1);
}
```

Advanced and more flexible use of MapPairs can be carried out similarly to other SkePU skeletons. For instance, it retains flexibility of Map with regards to variadicity (4-way variadic compared to Map being 3-way):

- Elementwise-V args
- Elementwise-H args
- Random-access args
- Uniform args

A more involved skeleton instance might look like this: `auto pairs = skepu::MapPairs<3, 2>(uf2);` Note that the skeleton is templated in number of containers for each set ("vertical" and "horizontal" dimensions). A compatible user function can be seen below:

```
int uf2(
        skepu::Index2D i,
        int ve1, int ve2, int ve3,
        int he1, int he2,
        skepu::Vec<int> test,
        int u1, int u2
)
{
        // some computation involving the above...
        return i.row + i.col + u1 + ve1 + ve2 + ve3
         + he1 + he2 + test.data[0] + u1 + u2;
}
```

1.10.7 MapPairsReduce

SkePU 3 also adds MapPairsReduce, analogous to MapReduce for MapPairs.

Just like MapReduce, the skeleton is initialized with two user functions: one matching the format of a MapPairs user function, and one meeting the restrictions of a Reduce user function.

MapPairsReduce supports arity <0,0> and up. If the arity is 0 in a dimension, it will determine the size of the intermediate Matrix by the values given in a call to setDefaultSize(<Hsize>, <Vsize>) member function beforehand.

The intermediate Matrix is not guaranteed to be stored in memory at any point.

MapPairsReduce supports two reduce modes, both reduce-to-vector: row-wise and col-wise. See the code example below.

```
int uf(int a, int b)
{
        return a * b;
}
```

```
int sum(int lhs, int rhs)
{
	return lhs + rhs;
}

{
	auto mpr = skepu::MapPairsReduce(uf, sum);

	skepu::Vector<int> v1(Vsize), h1(Hsize);
	skepu::Vector<int> resV(Vsize), resH(Hsize);

	mpr.setReduceMode(skepu::ReduceMode::ColWise);
	mpr(resH, v1, h1);

	mpr.setReduceMode(skepu::ReduceMode::RowWise);
	pairs(resV, v1, h1);
}
```

1.10.8 Call

`Call` is special in that it does not provide any pre-defined structure for computation. It is a way to extend SkePU for computations which does not fit into any skeleton, while still utilizing features such as smart containers and tuning. As such, Call provides a minimal interface.

1.11 Multi-Valued Return from Skeletons and User Functions

1.11.1 Multi-valued Return in User Functions

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 one sequence, potentially improving data locality compared to two separate skeleton invocations after each other. Though the values are returned in a tuple-like manner, the output containers are completely separate objects. 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, 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, construct this compound object by `skepu::ret([expression, ...])`.

Below is an example of a user function utilizing this:

```
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);
}
```

The skeleton instance declaration and invocation follows the syntax of oridinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order.

```
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);
```

Multi-valued return statements are available in the `Map` skeleton.

1.12 Multi-Variant User Functions

A new feature in SkePU 3 is the possibility to specify alternative, usually platform-specific, user functions beyond the platform-independent one [8]. This establishes a second level of selection, i.e. nested selection / nested composition. In other words, by adding *multi-variant user functions* to multi-backend skeletons (which are just special predefined multi-variant components) we obtain nested multi-variant components.

There are multiple scenarios where a user function with a singular definition can be too restrictive for the purposes of performance: use cases include algorithms with different trade-offs in time complexity versus memory complexity, instruction set architecture differences such as native double or half precision floating point arithmetics, the existence of SIMD vector instructions, or other hardware-accelerated implementations of common computations. Since these attributes are constrained on the underlying platform, the software-defined code variants must somehow be declared compatible only with the appropriate hardware configurations. For this we employ C++ attributes and the use of XPDL to model the target platform.

Below is a simple SkePU program performing element-wise vector addition:

Listing 1.6: Example: Simple vector addition using Map, with a generic user function.

```
float add(float a, float b) { return a + b; }

int main( int argc, char *argv[] )
{
 const size_t size = N; // multiple of 8
 auto vector_sum = skepu::Map<2>(add);
 skepu::Vector<float> v1(size), v2(size), res(size);
 vector_sum( res, v1, v2 );
}
```

A variant for the user function add, here specialized for systems that support AVX vector instructions, can be defined as follows:

Listing 1.7: Example (cont.): Source code for an alternative user function variant.

```
[[skepu::vectorize(8)]]
void add( float* c, const float *a, const float *b )
{
 __m256 av = _mm256_load_ps(a);
 __m256 bv = _mm256_load_ps(b);
 __m256 cv = _mm256_add_ps(av, bv);
 _mm256_store_ps(c, cv); // return by pointer
}
```

This is placed in a separate file. The presence of this user function variant is made known to SkePU through the use of a *manifest file*, using the syntax below. A list of variants include the name of the variant, which is also the name of the subdirectory that contains the source code, as well as a compile-time predicate (typically an XPDL query [12]) that determines if the variant is eligible for inclusion by the source-to-source compiler.

Listing 1.8: Example (cont.): Manifest file for user function variant.

```
skepu::VariantList {
   skepu::Variant (
   "add_avx" ,
   skepu::Requires (
   xpdl::includes<xpdl::cpu_1::isa_extensions, xpdl_avx>::value ),
   skepu::Backend::Type::CPU
 )
};
```

An XPDL [12] platform description (see Listing 1.9) is then supplied to the SkePU source-to-source compiler and, depending on the attributes in the model, user function variants are either included or removed from the resulting program.

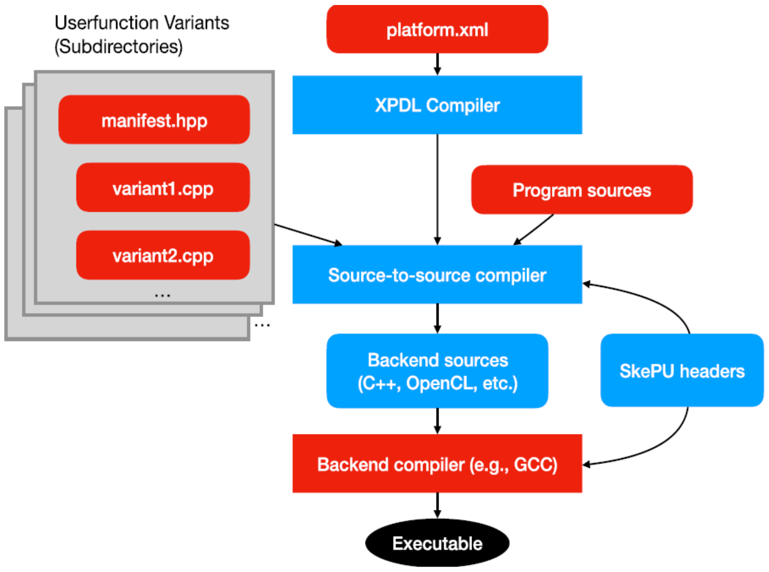

Figure 1.1: Workflow for using multi-variant user functions.

Listing 1.9: Example (cont.): XPDL platform model containing AVX instruction set availability.

```
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/xpdl_cpu"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.xpdl.com/xpdl_cpu.xsd">
 <xpdl:component type="cpu" />
 <xpdl:cpu name="Intel_Xeon_Gold_6130"
           num_of_cores="16"
           num_of_threads="32"
           isa_extensions="avx_avx2">
  <xpdl:group prefix="core_group" quantity="16">
   <xpdl:core frequency="2.1" unit="GHz" />
   <xpdl:cache name="L1" size="32" unit="KiB" set="16" />
   <xpdl:cache name="L2" size="1" unit="MiB" set="16" />
  </xpdl:group>
  <xpdl:cache name="L3" size="22" unit="MiB" set="1" />
  <xpdl:power_model type="power_model_Gold_6130" />
 </xpdl:cpu>
</xpdl:model>
```

See Figure 1.1 for an illustration of the workflow and which source and definition files are required. A SkePU application compiled directly by a C++ compiler ignores any variants, as the feature is purely additive and any variants are defined externally from the main source files.

We refer to our ParCo'19 paper [8], EXA2PRO Deliverable D2.3 or the *EXA2PRO Programming Guide* for further details.

1.13 Manual Backend Selection and Default Settings

1.13.1 Backend Specification API Changes

SkePU 3 changes the backend selection mechanism, in both API and implementation. Especially the OpenMP parameters are expanded with new scheduling mechanisms.

Global Backend Specification

In SkePU 3 it is now possible to set a global backend specification. This specification will be used by default for all skeleton instances. Overriding can be done on an instance basis using a member function, see example.

```
skepu::BackendSpec spec{/*string or SkePU::Backend::Type enum*/};

skepu::setGlobalBackendSpec(spec);

skepu::restoreDefaultGlobalBackendSpec();

skel.setBackend(other_spec); // now overrides global specification
```

1.14 Tuning of Skeleton Instances

A skeleton instance can be tuned for backend selection by going though a process of training on different input sizes of the *element-wise* arguments. This process is automated, but since there is significant overhead (during the tuning process, not afterwards) it has to be started manually. An instance is tuned by calling `instance.tune()`. Note that this is an experimental feature with limitations. Only the size of element-wise arguments can be used as the tuner's problem size, which is not applicable to all types of computations possible with SkePU.

Tuning creates an internal execution plan which is used as a look-up table during *skeleton invocation*. It is also possible to construct such a plan manually, and assign it to

1.15 Smart Containers

The smart containers available in SkePU are `Vector` and `Matrix`. Using these is mostly transparent, as they will optimize memory management and data movement dynamically between CPU and GPUs.

There is also manual interface for data movement: `container.updateHost()` will force download of up-to-date data from the GPUs, and `container.invalidateDeviceData()` forces a re-upload for the next skeleton invocation on a GPU.

Element access on the CPU can be done either with `operator[`*`index`*`]`, which includes overhead for checking remove copies, or `operator(`*`index`*`)` which provides direct, no-overhead access.

When smart containers are used as *element-wise parameters* to user functions, it is important to note that separate types are used, `Vec` and `Mat`. These proxy types do not provide the full smart container functionality and are used with a C-style interface. Elements are retrieved using `container.data[`*`index`*`]` member, and `size`, `rows`, and `cols` are members and not member functions. By default, the arguments are read/writeable and will encur copy operations both up and down from GPUs; by adding `const` qualifier, the copy-down is emliminated. Similarily, a `[[skepu::out]]` attribute will turn them into output parameters.

1.15.1 Smart Container Set

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". Smart container dimensionality in SkePU 3 is therefore static, though their sizes in each dimension is dynamic.

The interface for these containers are virtually indentical to those of the other containers, differing in the obvious ways of naming and element access detailed below. They are defined as follows:

```
template<typename T> Tensor3 { ... };
template<typename T> Tensor4 { ... };
```

Instances of these tensor types are created with one constructor argument for each dimension. Optionally an additional argument of type `T` specifices the default value of all elements in the container.

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

Compare with existing containers:

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
```

The Index object set in SkePU, useable in e.g. user function signatures, is extended with these structs:

```
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

This complements the existing structs. Note that the naming convention is different for matrix indices for compatibility reasons.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
```

Tensor Usage

`Tensor3<T>` and `Tensor4<T>` are useable in a way analogous to `Matrix<T>` in most cases.

Map	Over the full domain without regard to dimensionality Optional argument `Index3D` for `Tensor3<T>` and `Index4D` for `Tensor4<T>` ufs
Reduce	Over full domain or over inmost dimension.
MapReduce	See Map. For the reduce step, over full domain or the innmost dimension.
Scan	Over full domain.
MapOverlap	Overlap radius limited to 1 in each dim (default) Larger overlap dependent on backend support.
Call	See Map

1.15.2 Smart Container Element Access

SkePU 3 **deprecates** the angle bracket `[ ]`-notation for smart container element read/write access outside user functions. This is part of a simplification of the coherency systems for manual element access from the host (CPU) side.

The user should flush the whole container instead before doing single-element accesses of user function data, see Section 1.15.2.

Instead of angle brackets, the parantheses `( )`-notation is extended to higher dimensionality. This syntax accepts one index argument for each dimension of the underlying container. The indicies count must equal container dimensionality, otherwise there is a compile-time error.

Formally, the syntax is `container(i,[j, [k, [l]]]) [= value];`

This change means that there's no interface for 1D indexing of higher-dimensionality containers.

There is no longer a coherency-satisying single-element access mechanism in SkePU smart containers except inside user function proxy objects (`Vec<T>`, `Mat<T>`, etc). However, for correctness debugging purposes, there is a macro to enable explicit flush of a container upon access,
`-DSKEPU_ALWAYS_UPDATE_HOST_ON_CPU_ELEMENT_ACCESS [0,1]`,
but note that this has serious performance implications.

Memory Coherency

flush interface revised from feedback from partners flush with options

`enum class FlushMode  Default, Dealloc ;`

`FlushMode::Default` is implicit if no other value given.

Container member function flush as well as variadic free template function flush

Member function: dynamic flush mode, can be selected at runtime

Free function: flush mode is static constant, known to the compiler (and precompiler)

```
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);
```

There is no `#pragma` for flush declarations in SkePU, but the flush (member) functions are compiler-known symbols to the precompiler, as are smart container classes, so the presence or absence of flush operations in SkePU source code is subject to static analysis and optimization.

1.15.3 Matrix-row User Function Proxy Containers

SkePU has since version 2 allowed for flexible parameter lists for user functions, including so-called *random-access* containers in addition to the for skeleton programming standard element-wise mapped containers. While this allows for powerful expressivity, very little about the access patterns of these random-access containers are known to SkePU, and performance may thus not always be ideal.

One common pattern when using `Matrix` as a random-access container argument 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 container 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 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.

Note: it is required that the matrix container has at least as many rows as the result vector has elements.

Example. Matrix-vector multiplication using `MatRow<T>` may be implemented as folows:

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

Compared to the closest corresponding SkePU 2 implementation below (still valid in SkePU 3), the code is more succinct and there is more information about the access pattern avaiable to SkePU.

```
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;
}
```

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

Matrix-row user function proxy containers are available in user functions for `Map`, `MapReduce`, and `MapOverlap` skeleton instances that satisfy the above requirements.

1.16 Using Custom Types

It is possible to use custom types in SkePU containers or inside user functions. These types should be C-style structs for compatibility with OpenCL. **Note**: It is not guaranteed that a struct has the same data layout in OpenCL as on the CPU. SkePU does not perform any translation between layouts, so it is the responsibility of the user to ensure that the layout matches.

1.17 Calling Library Functions; Whitelisting

Sometimes, it can be beneficial to call built-in/library functions from inside user functions. By default, SkePU assumes that all called functions are to be processed by the precompiler, which will prevent using library functions, because SkePU does not know in general whether these are available on every accelerator type supported and functionally equivalent to their CPU counterparts.

To avoid this issue, use the `-fnames` argument of the SkePU precompiler. This flag tells SkePU to ignore any function with this symbol name (all possible overloads), and it is up to the user to ensure that the compiler and linker for each backend can find suitable functions to call.

This feature is useful for whitelisting mathematical functions or `printf` debugging, but is best used very carefully, especially if accelerator backends are enabled. See the example usage below, as part of the skepu-tool invocation. Multiple function names are separated by whitespace.

```
skepu-tool  -fnames "sin␣cos"
```

1.18 SkePU 3 Changelog

This section goes through all the syntactical and behavioral changes from SkePU 2 to SkePU 3.

In particular, the skeleton set has changed in SkePU 3, with the addition of the all-new `MapPairs` and `MapPairsReduce` skeletons, important extensions to the capabilities of the standard `Map` skeleton, and an interface change to improve usability of the `MapOverlap` skeleton.

The smart container set has also seen an extension in SkePU 3, the framework now has higher-dimensionality *tensors* in `Tensor3` and `Tensor4`. The coherency model of smart containers has also seen an update.

1.18.1 Namespace Change

The namespace for SkePU is changed in SkePU 3. Historically, the `skepu::` namespace was used by the initial SkePU release ("SkePU 1"), and since SkePU 2 was a major source-breaking change from SkePU 1, the decision was made to switch over the namespace as a way to communicate the source incompatibilities.

Today, there is to our knowledge little or no application code in active use which depends on SkePU 1. The decision was therefore made to switch back to the version-agnostic `skepu::` namespace for new releases of SkePU, starting with SkePU 3. This has the additional benefit of communicating that SkePU 3 also is a source-breaking transition from SkePU 2, although this time the scope of the changes is much smaller and transitioning between SkePU 2 to 3 is expected to be much simpler.

The intention is to keep this namespace for future versions of SkePU, and future source-breaking changes will be communicated in other ways.

The above namespace applies to skeletons and smart containers alike, as well as suporting constructs such as enums, backend specifications, container proxy objects, and index structs. In short, every C++ symbol in the library is affected. The exeption is C++11 attributes, whose names are forced to be namespaced but these namespaces are distinct from the standard C++ namespace definitions. This was introducted to SkePU in version 2 and have always been `skepu::`, (e.g. `[[skepu::out]]` and will remain as such.

In addition to the namespace change the default include header (the main entry point to the SkePU header library) has been changed. `#include <skepu2.hpp>` is replaced by `#include <skepu>` which aside from dropping the version now also mirrors the standard library with the absence of a file extension. (Note that the include directive may be different depending on the directory setup.)

1.18.2 Other Changes Summary

The following major changes, described in more detail above, have been applied in the transition from SkePU-2 to SkePU-3 (as of May 2020):

- New skeletons MapPairs, MapPairsReduce;
- Revised interface for MapOverlap and Reduce;
- New container types Tensor3, Tensor4 and new container proxy types MatRow, Ten3, Ten4;
- Deprecation of any STL-inherited dynamic features of the SkePU containers as a clarification that they are to be used as static objects;
- Multi-valued return from skeletons and user functions;
- Multi-variant user functions;
- Dynamic scheduling option for all skeletons except Scan and Call;
- New memory consistency model for smart containers: weak consistency;
- New MPI backend (available for some skeletons at this time), along with a new construct (interface still pending) to frame external I/O operations.

Several of these design changes are the result of feedback from application partners in the running H2020 FETHPC project EXA2PRO.

Moreover, the public distribution of SkePU has moved to github.com/skepu/skepu and the SkePU web page has moved from Linköping University to skepu.github.io.

The SkePU license has been changed from GPLv3 for SkePU-2 to a less restrictive *modified 4-clause BSD license* for SkePU-3.

1.19 More Information about SkePU

For further information about SkePU we refer to our publications.

SkePU 1 was introduced in 2010 and is presented in Enmyren and Kessler [5].

The integration of SkePU and StarPU to provide data-driven dynamic scheduling of asynchronous skeleton calls was presented by Dastgeer et al. [3].

The second generation of a back-end selection tuning framework for SkePU was developed by Dastgeer et al. [4]. Dastgeer also further developed the smart data-containers (Vector, Matrix) in [2]. Selection tuning and smart containers are still contained in today's SkePU implementation.

SkePU 1 supports sequential and multithreaded CPU execution as well as single- and multi-GPU execution in OpenMP and CUDA backends. These backends are, in modified form, still part of today's SkePU implementation. Experimental back-ends for SkePU 1 had also been developed for plain MPI [15] and Movidius Myriad 2 [19], but were not mature enough to be included in the public distribution and were finally abandoned at the transition to SkePU-2.

Case studies on SkePU 1 include the porting of the EDGE flow simulation code [18], which revealed a number of weaknesses in the SkePU 1 API and finally led to the design of SkePU 2.

SkePU 2 was introduced in 2016. It involved the complete redesign of the SkePU programming interface based on C++11, and is described in Ernstsson et al. [9].

The generalization of smart containers for lazy execution of skeletons to provide global run-time optimizations such as tiling and kernel fusion across data-flow graphs of containers and skeletons was presented by Ernstsson and Kessler [7].

A new hybrid CPU-GPU back-end for SkePU 2 was proposed by Öhberg et al. [16] and is included in today's implementation.

The concept of multi-variant user functions and their implementation (included in SkePU-3) is presented in Ernstsson and Kessler [8].

Panagiotou et al. [17] present a case study of using SkePU in a brain modeling application, including first scaling results for the new SkePU 3 cluster backend based on StarPU-MPI [1], which is included in the distribution.

A SkePU tutorial, including most of the new SkePU 3 features, can be found on the SkePU web page [14].

A paper presenting SkePU 3 in its entirety is currently in preparation.

Chapter 2

ComPU User Guide

Revision history

- **1.0:** Autumn 2019
  First EXA2PRO release.
- **1.1:** 2020-29-01
  January release of the EXA2PRO framework.
- **1.2** 2020-12
  December release (D3.6)
- **1.3** 2021-09-01
  Fixed some typos, added sections on MPI support and checkpointing. Added some references.

2.1 Introduction

ComPU is the EXA2PRO composition tool, aiming at enabling programmability and performance portability across heterogeneous many-core systems, including systems accelerated with GPUs and Maxeler DFE FPGAs. ComPU is an application build tool that processes annotated multi-variant components (see below) and binds them together with the application's non-componentized (outer) code to an executable application that runs atop the EXA2PRO runtime system (StarPU). For a more complete description of the composition framework with examples and illustrations, we refer to EXA2PRO deliverable report D3.6 [13].

Multi-variant components are separately defined modularized functions with multiple implementation variants for different supported platform-specific programming models (sequential C, OpenMP, CUDA, OpenCL, Maxeler DFE) that are annotated using metadata given in the EXA2PRO XML descriptors for application, interface, and implementation descriptors. For a description of the component annotation syntax with examples, we refer to the *EXA2PRO Programming Guide*, which can be found on the EXA2PRO web page[1], and to EXA2PRO Deliverable D2.3 [10].

Given a set of such XML descriptors, and the implementation source code file of each variant, ComPU can generate code that at run-time (using the StarPU runtime system) can select one of the available variants that performs better in the given context.

Code examples that illustrate how ComPU works are provided in Section 2.4 as well as in the examples folder of the ComPU repo. We also include sections about the use of the MPI support and high-level checkpointing support with examples.

2.2 Dependencies and Requirements

To build ComPU, it is required to have the following software / frameworks installed in your system:

- StarPU
- Xerces
- XPDL

2.3 Installation

In what follows, there are the steps required to install ComPU and the dependencies.

2.3.1 Installing StarPU

ComPU is tested with StarPU version 1.3.1. The installation instructions for StarPU can be found on the StarPU webpage: `http://starpu.gforge.inria.fr`. Please note that if you use the `Tensor` smart containers in your multi-variant components of ComPU, the StarPU version that supports Tensors is required. Similarly, if you want to use DFE implementations with a multi-variant component, then the corresponding StarPU version that supports DFE is needed. Also note that if ComPU and SkePU StarPU-MPI are used in conjunction, then the the StarPU versions required are the same as if using SkePU stand-alone. Please refer to the SkePU system requirements section in the SkePU user guide (chapter) for more information.

After the StarPU installation, for ComPU it is important to set the following StarPU-related environment variables:

[1] `https://exa2pro.eu`

```
export STARPU_HOME=YOUR_STARPU_INSTALL_PATH
export PKG_CONFIG_PATH=$STARPU_HOME/lib/pkgconfig:$PKG_CONFIG_PATH
export PATH=$STARPU_HOME/bin:$PATH
export LD_LIBRARY_PATH=$STARPU_HOME/lib:$LD_LIBRARY_PATH
```

2.3.2 Installing Xerces

ComPU is tested with Xerces version 3.1.2. The installation instructions for Xerces can be found on the Xerces webpage (`https://xerces.apache.org`).

After the installation of Xerces, set the environment variables as follows:

```
export XERCES_HOME=YOUR_XERCES_INSTALL_PATH
export PATH=$XERCES_HOME/bin:$PATH
export LD_LIBRARY_PATH=$XERCES_HOME/lib:$LD_LIBRARY_PATH
```

2.3.3 Installing XPDL

For installing XPDL we assume that Xerces is already installed in your system and its environment variables have been set. If Xerces is not installed yet, please refer to Section 2.3.2 of this document.

- Set the following environment variables:

  ```
  export XPDL_HOME=YOUR_XPDL_PATH
  export PATH=$XPDL_HOME/bin:$PATH
  ```

- Go to `src/`
- `make`
- `make run`

`make run` will generate the `xpdl.hpp` file. An example model of a platform described using XPDL is provided in the `XPDL_root/example` folder.

2.3.4 Set ComPU environment variables

Currently, ComPU requires to set the environment variables for the desired ComPU installation path. The exact expected variable names for ComPU are as follows:

```
export CT_HOME=YOUR_COMPU_PATH
export PATH=${CT_HOME}/src:$PATH
```

2.3.5 Installing ComPU

Now that all dependencies are installed and the environment variables are set, inside the `src` folder, run `make`, and if successful, the `compose` binary file will be generated inside the folder. One can export the `src` folder of ComPU to PATH for global access to `compose`.

2.4 Usage

Go to one of the examples provided in the examples folder, and run the following command:

```
compose main.xml
```

ComPU will generate the wrappers, the application build makefile, and other relevant files, see also the example in Section 2.5. By running the make command, the code will be compiled and ready for execution. Please note that we assume that the required tools for compilation of CUDA, OpenCL, or other components are already installed.

List of ComPU command line options

The general format of the *compose* command is: `compose <XML file> [options]`

The list of ComPU command line options is listed below, and can also be found by running `compose --help` on the composition tool.

Options:

- `-use`*`Perf`*`Models`: to enable usage of a StarPU performance model. *Perf* can be *History*, *Regression*, *NLRegression* or *Energy*, see Section 2.4.1.
- `-disableplatforms=CPU,CUDA,OPENCL,MAXDFE`: to disable one or more platforms for all components.
- `-readPerfModels`: to read the performance models' attributes from the XML performance model files generated by StarPU.
- `-useComir`: to allow changes in the COMIR representation by ComPU plug-ins before exporting it to the generated `comir_tmp.cpp` file. If we do not use the option **-useComir**, the `comir_tmp` file will be generated based on the metadata read from the XML descriptors. For more information about COMIR see Sect. 2.4.4.

Note that for the option `-useComir`, it is left to the programmer to make the appropriate changes inside the file
`src/plugins/comir_plugins/comirPlugins.cpp`.
for adding own ComPU plugin code using COMIR, at the intended point in the processing flow.

Please look at the included example programs and XML descriptors to get an idea of how everything works in practice.

2.4.1 StarPU Performance Models

As mentioned in Section 2.4, in order to enable a performance model, you should use the corresponding command line option `-use`*`Perf`*`Models`. The XML descriptors of the implementations should contain the attributes of the respective performance model that are going to be measured.

For the available models (which are inherited from StarPU, see D3.8), the model coefficients are named as follows (where T denotes time in milliseconds, E denotes energy in Joules, *Consumption* denotes the estimated task energy consumption in Joules, γ denotes the execution time penalty of a Joule, N denotes operand data size):

- **History model**: the following summary parameters are stored:
  min_size, *max_size* - the minimum and maximum size for which entries exist in the history model;
  min_mean, *max_mean*, *min_deviation*, *max_deviation* - the average execution time and

standard deviation for the minimum and maximum problem size, respectively, on the processing unit of this implementation, averaged over the samples of the respective size (depending on how many times the generated program runs the task with the respective minimum or maximum size). See also the StarPU handbook, Chapter 13.3.

- **Linear regression model**: $T = \alpha \cdot N^{\beta}$
- **Nonlinear regression model**: $T = a \cdot N^{b} + c$
- Support for the **multi-linear regression model** of D3.4 is currently being added in ComPU.
- **Energy model**: $E = \gamma \cdot Consumption$. See the StarPU handbook, Chapter 8.3, for further information on the energy model.

After running `compose` with the appropriate arguments, and after all the files have been generated, execute the code according to StarPU's instructions, in order to take the XML files with the information about the used performance model.

In order to read StarPU's performance models, we assume that the steps mentioned above have already been completed. Then, `compose` should be run again with the option `-readPerfModels`. This option will lead to the reading of the performance model's attributes and their inclusion in the `comir_tmp` file. In addition, it will perform *static implementation pruning*, i.e., disable those implementations that are completely dominated by the others across the entire relevant scope of problem sizes[2], for each component separately. Moreover, it will select the most efficient implementation, based on the problem size, and it will submit the task to an execution resource for the respective platform.

2.4.2 Using SkePU skeletons with multi-variant components

ComPU provides means to use SkePU skeletons together with multi-variant components. In such a case, the developer needs to explicitly annotate their applications through the implementation descriptor by setting the `type` to `skepu` for each of the source-code files that uses SkePU.

ComPU will generate the compiler statements specific to the SkePU pre-compiler (named `skepu-tool`), which should be included to your `PATH` environment variable. Depending on the enabled/disabled platforms, ComPU will instruct the SkePU precompiler to generate the corresponding skeleton backends. Unless otherwise specified, all skeleton backends will be generated.

2.4.3 MAXDFE Support

ComPU supports Maxeler DFE with the help of a StarPU version[3] extended for use with Maxeler DFE FPGAs. The programmer has to write the `Kernel` and the `Manager` for the tasks. Then, the Maxeler toolchain is used for generating the bitstream, according to the instructions on the following webpage:
`https://trac.version.fz-juelich.de/reconfigurable/wiki/Public`.

After the generation of the bitstream, the programmer has to add the following lines inside the MAXDFE function file:

```
#include "path_to_bitstream"
#include "MaxSLiCInterface.h"
```

[2]The programmer can modify the size scope of interest by changing the $Nmin$ and $Nmax$ values in the file *src/infoHolder/Component.cpp*.

[3]This version can, at the time of writing this document, be found in the public repository of StarPU at `https://gitlab.inria.fr/starpu/starpu/-/tree/fpga`.

The bitstream, together with the referencing MAXDFE (C) function file and the MAXDFE implementation descriptor are to be located in the MAXDFE subfolder of the component. As shown above, the bitstream is referenced from the C function, not directly in the descriptor.

The following steps are as usual: run `compose`, `make`, and execute the code.

2.4.4 COMIR

COMIR represents metadata about the application. Those metadata are originally stored in the XML descriptors. When `compose` is run, COMIR is finally exported by generating the file `comir_tmp.cpp` file.[4]

The standard information that COMIR (and hence, the generated file) contains, includes the application's components, implementation variants, the component's parameters and pre-selection configurations. Extra information can be added to `comir_tmp.cpp`, as well as the COMIR contents can be modified by using the COMIR API setter functions. The file `comir_tmp.cpp` will, together with the COMIR implementation in `comir.hpp` and `comir.cpp`, be compiled into a dynamically linked library file that will be used by the Mercurium-based precompiler in order to import COMIR.

The extra information that can be added are constraints and the performance models attributes (see Section 2.4.1). The programmer can modify the existing information by writing ComPU plug-in code to be called from the `src/plugins/comir_plugins/comirPlugins.cpp` source file and running `compose` with the `-useComir` argument.

2.4.5 Implementation / platform selection

In ComPU, there are three ways to select a platform for a component call. The first way is by forcing the application to run on a specific platform by disabling the other platforms. This can be done either generally for all calls by using ComPU's `compose` option `-disableplatforms`, or by disabling them for a specific run with the help of StarPU's corresponding commands. The second way is by letting StarPU decide which platform is the best for the task(s) generated from the call based on its internal performance model. The third way is by using the `x2p_call_select(`***`platform`***`)` function before a component call to which it applies.

For the latter, the `x2p_call_select(`***`platform`***`)` function call is being placed immediately before a component call in outer EXA2PRO code whose platform selection should be forced to ***`platform`***. The *platform* can be any of the available platforms for the component[5]. The advantage of call-specific selection control with the `x2p_call_select(`***`platform`***`)` function is that the programmer can select different platforms for different calls of the components, for example for debugging or for performance test purposes.

2.4.6 Streamable access pattern

When the access pattern of a component operand has been declared as `streamable`, then the elements are being accessed in a consecutive way in forward order. In order to achieve that, ComPU makes use of StarPU's functionality on partitioning the data. The default number of partitions (tiles) is 1, i.e., for each component call a single StarPU task will be created. In order to change the number of tiles (tasks), the programmer should call the function `x2p_set_tiles(`***`tiles`***`)` with the desired number of tiles *tiles*> 1 in the outer code just before the call.

[4]The `comir_tmp.cpp` file will be generated always when we run `compose`, either the default (`compose main.cpp`) or with a command option. The difference is that with some of the command options (`-disableplatforms`, `-readPerfModels`, `-useComir`), the content of the generated Comir file can differ from the read descriptor information, e.g., if we disable some platforms or if we read in the performance models.

[5]A call to an OpenCL implementation (`x2p_call_select(x2p_OPENCL)`) will execute only if the CUDA platform is disabled.

The *streamable* access pattern has two optional arguments. The first argument indicates the number of dimensions[6] that are going to be accessed as streamable, and the second argument indicates a tunable parameter `stream_tile_size_`*x*, which will specify the number of elements that are going to be processed together. For example `streamable(1,1)`, indicates that only the first dimension of the (possibly, higher-dimensional) operand is streamable and that the number of elements that are going to be processed together will be the number specified by the `stream_tile_size_1`.

2.5 Example

By a simple code example, we will demonstrate the descriptor syntax and the functionality of the composition framework.

We will use a simple vector scale example, which has one component (named `vector_scale`) with two implementation variants (CPU and CUDA). The folder structure with all the files is shown below.

Listing 2.1: The folder structure of the vector scale multi-variant application using ComPU.

```
    - vector_scale_application
        - vector_scale
            - CPU
              vector_scale_cpu.cpp
              vector_scale_cpu.xml
            - CUDA
              vector_scale_cuda.cu
              vector_scale_cuda.xml
            vector_scale.xml
        main.c
        main.xml
```

The application entry point (that is, `main.c`) and its corresponding application descriptor (that is, `main.xml`) are stored in the application's root folder, whereas all of the components used in this application are stored in their separate folders (for example, `vector_scale`). The content of `main.c` is shown in Listing 2.2, whereas Listing 2.3 shows the content of `main.xml`.

Listing 2.2: `main.cpp` – A simple code example of the application entry point for a simple vector scale application that uses one multi-variant component.

```
#include "exa2pro.h"
#include <iostream>
#include <cstdlib>

using namespace std;

#define NX 30

void display (float *vec)
{
    for(unsigned int i = 0; i < NX; ++i)
    {
        std::cout<<vec[i]<<" ";
    }
    std::cout<<"\n";
}

int main(int argc, char **argv)
{
        EXA2PRO_INITIALIZE(); // initialize exa2pro framework...

        float vector[NX];
```

[6]The ordering of dimensions is such that tiles are always contiguous memory blocks, assuming C/C++ memory layout of multidimensional array structures.

```
        for(int i=0;i<NX;i++)
        {
                vector[i] = (float)( rand() % 50 + 10 );
        }

        std::cout<<"BEFORE: ";
        display(vector);

        float factor = 3.5;

        for(int i=0;i<5;i++)
                vector_scale(vector, NX, factor);

        std::cout<<"AFTER: ";
        display(vector);

        EXA2PRO_SHUTDOWN(); // de-initialize exa2pro framework...

        return 0;
}
```

Listing 2.3: `main.xml` – A simple example of the application descriptor for the vector scale application that uses one multi-variant component.

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
        xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1 ComponentMetaData0.1.
            xsd">

        <x2p:implementation name="main" providedInterface="main" targetPlatform="CPU" >
                <x2p:requiredInterfaces>
                        <x2p:requiredInterface name="vector_scale" />
                </x2p:requiredInterfaces>

                <x2p:sourceFiles>
                        <x2p:sourceFile version="1.0" language="C++">
                                <x2p:compilation type="link" compiler="g++-7" version="4.5"
                                    flags="$(LDFLAGS) $(shell pkg-config --libs cuda-9.2
                                    cudart-9.2) -lOpenCL" output="main" />
                        </x2p:sourceFile>

                        <x2p:sourceFile name="main.c" version="1.0" language="C++">
                                <x2p:compilation compiler="g++-7" version="4.5" flags="-std=c
                                    ++11"/>
                        </x2p:sourceFile>
                </x2p:sourceFiles>
        </x2p:implementation>
</x2p:component>
```

For each component, a folder with the same name as the one provided through the `x2p:requiredInterface` should exist, and for each of the implementation variants of that component, a subfolder with the name of the target platform that implementation variant is written for should exist. In our example, the `vector_scale` folder has two subfolders, one for CPU and another for GPU (the list of available platforms is provided below). Below, in Listing 2.4 we show the content of the interface descriptor for our vector scale component, and for each of the implementation variants we show the implementation descriptors (Listing 2.6 and 2.8) and their source code (Listing 2.5 and 2.7).

Listing 2.4: `vector_scale/vector_scale.xml` – A simple example of the interface descriptor for the vector scale component.

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
        xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1 ComponentMetaData0.1.
            xsd">
        <x2p:interface name="vector_scale">
  <x2p:parameters>
    <x2p:parameter name="arr" type="float *" numElements="size" accessMode="readwrite" />
    <x2p:parameter name="size" type="int" accessMode="read" />
    <x2p:parameter name="factor" type="float" accessMode="read" />
  </x2p:parameters>
</x2p:interface>

</x2p:component>
```

Listing 2.5: `vector_scale/CPU/vector_scale_cpu.c` – The source code of the CPU implementation variant for our vector scale component.

```
#include <stdio.h>

void scale_cpu_func(float *arr, int size, float factor)
{
        /* scale the vector */
        printf("\n************* Hello in vector-scale CPU call ********************\n\n");
        unsigned i;
        for(i = 0; i < size; i++)
                arr[i] *= factor;
}
```

Listing 2.6: `vector_scale/CPU/vector_scale_cpu.xml` – The component implementation descriptor of the CPU implementation variant for our vector scale component.

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
        xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1 ComponentMetaData0.1.
            xsd">

        <x2p:implementation name="scale_cpu_func" targetPlatform="CPU" providedInterface="
            vector_scale">
                <x2p:sourceFiles>
                        <x2p:sourceFile name="vector_scale_cpu.c" version="1.0" language="C">
                                <x2p:compilation compiler="g++-7" version="5.4" />
                        </x2p:sourceFile>
                </x2p:sourceFiles>
        </x2p:implementation>
</x2p:component>
```

Listing 2.7: `vector_scale/CUDA/vector_scale_cuda.cu` – The source code of the CUDA implementation variant for our vector scale component.

```
#include <stdio.h>
#include <starpu.h>

static __global__ void vector_mult_cuda(float *val, unsigned n, float factor)
{
        unsigned i;
        for(i = 0 ; i < n ; i++)
               val[i] *= factor;
}

void scale_cuda_func(float *arr, int size, float factor)
{
    printf("\n************* Hello in vector-scale CUDA call ********************\n\n");
        vector_mult_cuda<<<1,1>>>(arr, size, factor);
        cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Listing 2.8: `vector_scale/CUDA/vector_scale_cuda.xml` – The component implementation descriptor of the CUDA implementation varioant for our vector scale component.

```
<?xml version="1.0"?>
<x2p:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:x2p="http://www.exa2pro.eu/ComponentMetaData0.1"
        xsi:schemaLocation="http://www.exa2pro.eu/ComponentMetaData0.1 ComponentMetaData0.1.
            xsd">

        <x2p:implementation name="scale_cuda_func" targetPlatform="CUDA" providedInterface="
            vector_scale">

                <x2p:sourceFiles>
                        <x2p:sourceFile name="vector_scale_cuda.cu" version="1.0" language="
                            CUDA">
                                <x2p:compilation compiler="nvcc" flags="--compiler-options -
                                    fpermissive" />
                        </x2p:sourceFile>
                </x2p:sourceFiles>
        </x2p:implementation>
</x2p:component>
```

Given the above source code and descriptors, the composition framework can be invoked using the command below:

```
compose main.xml
```

which will generate the following files:

- the `exa2pro.h` header file that includes the definitions of the `EXA2PRO_INITIALIZE()` and `EXA2PRO_SHUTDOWN()` functionality (see Listing 2.9),
- the `vector_scale_wrapper.h` file that includes the stubs generated for each variant as well as the functionality that submits tasks to the StarPU runtime system (see Listing 2.10), and
- the `makefile` that contains the compilation and linking commands (see Listing 2.11).

Listing 2.9: `exa2pro.h` – generated by the composition framework.

```
#ifndef EXA2PRO_H
#define EXA2PRO_H
#include <starpu.h>
#define USE_STARPU

#define x2p_CPU STARPU_CPU
#define x2p_CUDA STARPU_CUDA
#define x2p_OPENCL STARPU_OPENCL
#define x2p_MAXDFE STARPU_FPGA

#define stream_tile_size_0 2
#define stream_tile_size_1 10

#define x2p_task_wait starpu_task_wait()

unsigned long int x2p_nextcall_bitvector;

#define x2p_call_select(X) (x2p_nextcall_bitvector=X)

unsigned int Ntiles = 1;

#define x2p_set_tiles(N) if(N>1) {Ntiles=N; printf("Ntiles = %d\n",Ntiles);}

// Include wrappers here for each interfaces which can have multiple implementations.
// Here, we only have one interface...
#include "vector_scale_wrapper.h"

#ifdef USE_STARPU
#define EXA2PRO_INITIALIZE() do{ \
struct starpu_conf conf; \
starpu_conf_init(&conf); \
\
\
int _ret_ = starpu_init(&conf); \
STARPU_CHECK_RETURN_VALUE(_ret_, "starpu_init") }\
while(0)
#else
#define EXA2PRO_INITIALIZE()
#endif

#ifdef USE_STARPU
#define EXA2PRO_SHUTDOWN() \
\
starpu_shutdown()
#else
#define EXA2PRO_SHUTDOWN()
#endif

#endif
```

Listing 2.10: `vector_scale_wrapper.h` – generated by the composition framework.

```
#ifndef VECTOR_SCALE_WRAPPERR
#define VECTOR_SCALE_WRAPPERR

typedef struct
{
    float factor;
} ROA_vector_scale;

typedef struct
{
        struct starpu_codelet cl_vector_scale;

    int cl_vector_scale_init;
} struct_vector_scale;

extern void scale_cpu_func( float * arr, int size, float factor);

void scale_cpu_func_wrapper (void *buffers[], void *_args)
{
    scale_cpu_func (
        (float *)STARPU_VECTOR_GET_PTR( (struct starpu_vector_interface *)buffers[0]),
            STARPU_VECTOR_GET_NX( (struct starpu_vector_interface *)buffers[0]),((
            ROA_vector_scale *)_args)->factor
    );
}

extern void scale_cuda_func( float * arr, int size, float factor);

void scale_cuda_func_wrapper (void *buffers[], void *_args)
{
    scale_cuda_func (
        (float *)STARPU_VECTOR_GET_PTR( (struct starpu_vector_interface *)buffers[0]),
            STARPU_VECTOR_GET_NX( (struct starpu_vector_interface *)buffers[0]),((
            ROA_vector_scale *)_args)->factor
    );
}

static struct_vector_scale * objSt_vector_scale = NULL;

void vector_scale (  float * arr, int size, float factor )
{
    static ROA_vector_scale arg_vector_scale;

    arg_vector_scale.factor=factor;

    // static struct_vector_scale * objSt_vector_scale = NULL;

    if( objSt_vector_scale == NULL)
    {
        objSt_vector_scale = ( struct_vector_scale *) malloc (sizeof( struct_vector_scale ));
        memset( &(objSt_vector_scale->cl_vector_scale), 0, sizeof(objSt_vector_scale->
            cl_vector_scale));

        objSt_vector_scale->cl_vector_scale_init = 0;
    }

    if(! objSt_vector_scale->cl_vector_scale_init ) // codelete initialization only once, at
        first invocation
    {
        // objSt_vector_scale->cl_vector_scale.where =0|STARPU_CPU|STARPU_CUDA;

        objSt_vector_scale->cl_vector_scale.cpu_funcs[0]=scale_cpu_func_wrapper;
        objSt_vector_scale->cl_vector_scale.cpu_funcs[1]=NULL;
        objSt_vector_scale->cl_vector_scale.cuda_funcs[0]=scale_cuda_func_wrapper;
        objSt_vector_scale->cl_vector_scale.cuda_funcs[1]=NULL;
        objSt_vector_scale->cl_vector_scale.nbuffers = 1;
        objSt_vector_scale->cl_vector_scale.modes[0] = STARPU_RW;
        objSt_vector_scale->cl_vector_scale_init = 1;
    }

    starpu_data_handle_t arr_handle;

    starpu_vector_data_register(  &arr_handle, 0, (uintptr_t)arr, size, sizeof(arr[0] ));

    {
         struct starpu_task *task = starpu_task_create();

         task->synchronous = 1;
         task->cl = &(objSt_vector_scale->cl_vector_scale);
         task->handles[0] = arr_handle;
         task->cl_arg = &arg_vector_scale;
         task->cl_arg_size = sizeof(ROA_vector_scale);

         if(x2p_nextcall_bitvector != 0){
         // cout <<"next_bit_vector = "<< x2p_nextcall_bitvector << endl;
         unsigned long int x2p_where = x2p_nextcall_bitvector & objSt_vector_scale->
             cl_vector_scale.where ;
         // cout << "x2p_where = " << x2p_where <<endl;
         task->where = x2p_where;
         }

         /* execute the task on any eligible computational ressource */
         int ret = starpu_task_submit(task);

         if (ret == -ENODEV)
         {
             fprintf(stderr, "ERROR: No worker may execute this task\n");
             x2p_nextcall_bitvector = 0;
             return;
         }
     }

     starpu_data_unregister(arr_handle);

     x2p_nextcall_bitvector = 0;

}

 #endif
```

Listing 2.11: `makefile` – generated by the composition framework.

```
CFLAGS += $(shell pkg-config --cflags starpu-1.3) -DSTARPU
LDFLAGS += $(shell pkg-config --libs starpu-1.3)

all: main

main : main.o vector_scale_cpu.o vector_scale_cuda.o
        g++-7 main.o vector_scale_cpu.o vector_scale_cuda.o $(CFLAGS) $(LDFLAGS) $(shell pkg-
            config --libs cuda-9.2 cudart-9.2) -lOpenCL -o main
```

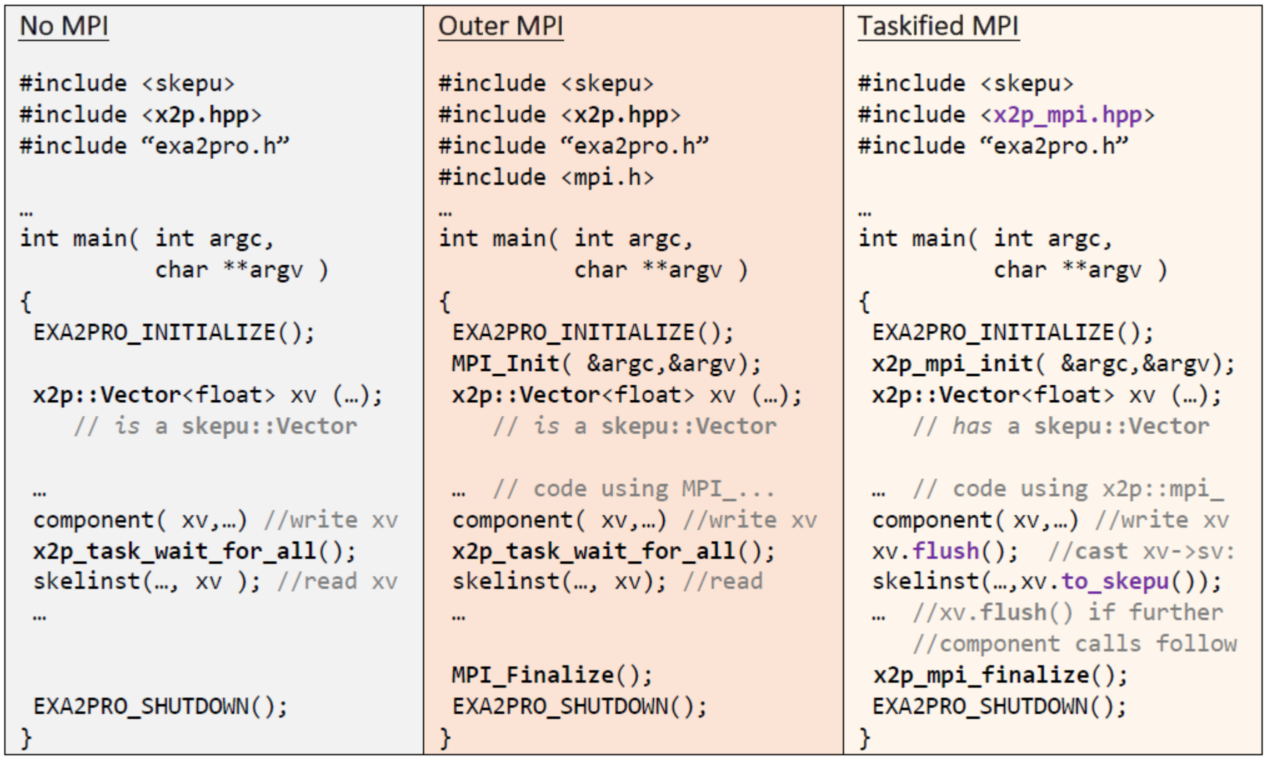
Figure 2.1: There are two different ways of using MPI with multi-variant components [10].

No MPI

```
#include <skepu>
#include <x2p.hpp>
#include “exa2pro.h”

…
int main( int argc,
          char **argv )
{
 EXA2PRO_INITIALIZE();

 x2p::Vector<float> xv (…);
   // is a skepu::Vector


 …
 component( xv,…) //write xv
 x2p_task_wait_for_all();
 skelinst(…, xv ); //read xv
 …


 EXA2PRO_SHUTDOWN();
}
```

Outer MPI

```
#include <skepu>
#include <x2p.hpp>
#include “exa2pro.h”
#include <mpi.h>
…
int main( int argc,
          char **argv )
{
 EXA2PRO_INITIALIZE();
 MPI_Init( &argc,&argv);
 x2p::Vector<float> xv (…);
   // is a skepu::Vector

 …  // code using MPI_...
 component( xv,…) //write xv
 x2p_task_wait_for_all();
 skelinst(…, xv); //read
 …

 MPI_Finalize();
 EXA2PRO_SHUTDOWN();
}
```

Taskified MPI

```
#include <skepu>
#include <x2p_mpi.hpp>
#include “exa2pro.h”

…
int main( int argc,
          char **argv )
{
 EXA2PRO_INITIALIZE();
 x2p_mpi_init( &argc,&argv);
 x2p::Vector<float> xv (…);
   // has a skepu::Vector

 …  // code using x2p::mpi_
 component( xv,…) //write xv
 xv.flush();  //cast xv->sv:
 skelinst(…,xv.to_skepu());
 …  //xv.flush() if further
    //component calls follow
 x2p_mpi_finalize();
 EXA2PRO_SHUTDOWN();
}
```

```
main.o : main.c
        g++-7 main.c $(CFLAGS) -std=c++11 -c -o main.o
vector_scale_cpu.o : ./vector_scale/CPU/vector_scale_cpu.c
        g++-7 ./vector_scale/CPU/vector_scale_cpu.c $(CFLAGS) -c -o vector_scale_cpu.o
vector_scale_cuda.o : ./vector_scale/CUDA/vector_scale_cuda.cu
        nvcc ./vector_scale/CUDA/vector_scale_cuda.cu $(CFLAGS) --compiler-options -
            fpermissive -c -o vector_scale_cuda.o

clean:
        rm -f main *.o *~

distclean: clean
        rm -f *.h comir* makefile
        rm -rf perf_models simulation *DFE_SIM
```

2.6 MPI Support

Figure 2.1 summarizes the two different ways of MPI programming with multi-variant components.

Outer MPI operations use the plain MPI API and thus use ComPU and StarPU only at node level. As with SkePU, plain MPI is unaware of the data-containers, and the operations thus operate directly on the payload arrays, as usual. Hence, data-containers are to be flushed before sending and after receiving if their payload arrays are source or destination to the corresponding MPI calls, respectively.

Taskified MPI operations are implemented atop the MPI binding of the StarPU runtime and have defered execution semantics, see below. They only work on x2p containers.

Outer MPI and Taskified MPI are incompatible with each other [7] and thus mutually exclusive,

[7]Apart from eager vs. deferred execution of the communication operations, outer MPI and Taskified MPI use different versions of the x2p data-containers.

i.e. should not be mixed in the same application. The programmer needs to decide beforehand which one to use.

Neither `MPI_...` nor `x2p::mpi_...` communication operations are applicable inside components. In particular, we do not support MPI implementations of components.

The `x2p::mpi_...` functions are defined in the header file `x2p_mpi.hpp` that needs to be included for EXA2PRO taskified-MPI code.

We also provide C versions of the `x2p::mpi::...` functions, prefixed `x2p_mpi...`, see e.g. Figure 2.1.

Note that the (C++) communication operations in Taskified MPI are polymorphic in the container shape and element type, while their counterparts in the C API are not, and only base-type element types are supported in the C version.

2.6.1 Using Outer MPI

```
// Outer MPI
#include <mpi.h>
...
skepu::Vector<State> state(neurons), returned(task_size);
skepu::Vector<Constants> constants(neurons);
...
SimulateTimestepPerNeuron( returned, local_wm, state, constants, ... );
returned.flush();
state.flush();
MPI_Allgatherv( returned.data(), task_size, MPI_STATE,
                state.data(), send_cnt, displs, MPI_STATE, MPI_COMM_WORLD );
...
```

Outer MPI code uses plain MPI operations such as `MPI_Send` and `MPI_Recv`, which describe the elements to be communicated by triplets of a raw pointer to the first element, the element count and the MPI element type code. This can be easily used with EXA2PRO data containers where the address of the first element can be accessed by `*container*.data()`.

2.6.2 Array Distribution and Distributed Containers in Taskified MPI

In Taskified MPI code, arrays can be distributed across multiple MPI ranks such that the partitions can be treated as different Vectors. When distributing an array, the programmer should specify a distribution function (a function mapping global array indices to MPI ranks), see D2.3 [10]. The following predefined distribution function is supported in the current prototype:

```
x2p::partition_mode::X2P_BLOCK
```

which models a 1D block distribution across the p MPI processes active at the point of specifying the distribution, mapping element j to rank j/p.

2.6.3 Sending and receiving messages in Taskified MPI

In Taskified MPI[8] the programmer uses variants prefixed `x2p::mpi_..` of the MPI operations, which become (inner CPU_SEQ) tasks for the runtime system, in the same way as asynchronous calls to EXA2PRO components. These "submission" variants of the MPI operations are asynchronous and just submit the operation within the task graph. The difference from non-blocking MPI operations

[8]Taskified MPI is only applicable to `x2p` containers, not to SkePU containers.

is that the latter require to later check whether the operation is completed. With these "submission" variants of the MPI operations, the EXA2PRO programmer can write code with submit-and-forget behaviour. For example, with a series of vectors, we can apply convolution on each of them:

```
// Taskified MPI
for (i= ... ) {
 x2p::mpi_recv( vectors[i], sourcerank, tag, NULL );
 convolution( vectors[i] );
 x2p::mpi_send ( vectors[i], destinationrank, tag );
}
```

This submits to the runtime system small task graphs composed of reception, computation, send, which will be processed by the runtime system as they become data-ready.

2.6.4 Example

The following source code (see also D2.3 [10]) shows an iterative 1D stencil computation in Taskified MPI, where the multi-variant component `stencil` implements one stencil sweep over the local partition. The component itself has the sequential implementation in C++ shown in Listing 2.12.

Listing 2.12: Sequential implementation of a 1D stencil component.

```
inline
void stencil( x2p::Vector<float> & out, x2p::Vector<float> & left,
              x2p::Vector<float> & in, x2p::Vector<float> & right ) noexcept
{
 auto out_it { out.begin() }; // iterators
 auto in_it { in.begin() };
 auto const in_end{ in.end()-1 };
 *out_it = (left(0) + *in_it + *(in_it+1)) / 3;
 ++out_it;
 ++in_it;
 for (; in_it != in_end; ++in_it, ++out_it)
    *out_it = (*(in_it-1) + *in_it + *(in_it+1)) / 3;
 *out_it = (*(in_it-1) + *in_it + right(0)) / 3;
}
```

Further implementations (e.g., CUDA etc.) of the component are to be provided by the programmer.

The outer code (with the main function and all message passing code) is then as shown in Listing 2.13.

Listing 2.13: Distributed 1D stencil computation using Taskfied MPI.

```
#include <random>
#include <sstream>
#include <x2p_mpi.hpp>
#include "exa2pro.h"
void inline init_vector( x2p::Vector<float> & v, std::vector<float> const & iv ) noexcept
{
 auto v_it = v.begin();
 auto iv_it = iv.begin() + (x2p::mpi_rank() * v.size());
 auto iv_end = iv_it + v.size();
 for (; iv_it != iv_end; ++v_it, ++iv_it)
 *v_it = *iv_it;
}

void inline calculate_expected_result( std::vector<float> & v, int const K ) noexcept
{
 auto buffer{ std::vector<float>(v.size()) };
 auto const v_end{ v.end() };
 auto const buff_end{ buffer.end() };
 for (int i(0); i < K; i+=2) {
    auto v_it{ v.begin() };  // iterators
    auto buff_it{ buffer.begin() };
   *buff_it = (*(v_end-1) + *v_it + *(v_it+1)) / 3;
   ++buff_it;
   ++v_it;
   for (; v_it != v_end-1; ++v_it, ++buff_it)
      *buff_it = (*(v_it-1) + *v_it + *(v_it+1)) / 3;
   *buff_it = (*(v_it-1) + *v_it + *v.begin()) / 3;
   v_it = v.begin();
   buff_it = buffer.begin();
   *v_it = (*(buff_end-1) + *buff_it + *(buff_it+1))/3
   ++buff_it;
   ++v_it;
   for (; buff_it != buff_end-1; ++v_it, ++buff_it)
   *v_it = (*(buff_it-1) + *buff_it + *(buff_it+1)) / 3;
   *v_it = (*(buff_it-1) + *buff_it + *(buffer.begin())) / 3;
 }
}

void inline work() noexcept
{
 int constexpr N { 5 };
 int constexpr K { 4 };
 int mpi_size { x2p::mpi_size() };
 int rank { x2p::mpi_rank() };
 auto left_rank = (mpi_size + rank - 1) % mpi_size;
 auto right_rank = (rank+1) % mpi_size;
 x2p::DistVector<float> dist_vec( mpi_size * N, x2p::partition_mode::X2P_BLOCK );
 auto & v { dist_vec.partition(rank) };
 auto & l { dist_vec.partition(left_rank) };
 auto & r { dist_vec.partition(right_rank) };
 x2p::DistVector<float> buffer( mpi_size * N, x2p::partition_mode::X2P_BLOCK );
 auto & bv { buffer.partition(rank) };
 auto v_first { x2p::VectorWindow<float>( v, 0, 1 ) }; // window and iterator
 auto v_last { x2p::VectorWindow<float>( v, N-1, 1 ) };
 auto bv_first { x2p::VectorWindow<float>( bv, 0, 1 ) };
 auto bv_last { x2p::VectorWindow<float>( bv, N-1, 1 ) };
 auto l_view { x2p::VectorWindow<float>( l, N-1, 1 ) };
 auto r_view { x2p::VectorWindow<float>( r, 0, 1 ) };
 auto left_tag{ x2p::mpi_tag() };  // get a unique MPI tag
 auto right_tag{ x2p::mpi_tag() };
 for (int i(0); i < K; i += 2) {
   // iterative loop over stencils, unrolled 1x:
   x2p::mpi_send ( v_first, left_rank, left_tag);
   // send left boundary element
   x2p::mpi_send ( v_last, right_rank, right_tag); // send right boundary element
   x2p::mpi_recv ( l_view, left_rank, right_tag);
   // receive left overlap element
   x2p::mpi_recv ( r_view, right_rank, left_tag);
   // receive right overlap element
   stencil( bv, l_view, v, r_view );
   // local stencil operation from v to bv
   x2p::mpi_send ( bv_first, left_rank, left_tag);
   x2p::mpi_send ( bv_last, right_rank, right_tag);
   x2p::mpi_recv ( l_view, left_rank, right_tag);
   x2p::mpi_recv ( r_view, right_rank, left_tag);
   stencil( v, l_view, bv, r_view );
   // local stencil operation from bv to v
 }
}

int main( int argc, char ** argv ) noexcept
{
 EXA2PRO_INITIALIZE( &argc, &argv );
 // start up local runtime system
 x2p_mpi_init( &argc, &argv );
 // start up taskified MPI support (including MPI)
 work();
 x2p_mpi_shutdown();
 // shut down taskified MPI support (including MPI)
 EXA2PRO_SHUTDOWN();
 // shut down EXA2PRO local runtime system
 return 0;
}
```

2.6.5 Example

The source code in Listing 2.14 of a simple example (`vector_scale` using Taskified MPI) can also be found in the EXA2PRO public repository under

`exa2pro-public/compu/tree/master/examples/mpi_taskified/container_scale`

Listing 2.14: `vector_scale` using Taskfied MPI.

```
#include <iostream>
#include <random>
#include <string>

#include <x2p_mpi.hpp>
#include "exa2pro.h"

auto inline
initialize ( x2p::Vector<int> & v ) noexcept -> void
{
 std::random_device rd;
 std::mt19937 gen(rd());
 std::uniform_int_distribution<int> dist(0,9);
 for(auto & e : v)
    e = dist(gen);
}

auto inline
work ( size_t size ) noexcept -> void
{
 x2p::DistVector<int> dv( size, x2p::partition_mode::X2P_BLOCK);
 auto & local_v = dv.partition( x2p::mpi_rank());
 initialize( local_v );
 std::vector<int> expected(size);
 std::cout << "Expected values: ";
 for (size_t i(0); i < size; ++i) {
    expected[i] = local_v(i) * 10;
    std::cout << expected[i] << " ";
 }
 std::cout << std::endl;
 std::cout << "mpi_size = " << x2p::mpi_size() << std::endl;
 for (int i(0); i < x2p::mpi_size(); ++i)
    vector_scale(dv.partition(i), 10);
 local_v.flush();
 bool errors{false};
 std::cout << "local_v: ";
 for (size_t i(0); i < size; ++i) {
    std::cout << local_v(i) << " ";
    if (local_v(i) != expected[i])
       errors = true;
 }
 std::cout << std::endl;
 if (errors)
    printf("Rank %i: There was an error in the vector scale.\n",
           x2p::mpi_rank());
}

auto inline
print_usage(std::string const name) noexcept -> void
{
 std::cout << "Usage: " << name << " <size>\n";
}

int main(int argc, char ** argv)
```

```
{
 EXA2PRO_INITIALIZE();
 {
    x2p::mpi_init(&argc, &argv);
    if (argc != 2) {
       if (!x2p::mpi_rank())
          print_usage(argv[0]);
          return 1;
    }
    work(std::stoul(argv[1]));
    x2p::mpi_finalize();
 }
 EXA2PRO_SHUTDOWN();
 return 0;
}
```

Note that the actual component implementation is the same as in the single-node case, as all MPI operations and distributed containers exist in outer code only.

2.7 High-Level Checkpointing API

Fault tolerance support in EXA2PRO is implemented by the EXA2PRO runtime system (StarPU) atop the StarPU-MPI interface, where checkpoints are asynchronously performed and sent to a buddy MPI process on a different node, as specified by the programmer. The principle has been described in Deliverable D4.6, and details of the API and implementation in StarPU can be found in Deliverable D3.9.

As our Taskified MPI platform is based on StarPU-MPI, we can also provide fault tolerance support for the high-level programming model, yet only in outer code, namely, within Taskified MPI code. Deliverable D2.3 [10], see also the *EXA2PRO Programming Guide*, defines a high-level API (for outer C++ code), summarized below, that allows the user to define checkpoints, to identify sensitive data structures by attaching their (Taskified-MPI) containers to checkpoints, hints for where and how to save checkpoints, and for loading and restoring checkpoints in the case of a node failure. The API is based on a C++ class for checkpoints, C and C++ functions, and one C preprocessor macro. Where the C API differs from the C++ API, we also show the C API for the checkpointing operations below their C++ counterparts. Pragma syntax for these operations, which might be resolved by the Mercurium precompiler in a possible future extension, is also defined, see below.

Note that high-level checkpointing support is, for obvious reasons, only available for programs using ComPU with Taskified MPI, thus running atop StarPU-MPI on a cluster. It is not available in single-node programs, nor in code using the *outer* MPI mode as outer MPI bypasses StarPU-MPI. Note also that high-level checkpointing has not been tested extensively, but is included for documentation purposes.

The C++ data type `x2p::checkpoint` and its C counterpart `x2p_checkpoint_t` are used to represent *checkpoint objects*. A checkpoint object `cp` contains a list of containers to be saved when checkpointing for `cp` and restoring them from the latest checkpoint on a spare process taking over in the case of a node failure.

A checkpoint object `cp` is also associated with an integer variable that is supposed to contain a step counter that matches checkpoint instances created by checkpointing for `cp`. This design, which follows the checkpointing technique in the EXA2PRO runtime system (see Deliverable D4.6), is well suited for iterative solvers where there could be one checkpoint per iteration. (The runtime system has the option to adaptively skip some of the checkpoint instances in order to decrease amortized checkpointing overhead in the fault-free case, see D4.6). Finally, the checkpoint object `cp` can be associated with a *buddy function*, i.e., a function of type `int -> int` that calculates which other MPI rank is supposed to archive the checkpointed data of the executing MPI rank. There is a built-in default buddy function that will be used if no other choice is made by the programmer. The C++

function resp. C function

```
x2p::checkpoint x2p::checkpoint_declare( int *it, int(*buddyf)(int) )
void x2p_checkpoint_declare( x2p_checkpoint_t *cp, int *it, int(*buddyf)(int) )
```

allocates space and initializes a new checkpoint representation object `cp`, and associates an integer variable (supposed to count iterations, with one checkpoint per iteration) and the buddy function `it` pointed to by `buddyf` with `cp`. If `buddyf` is NULL, the default buddy function (namely, the next higher rank modulo the MPI communicator size) will be used instead. Finally, the checkpoint object `cp` will be registered for management by the runtime system.

```
template<typename Container>
void x2p::checkpoint_register( x2p::checkpoint &cp, Container & c )
void x2p_checkpoint_register_Vector_int( x2p_checkpoint_t *cp, x2p_vector_int *)
```

registers an x2p container such as `x2p::Vector, x2p::Matrix` etc. with the checkpoint object `cp` so it will be automatically saved by the runtime system whenever checkpointing for `cp`. In the C API, similar registration functions as shown for `x2p::Vector` above exist for `Matrix`, `Tensor3`, and `Tensor4` containers. In the C API prototype implementation, the base types int, float and double are supported.

The C++ construct

```
x2p_checkpoint_load( x2p::checkpoint &cp )
{
 ... // initialize containers and the iteration variable registered in cp
}
```

respectively, its counterpart for C,

```
x2p_checkpoint_load( x2p_checkpoint_t *cp )
{
  ... // initialize containers and the iteration variable registered in cp
}
```

are implemented as a C preprocessor macro and guard the initialization of containers associated with checkpoint object `cp` in the compound statement which will be executed in the fault-free case, and otherwise restore the registered containers in `cp` and the registered iteration variable's value in the case of a spare MPI process taking over the rank of a lost process, controlled by the runtime system.

```
void x2p::checkpoint_restore( x2p::checkpoint &cp )
void x2p_checkpoint_restore( x2p_checkpoint_t *cp )
```

is an internal C++ resp. C function to be called by the spare MPI process when resuming a checkpoint from the buddy of the lost MPI process. Finally,

```
void x2p::checkpoint_save ( x2p::checkpoint &cp, int priority )
void x2p_checkpoint_save ( x2p_checkpoint_t *cp, int priority )
```

performs a checkpoint by asynchronously sending the current contents of all data containers attached to the checkpoint object cp to the registered buddy rank for archival. Listing 2.15 shows a simple example code using checkpoints.

Listing 2.15: Checkpointing example

```
int buddy_function( int rank )
// example only, happens to be equal to the default fn
{
   return (rank+1) % nodes;
}

int main( void )
   {
   int iter;
   // iteration counter variable
   x2p::Vector<double> v1;
   x2p::Matrix<double> m1;
   ...
   x2p::checkpoint cp1; // defining a checkpoint object cp1
   cp1 = x2p::checkpoint_declare( &iter, buddy_function );
   x2p::checkpoint_register( &cp1, &v1); // one such call per container
   x2p::checkpoint_register( &cp1, &m1 );
   x2p_checkpoint_load( &cp1 ) // construct guarding a compound statement
   {
      Initialize_v1();
      // init values (e.g. read from file or preset with constants)
      Initialize_m1();
      // ... more container initializations
      iter = 0;
   }
   for ( ; iter < end; iter++) // e.g. outer loop of an iterative solver
   {
      component1_call(v1, m1);
      component2_call( ... );
      component3_call( ... );
      ...
      x2p::checkpoint_save( &cp1, priority );
   }
}
```

Chapter 3

Bibliography

[1] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. 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, 2012.

[2] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.

[3] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *Proc. ParCo-2011 Int. Conference on Parallel Computing, Ghent, Belgium. In: Advances in Parallel Computing vol. 22*, pages 159–166. IOS press, 2012.

[4] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the SkePU skeleton programming library. In *Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.

[5] Johan Enmyren and Christoph W Kessler. 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.

[6] August Ernstsson. SkePU 2: Language embedding and compiler support for flexible and type-safe skeleton programming. Master's thesis, Linköping University, Linköping, Sweden, 2016. LIU-IDA/LITH-EX-A--16/026--SE.

[7] August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019. e5003 cpe.5003.

[8] August Ernstsson and Christoph Kessler. 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, March 2020.

[9] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, pages 1–19, 2017.

[10] Christoph Kessler, August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, Tomas Öhberg, Lazaros Papadopoulos, Sotirios Panagitiou, Nicolas Vandenbergen, Sakis Papadopoulos, Mathieu Haefele, and Samuel Thibault. D2.3 – final prototype implementation of exa2pro high-level

programming interface. EXA2PRO project deliverable report, submitted Oct. 2020, to appear at `https://exa2pro.eu` after approval by the European Commission, 2020.

[11] Christoph Kessler, August Ernstsson, Suejb Memeti, and Johan Ahlqvist. 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, March 2020. ISBN 978-3-902457-55-4.

[12] Christoph Kessler, Lu Li, Aras Atalar, and Alin Dobre. Xpdl: Extensible platform description language to support energy modeling and optimization. In *Proc. 44th International Conference on Parallel Processing Workshops, ICPP-EMS Embedded Multicore Systems, in conjunction with ICPP-2015, Beijing, 1-4 sep. 2015*, pages 51–60. IEEE, 2015. DOI 10.1109/ICPPW.2015.17.

[13] Christoph Kessler, Stavroula Zouzoula, Johan Ahlqvist, August Ernstsson, , Suejb Memeti, Oleg Sysoev, Tobias Becker, Alexander Cramb, and Nils Voss. D3.6 – final version of the composition and performance modelling framework. EXA2PRO project deliverable report, submitted Dec. 2020, to appear at `https://exa2pro.eu` after approval by the European Commission, 2020.

[14] Christoph Kessler et al. SkePU: Autotunable multi-backend skeleton programming framework for multicore CPU and multi-GPU systems. `http://skepu.gitlab.io`.

[15] Mudassar Majeed, Usman Dastgeer, and Christoph Kessler. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *Proc. Int. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA-2013), Las Vegas, USA*, July 2013.

[16] Tomas Öhberg, August Ernstsson, and Christoph Kessler. Hybrid CPU-GPU execution support in the skeleton programming framework SkePU. *J. Supercomput.*, March 2019.

[17] Sotirios Panagiotou, August Ernstsson, Johan Ahlqvist, Lazaros Papadopoulos, Christoph Kessler, and Dimitrios Soudris. Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU. In *Proc. SCOPES'20*. ACM, May 2020.

[18] Oskar Sjöström. Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library. Master's thesis, Linköping University, Linköping, Sweden, 2015. LIU-IDA/LITH-EX-A--15/001--SE.

[19] Sebastian Thorarensen, Rosandra Cuello, Christoph Kessler, Lu Li, and Brendan Barry. Efficient execution of SkePU skeleton programs on the low-power multicore processor Myriad2. In *Proc. Euromicro PDP-2016 Int. Conf. on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece*, pages 398–402. IEEE, February 2016.