EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems

Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I. Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, Samuel Thibault, Athanasios Salamanis, Theodoros Ioakimidis, and Dionysios Kehagias

Abstract—Programming upcoming exascale computing systems is expected to be a major challenge. New programming models are required to improve programmability, by hiding the complexity of these systems from application developers. The EXA2PRO programming framework aims at improving developers' productivity for applications that target heterogeneous computing systems. It is based on advanced programming models and abstractions that encapsulate low-level platform-specific optimizations and it is supported by a runtime that handles application deployment on heterogeneous nodes. It supports a wide variety of platforms and accelerators (CPU, GPU, FPGA-based Data-Flow Engines), allowing developers to efficiently exploit heterogeneous computing systems, thus enabling more HPC applications to reach exascale computing. The EXA2PRO framework was evaluated using four HPC applications from different domains. By applying the EXA2PRO framework, the applications were automatically deployed and evaluated on a variety of computing architectures, enabling developers to obtain performance results on accelerators, test scalability on MPI clusters and productively investigate the degree by which each application can efficiently use different types of hardware resources.

Index Terms—Programming models, skeleton programming, task-based runtime systems, programming productivity, heterogeneous systems, exascale computing

◆

1 INTRODUCTION

THE building of exascale computing systems is a huge investment both financially and in terms of scientific effort. Therefore, it is beneficial to enable more applications to access these systems, or at least, promote their transition towards the exascale computing direction. Towards this end, advanced programming models and environments are needed that will assist developers in addressing critical

- *Lazaros Papadopoulos and Dimitrios Soudris are with the Department of Electrical and Computer Engineering, National Technical University of Athens, 10682 Athens, Greece. E-mail: {lpapadop, dsoudris}@microlab.ntua.gr.*
- *Christoph Kessler, August Ernstsson, and Johan Ahlqvist are with the Department of Computer and Information Science, Linköping University, 58183 Linköping, Sweden. E-mail: {christoph.kessler, august.ernstsson} @liu.se, johah956@student.liu.se.*
- *Nikos Vasilas, Athanasios I. Papadopoulos, and Panos Seferlis are with the Chemical Process and Energy Resources Institute, Centre for Research and Technology Hellas, 57001 Thessaloniki, Greece. E-mail: {nvasilas, spapadopoulos}@certh.gr, seferlis@cperi.certh.gr.*
- *Charles Prouveur is with the Maison de la Simulation, CEA, CNRS, 75794 Paris, France. E-mail: charles.prouveur@cea.fr.*
- *Matthieu Haefele is with the Université de Pau et des Pays de l'Adour, 64000 Pau, France. E-mail: matthieu.haefele@univ-pau.fr.*
- *Samuel Thibault is with the Bordeaux University, 33076 Bordeaux, France. E-mail: samuel.thibault@u-bordeaux.fr.*
- *Athanasios Salamanis, Theodoros Ioakimidis, and Dionysios Kehagias are with the Information Technologies Institute, Centre for Research and Technology Hellas, 57001 Thessaloniki, Greece. E-mail: {asal, theioak, diok}@iti.gr.*

challenges in terms of efficient application deployment, such as maintaining development productivity in the heterogeneous exascale computing environments.

EXA2PRO is a framework developed in the context of the EXA2PRO H2020 EU project [1] and its goal is to allow the transition of more HPC applications to exascale computing, by focusing on development productivity. More specifically, the EXA2PRO framework allows developers to evaluate applications across a variety of different architectures and therefore, to efficiently exploit accelerators, such as GPUs and FPGAs. Thus, EXA2PRO lowers the barrier of access and enables more HPC applications to exploit the exascale computing systems, by hiding their complexity from application developers.

The EXA2PRO framework combines the skeleton programming model with a multi-backend approach to enable portability and programmability across a variety of architectures. It provides high-level software abstractions (skeletons, multi-variant components), through the SkePU tool, that implement data parallel computational patterns commonly found in HPC applications (map, MapReduce, stencil, etc.), which are automatically generated for various backends (OpenMP, CUDA, OpenCL, etc.). The framework encapsulates all low-level tuning and optimization tasks to allow developers to focus on application programming and to automatically evaluate applications across different architectures, instead of manually optimizing and porting to accelerators, thus greatly increasing development productivity. Additionally, the EXA2PRO framework makes advanced programming models more accessible to application developers.

EXA2PRO supports the data-flow programming model and MPI through its runtime system, allowing developers to efficiently exploit large-scale heterogeneous computing systems, by hiding their complexity. More specifically, the EXA2PRO framework integrates the StarPU runtime system, which can be used either as another EXA2PRO backend or it can be used in cases in which the skeletons may not fit the overall software design of the application and StarPU task-based programming model maybe simpler to apply (even if much more involved in terms of lines of code and expressiveness complexity).

The purpose of this work is to highlight the key features of the EXA2PRO framework and to evaluate and demonstrate its development productivity features, based on a set of scientific applications that need to be evaluated in more complex architectures and target their transition to exascale computing. The main contributions are the following:

- We present the EXA2PRO framework, which enables the evaluation of applications in heterogeneous exascale computing systems.[1]
- We demonstrate the use of EXA2PRO in a variety of applications, which target the transition to exascale computing systems and show how the multi-back-end feature of EXA2PRO can improve the development productivity.

The rest of the paper is organized as follows: Section 2 describes the related work. The EXA2PRO framework and its components are presented in Section 3. The evaluation results are analyzed and discussed in Section 4. Finally, in Section 5 we draw conclusions.

2 RELATED WORK

The EPEEC programming framework has similar goals as EXA2PRO [2], which also targets heterogeneous exascale supercomputers. A programming framework, developed in the context of the EPIGRAM-HS project, addresses heterogeneity challenges both in terms of acceleration (GPUs, FPGAs) and in terms or memory (HBM, NVM) [3]. The ASPIDE project develops a programming framework tailored to Big Data applications deployed on exascale computing systems [4]. Programming models and runtimes for exascale are developed in the context of the US Exascale Computing Project, such as enhancements of the MPI standard for a task-based programming model [5]. Kokkos provides high-level abstractions for parallel loop executions, which are mapped to a runtime for achieving performance across various architectures [6]. Although the above libraries and frameworks have similar goals, they focus on different aspects of the programmability of the exascale computing systems, utilize different programming models and attempt to address different exascale computing challenges. The EXA2PRO framework focuses on the programmability of heterogeneous systems and integrates programming models that are different from the aforementioned models: skeleton programming in EXA2PRO is used to implement data-parallel patterns and the data-flow programming model is used to support FPGA-

1. A docker and a singularity container of the framework are available on the EXA2PRO website: https://exa2pro.eu/#developers
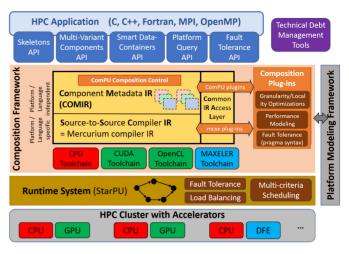

Fig. 1. Overview of the EXA2PRO programming API and framework stack.

based data-flow engines (DFE) [7]. The goal is to enable the transition of more HPC applications towards the exascale computing direction.

OpenACC is an API that targets programmability and portability across diverse architectures and it is based on compiler directives [8]. OpenACC, as well as SYCL and other programming models can be considered complementary to EXA2PRO. However, SkePU skeletons are more high-level and more powerful, as their implementations are not constrained by the given loop nest structure in an OpenACC source program.

Among libraries exposing a task-based programming model, OmpSs [9] PaRSEC [10] and DTD (Dynamic Task Discovery) also provide support for heterogeneous accelerator-based and distributed systems, and support for dynamic task scheduling to some extent. In comparison, the EXA2PRO runtime, StarPU [11], reaches more advanced task scheduling, by achieving compromises between task acceleration, data transfer cost, and energy cost. Its pluggable task scheduling interface allows to integrate state-of-the-art heuristics (e.g., HEFT, HeteroPrio).

3 EXA2PRO FRAMEWORK

The EXA2PRO framework is based on a set of advanced programming models, supported by a toolflow that allows developers to evaluate applications across a variety of architectures and to efficiently exploit accelerators, such as GPUs and FPGAs, thus supporting the transition of applications to exascale computing through increased development productivity. Its software stack is shown in Fig. 1. The EXA2PRO high-level programming model is based on a set of skeletons, multi-variant components, smart containers and other APIs. To apply the EXA2PRO framework to an application, developers are expected to replace the algorithmic pattern of the application with the EXA2PRO high-level programming interface.

The most important features of the EXA2PRO framework are the following:

- It provides a simple sequential API for several common parallel algorithmic patterns.
- The EXA2PRO tools automatically generate optimized and tunable implementations for a variety of
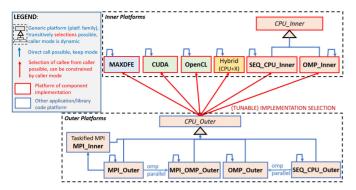
Fig. 2. The EXA2PRO "gearbox": Execution platforms/modes and implementation selection.

heterogeneous systems (implementation variants). Users may provide custom implementations (multi-variant components).

- Suboptimal variants are manually or automatically discarded, while the ones expected to provide better results are forwarded to the EXA2PRO runtime system.
- The EXA2PRO runtime system provides optimized variant selection, scheduling and data transfers to accelerators.

The EXA2PRO framework hides the complexity and the low-level implementation details of large-scale heterogeneous systems from application developers. It encapsulates accelerator code optimizations, scheduling decisions, synchronization issues, data transfer optimization and other low-level tasks. Thus, by providing high-level software abstractions for parallel patterns widely used in HPC applications, the EXA2PRO framework gives the opportunity to more applications to access complex large-scale computing systems.

3.1 EXA2PRO High-Level Programming Model

At the top level, an EXA2PRO application consists of zero or more *multi-variant components*, i.e., modularized functions considered for possible execution on an accelerator, and of *outer code* (Fig. 1). Outer code is CPU code which might also contain explicit parallelization constructs (e.g., MPI) and skeleton calls, see Section 3.2). The components may have different, user-provided implementation variants for different programming and execution platforms in a heterogeneous system (e.g., OpenMP for CPU, CUDA and OpenCL for GPU, etc.), i.e., *inner code*, and are annotated with metadata e.g. about their dependencies and deployment. The EXA2PRO framework synthesizes an executable application from the components and outer code, which will be free to select among implementation variants at runtime (see the selection transitions in the "gearbox" in Fig. 2), depending on the current execution context (such as operand sizes and locations), and supported by the EXA2PRO runtime system. Also skeleton calls allow to select statically or (automatically) at runtime[2] among their implementations, which are provided by the SkePU skeleton framework instead.

---

2. In node-level SkePU, dynamic selection of the expected best implementation based on skeleton call operand sizes and locations can be done in SkePU itself after off-line training [12], [13]; in cluster-level SkePU and for multi-variant components, automated performance modeling and selection are delegated to the runtime system (Section 3.5).

TABLE 1
SkePU Skeleton Descriptions

Skeleton	Description
Map	Element-wise 1D–4D transformation of arbitrary arity
MapPairs	Generic 2D Cartesian-product style computation, arbitrary arity
MapOverlap	1D–4D convolution with arbitrary per-dim. stencil radius
Reduce	Generic reduction
Scan	Generic prefix-sums computation
MapReduce	Fusion of Map and Reduce
MapPairsReduce	Fusion of MapPairs and row-wise or column-wise Reduce

For API properties see Table 2.

The SkePU programming interface and implementation is inherently based on modern C++, especially on template metaprogramming and variadic templates, and user functions must be written in C++. But SkePU also offers a prototype FORTRAN API for access to SkePU data-containers from FORTRAN code, and the SkePU pre-compiler can automatically generate FORTRAN wrapper functions for skeleton instance invocation.

3.2 SkePU

Skeletons are closely related to so-called (parallel) algorithmic *patterns* [14]; in fact, skeletons *implement* patterns. *Multi-backend skeletons* are portable skeletons that realize data-parallel patterns such as *map*, *reduce*, *scan*, *stencil* etc., and that have internally optimized implementations for different resources (CPU, GPU, …) of heterogeneous parallel systems.

In EXA2PRO, we developed the third generation of the skeleton programming framework SkePU [15], which is available both stand-alone as SkePU-3 [16] but also integrated into the EXA2PRO high-level programming model. Compared to SkePU-2 [17], new skeletons and data-container types have been added, the programming interface has been streamlined and the memory consistency model for the smart data-containers has been relaxed. SkePU-3 is available as open-source under a modified BSD license [16].

SkePU [15], [16] provides a single, sequential-like programming interface extending modern C++ with data-parallel skeletons, see Tables 1 and 2. For each skeleton, implementation variants (backends) for sequential C++, OpenMP, CUDA (for Nvidia GPUs) and OpenCL (for all GPUs) are available, and also multi-GPU and hybrid CPU-GPU code can be selected [18]. In EXA2PRO, an additional backend for the MPI bindings of StarPU has been developed. (Access to further platforms such as FPGA accelerator cards can be achieved indirectly via SkePUs interoperability with multi-variant components, which are described in Section 3.3.)

Table 1 gives short descriptions of the current data-parallel skeletons in SkePU, and the interface properties (such as supported arity, dimensionality of operands and special operand access patterns) are summarized in Table 2; for detailed descriptions and illustrations for all skeletons we refer to [15]. In a co-design effort, based on feedback by project partners developing and porting HPC applications

TABLE 2
SkePU-3 Skeletons With Interface Properties and Currently Provided Backends (● Supported, ◐ Partly Supported, ○ Under Work)

Skeleton	Out Args Arity / Dim.	El.-wise In Args Arity / Dim.	Proxy In Args	Uniform In Args	Index Avail.	Special Access Proxies Supported	Node-Level SkePU Backends				Cluster-Level SkePU Backends	
							Seq.	OpenMP	CUDA	OpenCL	MPI+OpenMP	MPI+CUDA
`Map`	Variadic / 1D-4D	Variadic / 1D-4D	Variadic	Variadic	Yes	`MatRow, MatCol`	●	●	●	●	●	●
`MapPairs`	Variadic / 2D	Variadic(x2) / 1D	Variadic	Variadic	Yes	`MatRow, MatCol`	●	●	●	●	●	●
`MapOverlap`	1 / 1D-4D	1 / 1D-4D	Variadic	Variadic	Yes	`Region`	●	●	◐	●	●	○
`Reduce`	1 / 0D-1D	1 / 1D-4D	—	—	—	—	●	●	●	●	●	●
`Scan`	1 / 1D	1 / 1D	—	—	—	—	●	●	●	●	●	○
`MapReduce`	Variadic / 0D	Variadic / 1D-2D	Variadic	Variadic	Yes	`MatRow, MatCol`	●	●	●	●	●	○
`MapPairsReduce`	Variadic / 1D	Variadic(x2) / 1D	Variadic	Variadic	Yes	`MatRow, MatCol`	●	●	○	●	●	○

to the new programming model, SkePU has been extended with new skeletons such as `MapPairs` (generalized Cartesian-product style computation pattern, supporting a variadic number of vector operands in each dimension) and `MapPairsReduce` (fused `MapPairs` with subsequent row-wise or column-wise reduction), and its programming interface has been generalized and streamlined, such as multi-valued return from all `Map`-based skeletons and user functions (as known from Python) and a streamlined interface for the `MapOverlap` (stencil pattern) skeleton. Minor improvements include optional dynamic scheduling in the OpenMP backend of most skeletons.

User functions are problem-specific functions that can be plugged as code parameters into skeletons at instantiation. They must be side-effect-free C++ functions with restricted C++ features (in fact, they are basically C functions with container access proxy objects (see Section 3.4 and the example in Fig. 3) because they must be translatable into platform-specific code with limited or no C++ support, in particular for OpenCL. For the same reason, library functions called from user functions must be portable, i.e., exist on all supported target platforms, or explicitly implemented as user functions too. In-line definition of user functions in skeleton instantiations using C++ lambda functions is supported.

Cluster support in SkePU comes in two different (mutually exclusive) flavors:

- *Outer MPI* (also known as MPI+SkePU) applications use SkePU only at node-level and let the programmer use ordinary MPI calls for explicit inter-node communication, duly framed by `skepu::external` guards (explained later in this paragraph) to preserve sequential semantics and guarantee memory consistency for communicated operands that are managed by SkePU data-containers. For skeleton calls in outer MPI code, the MPI backend of the skeleton is disabled.
- *Inner MPI* applications have MPI-free, plain sequential-looking SkePU source code where all cluster-level parallelism is exploited by the SkePU MPI backend working on distributed data-containers, hence all communication is implicit.

As an illustrating example, Fig. 3 shows an excerpt of a brain modeling simulation code in SkePU (see also Section 4.3). It is a time-driven simulator, in which each neuron accumulates the interaction of adjacent neurons and then updates its current state. The user function `Simulate-Neuron_Skepu` is used in an instantiation (line 29) of the `Map` skeleton, here with no (<0>) element-wise accessed

operands. The user function takes as its main operands a weight matrix `m` and the `state` vector, prepared for random access with the `Mat` access proxy (alternatively, row-wide random access with `MatRow`), resp. `Vec` access proxy

```
1  #include <skepu>
2  ...
3
4  // definition of a user function:
5  State SimulateNeuron_Skepu (
6    skepu::Index1D row,
7    const skepu::Mat<float> m,      // Matrix, random access
8    const skepu::Vec<State> state,  // Vector, random access
9    const skepu::Vec<Constants> constants,
10   float time, float dt )
11  {
12    size_t const i = row.i;  // get context for i-th neuron
13    State const & statePrev = state(i); // current state
14    float iGapTotal = 0; // interaction with other neurons
15    for (size_t j = 0; j < m.cols; j++)
16      iGapTotal += fGap( m(i,j), statePrev.v, state(j).v);
17    // calculate the next state of the neuron:
18    State stateNext = InnerDynamics_Integrate(
19            constants(i), statePrev, iGapTotal, time, dt);
20    return stateNext;
21  }
22
23  int main( int argc, char *argv[] )
24  { ...
25    auto spec = skepu::BackendSpec{  // select by user:
26            skepu::Backend::typeFromString(argv[5])};
27
28    // instantiate Map skeleton to build a "function":
29    auto SimulateTimestepPerNeuron skepu::Map<0>(
30                            SimulateNeuron_Skepu );
31    SimulateTimestepPerNeuron.setBackend(spec);
32    ...
33
34    // data-containers for the data sets of the problem:
35    skepu::Vector<State> state(neurons), state2(neurons);
36    skepu::Vector<Constants> constants(neurons);
37    skepu::Matrix<float> weight_matrix(neurons, neurons);
38    ...
39    // for buffer swapping, use pointers to containers:
40    auto *pOne = &state;
41    auto *pTwo = &state2;
42    ...
43
44    // run the simulation:
45    for (int t = 0; t < steps; t++) {
46      float time = t * dt; // time point in simulation
47      // call the skeleton instance:
48      SimulateTimestepPerNeuron( *pTwo, weight_matrix,
49              *pOne, constants, time, dt);
50      std::swap( pOne, pTwo ); // swap the buffers
51    }
52    ...
53
54    // output:
55    if (write_to_file) {
56      skepu::external( skepu::read(*pOne), [&]
57      { ...
58        for (size_t i = 0; i < neurons; i++)
59          fprintf(..., "%+03.6f\t", State((*pOne)(i)).v);
60      } );
61    }
62  }
```

Fig. 3. Excerpt of the brain modeling simulation code in SkePU.

in the user function; see Section 3.4 for SkePU data-container and access proxy types).

The skeleton instance is called in line 48. By default, SkePU selects the most parallel backend, but here the user explicitly prescribes a backend (line 31) passed in as a command-line argument string, which is set in the `spec` object attached to the skeleton instance `SimulateTimestep-PerNeuron`. Alternatively, the selection autotuning mechanism of SkePU could be used.

As illustrated in the example code snippet in Fig. 3, SkePU programs have three different code scopes: *managed scope* (user functions, with restricted functionality due to strong portability requirements), *unmanaged scope* (code outside `skepu::external`, where data-containers can be defined and skeletons be instantiated and called), and *external scope* (guarded code within `skepu::external`. `skepu::external` is used to frame I/O and MPI operations that access data managed by data-containers to make sure that, given the weak memory consistency model of SkePU-3 in non-managed code, data to be communicated is valid in main memory. The last part in Fig. 3 (lines 54–61) illustrates this: For the data-container list in the optional `read` argument of `skepu::external`, SkePU makes sure to flush any possibly software-cached elements of these data-containers back to main memory, as some may still reside (only) in device memory after being written by a skeleton call (which is in general, such as here, statically unknown). For details we refer to [15].

Where a skeleton does not fit well, e.g., for a parallel loop with irregular, indirect access pattern through a permutation index vector, an overly general skeleton such as a `Map<0>` with an access proxy for a random-access operand instead of an element-wise (or, e.g., row-wise) accessed operand has to be used, which leads to higher communication volume for distributed memory (GPU, cluster) execution and also prevents execution on accelerators where the indirectly accessed data-container does not fit in device memory in its entirety.

3.3 Multi-Variant Components

Multi-variant components are modularized, user-provided functions that may have multiple, user-provided implementation variants written using different programming models such as C++, OpenMP, CUDA, OpenCL or MAXJ (i.e., the programming model of DFEs), and that are annotated with metadata in an XML based annotation framework to support target platform-specific code generation and deployment. EXA2PRO extends this concept from earlier work [19] by adding new target platform types (MPI, Maxeler DFE) and meta-modeling new function properties such as operand access patterns. The deployment toolchain for this is called *ComPU*.

EXA2PRO multi-variant components are interoperable with SkePU skeleton instances as they work on the same data-container types (Section 3.4), see also Fig. 4 for an illustration of SkePU-ComPU interoperability, and provide a flexible escape mechanism for cases where SkePU skeletons do not fit (well) a computation's structure.

3.4 Smart Data-Containers and Access Proxies

Smart data-containers are high-level STL-based generic abstractions for array-based data structures passed as

Fig. 4. Principle of interoperability between SkePU skeleton code and ComPU multi-variant components, using the `x2p` equivalents of the `skepu` data-container types.

operands to and from calls to skeletons and multi-variant components. We call them "smart" as they can transparently perform run-time optimizations for data transfer [20], [21], data locality [22] and memory management. In EXA2PRO (and SkePU-3), currently four such data-containers are supported for 1D to 4D data abstractions: `Vector<...>` (generic 1D vector type similar to STL `std::vector`), `Matrix<...>` (generic dense matrix), `Tensor3<...>` and `Tensor4<...>` (generic 3D and 4D dense tensor, respectively).

In addition, there are a number of different container access proxies that are to be used in managed scope and that encode different access patterns, which help SkePU to reduce communication effort. For example, the most generic `Mat<...>` container access proxy allows for random access to any element of a `Matrix<...>` operand passed to a skeleton instance call, while the more specific `MatRow<...>` and `MatCol<...>` access proxies allow random access only within one row or column of a `Matrix<...>` operand respectively. For pure element-wise access, no container proxy is needed at all on the user function side. For details see [15].

3.5 Runtime System: StarPU

EXA2PRO leverages the StarPU [11] task-based runtime system. StarPU aims at providing optimized application execution over large clusters of heterogeneous systems, such as composed of CPUs, GPUs and FPGAs. It uses a task-based programming paradigm which captures high-level information from the application, and allows its scheduler to be very well informed of the computation performed by the application.

The fundamental model of the StarPU runtime system is the submission of a Directed Acyclic Graph (DAG) of tasks. This is expressed thanks to the notions of *data*, *codelet*, and *tasks*. The dependencies between tasks can be set explicitly, or inferred from the data dependencies for the convenience of the programmer.

In the context of the EXA2PRO framework, StarPU is used as the execution backend for the composition framework. SkePU automatically generates, from the skeleton-based application, a source code that leverages the StarPU interface to submit a task graph. For each task, it provides StarPU with the different implementation variants for the different architectures available on the target platform (CPU, GPU, DFE, ...), by gathering them in a codelet (i.e., the collection of functions which achieve the same computation, thus various *implementations variants* for the computation). Similarly, the different implementation variants provided by the application in the multi-variant components are exposed in StarPU codelets.

From the description of the task graph by SkePU and the composition framework, the StarPU runtime system

TABLE 3
Applications Used to Evaluate the EXA2PRO Framework and Computing Systems Specifications

Application	Motivation	EXA2PRO tool(s) applied	Metrics	Computing Systems CPU/GPU/DFE Specs
CO_2 capture	(i) Parallelize calculations of CO_2 capture process model (ii) Evaluate OpenMP, CUDA and StarPU-MPI	(i) SkePU: Map skeleton (ii) StarPU: (through SkePU StarPU-MPI support)	(i) Speedup on single node compared to original (ii) Scalability	(i) Local cluster: Xeon E-2174G (4 cores @3.8GHz), NVIDIA Quadro P620 (ii) ARIS: Xeon E5-2680v2 (20 cores @2.8GHz) (iii) Piz Daint: Xeon E5-2690v3 (12 cores @2.6GHz), NVidia Tesla P100
Metalwalls	(i) Evaluate SkePU, StarPU and DFE implementations (ii) Evaluate scalability through SkePU, StarPU	SkePU: MapPairsReduce StarPU	Speedup, scalability and Perf./Watt compared to original	(i) Intel Haswell E5-2660v3 (12 cores @2.6GHz), Maxeler DFE Max5C (ii) Piz Daint: Xeon E5-2690v3 (12 cores @2.6GHz), NVidia Tesla P100
Brain Modeling	Evaluate OpenMP, GPU and StarPU-MPI implementations	SkePU: Map skeleton with MatRow access proxy	(i) Execution time on single node compared to original (ii) Scalability	(i) Local Cluster: 2x Intel Xeon Gold 6138 (40 cores @2GHz) NVidia Tesla V100 GPU (ii) Tetralith cluster: 2x Xeon Gold 6130 (2x16 cores @2.1GHz)
ODE solver	Evaluate SkePU with an iterative data-parallel application with a long dependence chain	SkePU: Map, Reduce, MapReduce, MapOverlap	(i) Speedup on single node (ii) Scalability	Tetralith cluster: 2x Xeon Gold 6130 (2x16 cores @2.1GHz)

orchestrates the execution of tasks among the different execution units and memory locations. Since it has a complete vision over the data and the tasks to be computed, it is able to provide the application with a flurry of features *without further effort from the application programmer*, such as optimized task scheduling or data transfer overlapping.

4 EVALUATION

The EXA2PRO framework was applied to four HPC applications: CO_2 capture process design and control, a supercapacitor simulator (Metalwalls), a neural simulator (brain modeling) and an Ordinary Differential Equations (ODE) solver. The applications were selected based on the following two criteria: i) To belong to different application domains ii) To be HPC applications currently deployed in HPC clusters, which aim at evaluation on heterogenous large-scale computing systems. Specific tools of the EXA2PRO framework were applied to each application, based on each application requirements, as shown on Table 3.

4.1 CO_2 Capture Process Design and Control

This application pertains to the design of efficient and robust CO_2 capture systems. The employed CO_2 capture process models and overall design optimization approach are described in [23]. The goal was to test for the first time advanced programming models, to evaluate the application on heterogeneous systems and large number of nodes and to assess its scalability and speedup. The size of the solved problems ranges from 4×10^4 to 4×10^6 algebraic equations, whereas tests are performed in up to 1000 CPU threads and to GPUs in a local cluster, the ARIS and Piz Daint supercomputers. The Map skeleton of SkePU is used as illustrated in Fig. 5, where it is compared to the original, sequential code.

In the CO_2 capture case, the core operation ported to SkePU is represented by the non-linear system of algebraic equations that are used to model the chemical process system [23]. The ported functions are used in a sequence of two calculation stages. In the first stage, the functions are used within a constrained optimization problem formulation, to determine the optimum solution under steady-state operation. In the second stage, the Karush-Kuhn-Tucker optimality conditions [24] are used to transform the problem into an equivalent algebraic formulation [25]. This is used as

part of a homotopy-continuation method that employs a predictor-corrector numerical technique to identify an optimum solution under external variability, within a control structure. In both stages, the equality and inequality constraints of the constrained optimization problem and their gradients are implemented in SkePU. The algorithms used in the two stages are IPOPT [26] and PITCON [27]. The former calls the functions in SkePU to calculate the Hessian and to perform the Newton step for the solution of the nonlinear equations.

Fig. 6a depicts the speedup based on the total execution time, using SkePU parallel backends over single-core execution, to evaluate the CO_2 capture model for an increasing model size. The total execution time comprises the times for the following tasks: (a) The time to create-allocate a SkePU vector to store the output. (b) The call to the SkePU skeleton (c) The SkePU flush operation to make sure that the vector's elements are valid in main memory. In contrast with (a) and (b), this task has an impact only when a GPU backend is used.

Fig. 6b shows the time performance of each task that makes up the total time reported in Fig. 6a. The total time used in the speedup bar for OpenMP is the sum of the time for the OpenMP call task (orange line) and the time for the SkePU vector create task (blue line). The total time used in the speedup bar for CUDA is the sum of the time for the CUDA call task (green line), the time for the CUDA flush task (red line) and the time for the SkePU vector create task (blue line). The OpenCL case is not discussed as the performance is similar to that of CUDA.

In the OpenMP case, when the mathematical model comprises fewer equations, the total time is mainly due to the OpenMP call task (orange line), with the SkePU vector create task having a much smaller contribution. When the number of equations increases, the OpenMP call task is still slower but the vector create task has a significant contribution to the overall time. This behavior leads to an increase in the total time, which is reflected in the reduction of the speedup shown in Fig. 6a. In the CUDA case, it appears from Fig. 6b that for fewer equations the total time is dominated by the CUDA call task (green line). As the number of equations increases, there is an intense change (increase) in the slope of the CUDA skeleton call task (green line), indicating the additional effort needed to solve a problem of larger size. The similar execution time between CUDA and

```
1  ! Original kernel:
2  ...
3  type(Modules) :: m(MODEL_SIZE)
4
5  ! Declaration of variables v, e, pr, coef, hv
6  ...
7
8  do i = 1, MODEL_SIZE
9    ! Core operations of CO2 capture model
10   get_variables(jvar(i, ABS_TOP), &
11     jfeed(i, ABS_TOP), x, v)
12   get_enthalpy(v, e)
13   lagrange(v%leng, v%presbot, coef, pr)
14   get_helper_variables(v, pr%prescp, hv)
15   ev_f(v, e, pr, coef, hv, m(i)%f_abs_top)
16   ... ! For other modules in the process
17 end do
18 ...
```

```
1  // EXA2PRO applied to the kernel:
2  #include <skepu>
3  ...
4  Modules co2_model_skepu( // definition of user function
5    skepu::Index1D index,
6    const skepu::Vec<int> jvar,
7    const skepu::Vec<Feed> jfeed,
8    const skepu::Vec<Vars> vars)
9  {
10   Modules m;
11   Feed feed_ = jfeed.data[index.i];
12   const Vars vars_ = vars.data[index.i];
13
14   // Declaration of variables v, e, pr, coef, hv
15   ...
16
17   int jvar_ = jvar.data[ABS_TOP];
18   int *jfeed_ = &feed_.abs_top[0];
19   // Core operations of CO2 capture model
20   get_variables(jvar_, jfeed_, vars_.x, &v);
21   get_enthalpy(&v, &e);
22   lagrange(v.leng, v.presbot, &coef, &pr);
23   get_helper_variables(&v, pr.prescp, &hv);
24   ev_f(&v, &e, &pr, &coef, &hv, &m.f_abs_top[0]);
25
26   ... // For other modules in the process
27   return m;
28 }
29 ...
30 extern "C" void x2p_skepu(
31   const int n, const double x[],
32   const int m, double f[])
33 {
34   skepu::Vector<Modules> vector(MODEL_SIZE);
35   // instantiate Map skeleton to build a "function"
36   auto calculate = skepu::Map<0>(co2_model_skepu);
37
38   // call the skeleton instance
39   calculate(vector, jvar_skepu_vec(),
40     jfeed_skepu_vec(), vars_skepu_vec(n, x));
41
42   vector.flush();
43   copy_skepu_vector_to_fortran(vector, f);
44 }
```

Fig. 5. Original and EXA2PRO CO_2 capture code.

Fig. 6. CO_2 capture: (a) Speedup using SkePU for various backends on a single node, considering all tasks of Fig. 5 , (b) Break down of time for tasks create of SkePU vector, call with OpenMP, call with CUDA and flush with CUDA. (Results on Local Cluster (single node) for both figures).

Fig. 7d shows the execution time, in a log-log graph, of 1000 simulations of a CO_2 capture process model consisting of 4×10^8 equations for an increasing number of CPU cores and up to 64 nodes. Overall, the method scales nicely up to 200 cores as it is near to the ideal line. However, as the number of cores increases, and a fixed workload is distributed to the parallel cores, the task size per core decreases. In this respect, the application does not scale above 200 cores, since the task size becomes lower than 1ms and the runtime overhead per task, as explained in earlier works [28], has significant impact on the execution time. Therefore, even if we run experiments above 1000 cores, no speedup is expected. The model of the CO_2 capture process is flexible and enables the representation of phenomena at different levels of fidelity. The considered range of equations represents typical models of medium to higher fidelity, exhibiting significant speedup and scalability compared to the original implementation.

The effort of applying SkePU to the CO_2 capture was performed by a senior engineer, already familiar with the application, that required about 2 weeks of training on EXA2PRO and 4 weeks of development.

4.2 Supercapacitor Simulation

Metalwalls is a classical molecular dynamic code dedicated to the accurate simulation of electrochemical systems like supercapacitors, devices able to store energy under electrostatic

OpenMP at or above 400×10^3 equations, is due to the very efficient CPU of the local cluster, as opposed to its less efficient GPU (see also Fig. 7c). The speedup is considerably higher for a larger number of equations, because the larger problem size is executed considerably faster in the GPU, compared to the single CPU. This is despite that the actual CUDA call time increases as the number of equations increases.

Fig. 7a shows that when only the call task is considered, speedup is observed due to the use of SkePU in both the OpenMP and CUDA cases. The same pattern also appears in Fig. 7b, but Piz Daint enables higher speedup, especially for the CUDA case and large number of equations. The reason, is that, as shown in Fig. 7c, the Piz Daint GPU (Tesla P100) is much faster than the local cluster GPU (Quadro P620).
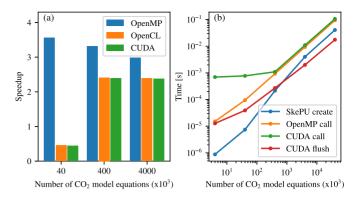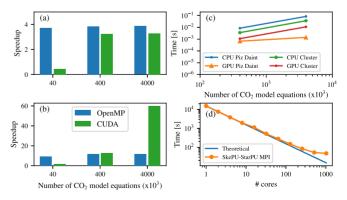

Fig. 7. CO_2 capture: (a) Speedup when only the call task is considered in all cases. (Local cluster) (b) Speedup for increasing number of equations when only the call task is considered (Piz Daint). (c) Execution time on CPUs and GPUs of Piz Daint and local cluster (d) Execution time on ARIS using SkePU-StarPU MPI backend for multiple MPI nodes.

```
1  // Original kernel:
2  double V[num_atoms], z[num_atoms], q[num_atoms];
3  for (i = 0; i < num_atoms; i++) {
4    vi = 0.0;
5    for (int j = 0; j < num_atoms; j++) {
6      zij = z[j] - z[i];
7      pot_ij = exp(-zij*zij;) + zij*erf(zij));
8      V[i] = V[i] - q[j] * pot_ij;
9    }
10 }
```

```
1  // SkePU-ized code:
2  [[skepu::userfunction]]
3  real_t map_function( skepu::Index2D i, const real_t zi,
       const real_t zj, skepu::Vec<real_t> q){
4    real_t zij = zi - zj;
5    real_t qj = q[i.col];
6  return - qj * ( exp(-zij*zij) + zij*erf(zij) );
7  }
8
9  [[skepu::userfunction]]
10 real_t plus (real_t a, real_t b) { return a + b; }
11
12 auto pairs2reduce = skepu::MapPairsReduce(map_function,
       plus);
13 pairs2reduce.setReduceMode(skepu::ReduceMode::RowWise);
14 pairs2reduce(V,z, z, q);
```

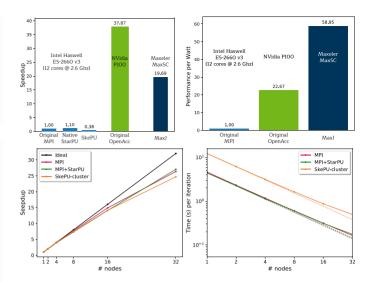Fig. 8. Original and SkePU-ized code for the serial version of a single kernel of Metalwalls.

Fig. 9. Metalwalls: Comparison of software technologies on Metalwalls in terms of execution time (top left), performance per Watt (top right), multi-node speedup (bottom left) and multi-node time to solution on Piz Daint.

form. The typical simulated system is made of two carbon electrodes immersed in an ionic liquid. At each time step, the simulation computes with a matrix-free conjugate gradient the charge density on electrodes such that they conserve a constant potential which is the heart of the mini-app studied in this paper. Details on the code and references to the model can be found in [29].

The motivation of using EXA2PRO for this application is to evaluate the programming effort to rewrite its compute intensive part and measure to which extent the original good performance can be retrieved with SkePU, StarPU and MaxJ (i.e., the programming model of DFEs).

The original code is written in Fortran 90 and is parallelized with MPI. With the introduction of OpenACC directives, a single GPU version is also available. Fig. 8 shows how its serial version has been rewritten in the SkePU framework with a single *MapPairReduce*. The code complexity is similar to serial code while enabling its parallel execution on CPU and GPU. In order to maximize data locality, the original implementation computes by blocks the contribution, of each atom pairs, to the electrostatic potential V, instead of spanning the whole atoms array on both loops as shown here. Blocks are then distributed among the different MPI ranks to enable parallel computations on cores and nodes. As there is no data distribution, all ranks have all data and a single MPI_Allreduce aggregates all contributions and updates all ranks. The StarPU porting introduces a shared memory level of parallelization. It turns each block computation into a task and the runtime distributes them on the different cores within a node. Finally, the DFE implementation required to rethink completely the algorithm in order to match FPGA constraints. All initialisation and setup phases were left untouched, but the roughly 2K lines of C++ for the computationally intensive kernels have been rewritten in the MaxJ language, compiled with the Maxeler toolchain and linked with the original code as a dynamic library.

These implementations have been evaluated on the test-case used in production in [30] that contains 42490 electrode atoms. In the context of material science for supercapacitors, this is a large test case. It is however small considering computing platform capabilities and is thus a limiting factor for extreme scalability.

Fig. 9 top left shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10 percent. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive. Fig. 9 top right shows the performance per Watt behaviour of these implementations. The 50W TDP to operate the FPGA compared to 250W TDP of a GPU makes it more than two times more efficient in terms of performance per watt. This factor is above 50 if we compare to the CPU and makes FPGA a relevant device for the exascale era. Fig. 9 bottom left shows the multi node speedup normalized to the original pure MPI version. All versions scales very well up to 32 nodes (384 cores). Finally, Fig. 9 bottom right shows execution times and the ordering of Fig. 9 top left is respected.

In terms of programming effort, the compute intensive part of the application (2000 LOC in total) was successfully ported to SkePU and StarPU programming models in about two weeks. The resulting code base of the StarPU implementation (650 additional LOC) has a higher complexity compared to the original MPI implementation, while the
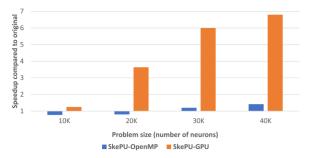
Fig. 10. Brain modeling: Speedup on local cluster (single node) of SkePU-OpenMP and SkePU-GPU compared to original OpenMP implementation.

SkePU one (450 specific LOC that replaces original kernels) is much simpler and slightly shorter as it is very close to a serial implementation. Earlier works that evaluate the complexity of StarPU implementations can be found in the literature [28]. The DFE porting with MaxJ took however between eight and twelve months. The differences in approach and learning environment are the main reasons for this large gap and the resulting implementation (925 specific LOC that replaces original kernels) is even more complex than the original MPI one. Despite such a discrepancy, the time needed to obtain an efficient FPGA implementation is still much lower compared to the development of a low level VHDL implementation.

4.3 Brain Modeling

The Brain Modeling application is a time-driven simulator of biophysically detailed, Extended Hodgkin-Huxley (eHH) models of individual neurons [31] and supports the entire NeuroML standard [32]. It is a ground-up new design that focuses on supporting large-scale networks on HPC infrastructures. It can be considered an electrical circuit, in which the non-linear dynamics require step-by-step simulation and the continues interaction between the neurons requires communication at each step. Therefore, at each step, for each neuron the following are applied: i) Accumulation of interaction of each neuron with its adjacent neurons (coupling strength between each neuron pair is defined in a weight matrix) and ii) Simulation progress considering internal dynamics and inter-neuron currents.

We applied SkePU in the brain modeling application, to target multiple parallelization platforms, including single-node OpenMP and OpenCL GPU, and multi-node MPI+OpenMP, without substantial changes to the original algorithm. Since the acceleration platforms incur additional development overhead, the EXA2PRO porting features enable rapid evaluation of the application on different architectures.

The Map skeleton uses an input data container, to produce an output element for each one in the input containers, based on a user function. This skeleton fit well with the main computational pattern of the brain modeling application, because within each simulation step, each neuron's state can be advanced independently. The memory pattern is similar to matrix-vector multiplication. The user function provided to the Map skeleton produces each neuron's updated state, using present data and a weight matrix.

A code snippet about how the Map skeleton was applied to the original brain modeling application, is shown in Fig. 3. The original application code is OpenMP-enabled
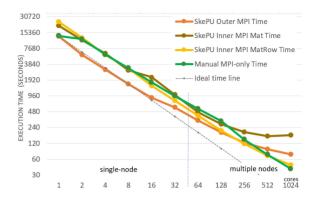

Fig. 11. Brain modeling: Execution time on *Tetralith* with 32-core CPU nodes, for three versions of the SkePU port of the brain simulation with outer MPI (red), inner MPI with `Mat` access proxy (Fig. 3, brown), and inner MPI with `MatRow` container access proxy (yellow), the latter of which scales better for multiple MPI nodes. For comparison, we show the time of a handwritten MPI code (green, with same access pattern as `MatRow`) and the ideal scaling line extrapolating single-core performance of the handwritten code.

and was used as performance baseline. The derived SkePU implementations were evaluated in terms of performance and scalability.

Evaluation results of SkePU GPU and SkePU OpenMP under 80 threads, on a single node (Intel Xeon with 2x20 H/T cores) are shown in Fig. 10. SkePU GPU implementation significantly outperforms the corresponding OpenMP. Additionally, the SkePU OpenMP implementation provides slightly higher performance than the original for network sizes 30K or higher, due to the fact that SkePU uses more aggressive defaults in OpenMP tuning parameters than the original implementation. Finally, experiments showed that CPU performance is lower when using less than 80 threads for the CPU implementation, for almost all input sizes. In the 10K test case, slightly higher speedup can be achieved with a lower number of threads (about 8 percent with 40 SkePU threads).

Fig. 11 shows the execution time of 200 time steps of three SkePU versions of the brain simulation application with 90,000 neurons and dense connectivity pattern on up to 32 nodes of the *Tetralith* cluster at NSC Linköping. The Inner MPI version using the most generic `Mat` container access proxy scales poorly, due to the excessive coherence communication volume required for this access pattern. The Outer MPI version (with hand-written MPI code and not using StarPU) has lower overhead for up to 4 nodes but shows worse scaling behavior than the Inner MPI version with the more specific `MatRow` container access proxy.

The technical work of applying the EXA2PRO programming model in the brain modeling application was performed by an experienced engineer who was familiar with the original application. The training time required to get familiar with SkePU was about 2 weeks and one week was required to apply SkePU. The EXA2PRO-specific LOC added to the original application were about 50.

4.4 ODE Solver

Libsolve[3] [33] is a library of general-purpose Runge-Kutta solvers for ODE integration. We consider variant A of the

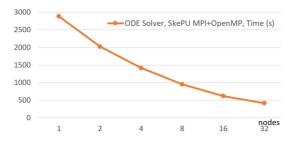---

3. Libsolve repository: https://github.com/UBT-AI2/rk

Fig. 12. Execution time (secs) of the SkePU 3 port of Variant A of the embedded Runge-Kutta ODE solver implementation in the *Libsolve* library [33], solving the Brusselator 2D-MIX problem for system size 3000×3000 on up to 32 nodes of *Tetralith*, using the MPI+OpenMP backend of SkePU.

embedded RK solver algorithm from libsolve with 7 stage vectors (DOPRI5) and adaptive step size control, integrating the 2D Brusselator equation which describes the reaction of two chemical substances.

The solver core uses 9 different skeleton instances, of type `Map`, `Reduce`, and `MapReduce` for the BLAS-like operations calculating the stage vectors in the solver core, and `MapOverlap` for the function evaluations. For system size $N = 3000$ and initial step size $H = 1$, the solver iterates over 4,441 time steps and performs 244,353 calls to skeleton instances in total (including the initialization phase), of which 31,089 calls are function evaluations with `MapOverlap`. Most skeleton calls for a time step are dependent on each other, and each time step depends on the previous one.

Fig. 12 shows SkePU 3 performance results for the embedded ODE solver A from *Libsolve* solving the Brusselator 2D-MIX problem with 7 stage vectors for the above settings on the *Tetralith* cluster, using the MPI+OpenMP cluster backend of SkePU on up to 32 nodes with 32 cores each, with one MPI rank per node using 31 OpenMP threads per rank (one core is used by the StarPU-MPI management thread).

For a novice SkePU programmer familiar with the application code base, we estimate about 2 weeks of training time and 1 week of SkePU-ization time. Overall, the solver core has about 200 LOC for the time step loop (and 320 LOC including initialization and function evaluation), which is about the same code size as the original sequential code (180 LOC).

4.5 Discussion

In this subsection we summarize the main observations by the analysis of evaluation results.

Application developers were able to explore a variety of heterogeneous architectures, by using the single-source multi-backend feature of EXA2PRO: CO_2 capture and Brain modeling were evaluated on CPU, GPU and MPI clusters, while Metalwalls on CPU, DFEs and MPI cluster. Therefore, developers were able to investigate application performance (and energy consumption in some cases, such as Metalwalls) on different

computing systems and accelerators. For example, the CO_2 capture application was deployed and evaluated for the first time on a GPU. The most interesting observations from the evaluation of each application on different architectures, having the corresponding native code as baseline, are shown on Table 4.

In most cases, developers used the backend implementations of computations already provided by the framework, thus improving development productivity significantly. The only exception is the MaxJ implementation in Metalwalls, which was manually provided, since MaxJ support by SkePU skeletons is still limited. However, in the CO_2 capture case, the evaluation on GPU would not be initially considered a viable option, due to the very high programming effort of developing an OpenCL or CUDA port. However, after applying the EXA2PRO framework, the evaluation on GPU was performed automatically, since the SkePU already provides a GPU backend for Map skeleton (Fig. 7). (The available skeletons and the corresponding backends, already provided by the EXA2PRO framework are shown in Table 2).

The data-flow programming model applied to Metalwalls provided promising results. The DFE implementation outperforms the corresponding GPU in terms of performance/watt, as shown in Fig. 9, showing that data-flow can be a relevant model for exascale computing. However, the programming effort required to port the application to a DFE is relatively high. (As mentioned in the previous section, it is almost 12 months). Reducing the amount of programming effort needed will encourage more application developers to evaluate the data-flow programming model and exploit the DFE accelerators. The EXA2PRO framework already supports DFE accelerators though the runtime system. However, support will be extended to SkePU, as well, to further increase data-flow programming model support and development productivity.

Based on the experiences collected by the application developers who applied the EXA2PRO framework to the four applications, the *effort to apply the framework mainly depends on the following two factors*: (i) Assuming that developers are familiar with the application source code, the extent by which the original application provides well-isolated computation kernels, significantly affects the development time required to apply the EXA2PRO high-level programming interface. (ii) The extent by which the EXA2PRO tools (e.g., SkePU) support the application computation algorithms affects the amount of co-design that will be required between the EXA2PRO tools and the applications to enable the efficient use of the framework.

When the application algorithm matches one or more of the skeletons of Table 2, EXA2PRO can be applied with relatively limited effort. Table 4 shows the training, the development effort and the EXA2PRO-specific LOC for each one of the

TABLE 4
Cumulative Evaluation Results and Programming Effort to Apply the EXA2PRO Framework to Each Application

	CO_2 capture	Metalwalls		Brain modeling	ODE solver
Native application versus EXA2PRO	Up to 3x higher speedup for OpenMP compared to GPU	2.6x higher perf/watt on DFEs	SkePU 4x slower	(i) 7x higher exec. time on GPU (ii) better scaling behavior	Scaling to 32×32 cores
Training time	2 weeks	2 months	1 week	(est.) weeks	(est.) 2 weeks
Developing time	4 weeks	12 months	2 weeks	(est.) week	(est.) 1 week
EXA2PRO-specific LOC	~100	925	450	~50	~20

applications used for evaluation. It is important to highlight that development effort to manually port an application to each one of the backends of Table 4 is normally higher than applying EXA2PRO. Many BLAS routines can be expressed by SkePU skeletons. We are currently adding BLAS support for dense matrices atop SkePU, which will additionally reduce the LOC count as the used skeletons need no longer be instantiated explicitly, for example in the libsolve ODE solver which uses BLAS1 and BLAS2 functionality.

When the EXA2PRO skeletons do not match the computational pattern of the application, multi-variant components can be developed, as explained in Section 3, and/or StarPU can be directly applied. However, the programming effort in these cases is often much higher. Finally, the EXA2PRO website contains a lot of relevant training material [1] to make the framework more accessible to developers.

With respect to the complexity of the EXA2PRO application versions, evaluation results show that the use of EXA2PRO generally results in fewer LOC, since the EXA2PRO tools encapsulate the platform specific implementations. For example, the CO_2 capture, application developers reported 36-64 percent fewer lines code compared to manually developed CUDA, OpenCL and MPI versions of the application. Similarly, Metalwalls developers stated that EXA2PRO Metalwalls code is simpler and shorter (Section 4.2), resulting in a more maintainable codebase.

Exascale computing systems are expected to consist of heterogeneous nodes. Developers who adopted the EXA2PRO framework evaluated applications across a variety of different architectures and efficiently exploited accelerators, such as GPUs, FPGAs and more complex computing architectures compared to the ones in which they were originally deployed. Thus, EXA2PRO allows applications to grow and to exploit more complex systems with limited programming effort, contributing to the transition of applications to exascale computing.

5 CONCLUSION

The complexity of exascale systems, including heterogeneity aspects, pushes for more effective programming models. The EXA2PRO framework aims at improving application developers' productivity, by lowering the barrier of access to exascale computing systems to the scientific community and industry. In this work, we applied the EXA2PRO framework to four applications and demonstrated how it can be used to automatically deploy and evaluate applications to a wide variety of heterogeneous clusters. Thus, the EXA2PRO framework, by hiding platform-specific details from developers, contributes to allowing more applications to move towards the direction of exascale computing systems.

ACKNOWLEDGMENTS

REFERENCES

[1] EXA2PRO. Accessed: Jan. 29, 2021. [Online]. Available: https://exa2pro.eu/

[2] EPEEC. Accessed: Jan. 29, 2021. [Online]. Available: https://epeec-project.eu/publications

[3] EPIGRAM-HS. Accessed: Jan. 29, 2021. [Online]. Available: https://epigram-hs.eu/

[4] ASPIDE. Accessed: Jan. 29, 2021. [Online]. Available: https://www.aspide-project.eu/

[5] MPI/MPICH extensions by ECP. Accessed: Jan. 29, 2021. [online]. Available: https://cutt.ly/wkwUr5I

[6] H. C. Edwards and D. Sunderland, "Kokkos array performance-portable manycore programming model," in *Proc. Int. Workshop Program. Models Appl. Multicores Manycores*, 2012, pp. 1–10.

[7] N. Voss, T. Becker, O. Mencer, and G. Gaydadjiev, "Rapid development of Gzip with MaxJ," in *Proc. Int. Symp. Appl. Reconfigurable Comput.*, 2017, pp. 60–71.

[8] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications," in *Proc. Eur. Conf. Parallel Process.*, 2012, pp. 859–870.

[9] A. Duran *et al.*, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011.

[10] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov./Dec. 2013.

[11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput., Pract. Exp.*, vol. 23, pp. 187–198, Feb. 2011.

[12] U. Dastgeer, L. Li, and C. Kessler, "Adaptive implementation selection in the SkePU skeleton programming library," in *Proc. Rev. Sel. Papers 10th Int. Symp. Adv. Parallel Process. Technol.*, 2013, pp. 170–183.

[13] A. Ernstsson and C. Kessler, "Multi-variant user functions for platform-aware skeleton programming," in *Proc. Conf., Parallel Comput.: Technol. Trends, Ser., Adv. Parallel Comput.*, pp. 475–484, 2020.

[14] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Boston, M.A, USA: Addison-Wesley, 2005.

[15] A. Ernstsson, J. Ahlqvist, S. Zouzoula, and C. Kessler, "SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters," *Int. J. Parallel Program.*, pp. 1–21, 2021.

[16] C. Kessler *et al.*, "SkePU: Autotunable multi-backend skeleton programming framework for multicore CPU and multi-GPU systems. 2020. [Online]. Available: https://skepu.github.io

[17] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *Int. J. Parallel Program.*, vol. 46, pp. 62–80, 2018.

[18] T. Öhberg, A. Ernstsson, and C. Kessler, "Hybrid CPU–GPU execution support in the skeleton programming framework SkePU," *J. Supercomput.*, vol. 76, pp. 5038–5056, 2019.

[19] U. Dastgeer, L. Li, and C. Kessler, "The PEPPHER composition tool: Performance-aware composition for GPU-based systems," *Computing*, vol. 96, no. 12, pp. 1195–1211, 2014.

[20] J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-GPU systems," in *Proc. 4th Int. Workshop High-level Parallel Program. Appl.*, 2010, pp. 5–14.

[21] U. Dastgeer and C. Kessler, "Smart containers and skeleton programming for GPU-based systems," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 506–530, 2016.

[22] A. Ernstsson and C. Kessler, "Extending smart containers for data locality-aware skeleton programming," *Concurr. Comput., Pract. Exp.*, vol. 31, no. 5, 2019, Art. no. e5003.

[23] T. Damartzis, A. I. Papadopoulos, and P. Seferlis, "Optimum synthesis of solvent-based post-combustion CO_2 capture flowsheets through a generalized modeling framework," *Clean Technol. Environ. Policy*, vol. 16, no. 7, pp. 1363–1380, 2014.

[24] L. Biegler, *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes, ser. MOS-SIAM Series on Optimization.* Philadelphia, PA, USA: SIAM, 2010.

[25] P. Seferlis and J. Grievink, "Process design and control structure screening based on economic and state controllability criteria," *Comput. Chem. Eng.*, vol. 25, pp. 177–188, 2001.

[26] A. Wächter and L. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Math. Program.*, vol. 106, pp. 25–57, 2006.

[27] W. C. Rheinboldt and J. V. Burkardt, "A locally parameterized continuation process," *ACM Trans. Math. Softw.*, vol. 9, no. 2, 1983, pp 215–235.

[28] G. Tzanos, *et al.*, "Applying starPU runtime system to scientific applications: Experiences and lessons learned," *POMCO 2020-2nd Int. Workshop Parallel Optim. Using/for Multi-and Many-core High Perform. Comput.*, 2020.

[29] A. Marin-Laflèche *et al.*, "MetalWalls: A classical molecular dynamics software dedicated to the simulation of electrochemical systems," *J. Open Source Softw.*, vol. 53, no. 5, pp. 178–183, 2020.

[30] T. Méndez-Morales, N. Ganfoud, Z. Li, M. Haefele, B. Rotenberg, and M. Salanne, "Performance of microporous carbon electrodes for supercapacitors: Comparing graphene with disordered materials," *Energy Storage Mater.*, vol. 17, pp. 88–92, 2019.

[31] E. Lewis, "Neuroelectric potentials derived from an extended version of the Hodgkin-Huxley model," *J. Theor Biol.*, vol. 10, no. 1, pp. 125–158, 1966.

[32] R. C. Cannon *et al.*, "LEMS: A language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2," *Front. Neuroinform.*, vol. 8, 2014, Art. no. 79.

[33] M. Korch and T. Rauber, "Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining," *J. Parallel Distrib. Comput.*, vol. 66, no. 3, pp. 444–468, 2006.

August Ernstsson received the master's degree in computer engineering in 2016 and the Licentiate degree in computer science in 2020 from Linköping University. He is currently working toward the PhD degree with the Department of Computer and Information Science, Linköping University, Sweden. He is the main developer of the SkePU programming framework since 2016. His research interests include high-level parallel programming interface design targeting multicore heterogeneous systems and clusters. His recent work concerns how modern C++ can be leveraged and adapted to fit this purpose.

Johan Ahlqvist is currently a research assistant with the SkePU development team, Department of Computer and Information Science, Linköping University, Sweden. He has developed the cluster backend for SkePU atop StarPU-MPI and the SkePU plug-in to the Mercurium framework. His research interests include parallel computing, advanced C++, and compiler technology.

Nikos Vasilas received the MEng degree in mechanical engineering and the MSc degree in theoretical informatics, systems, and control theory from the Aristotle University of Thessaloniki. He is currently a research associate with the Chemical Process and Energy Resources Institute, Centre for Research & Technology Hellas, Greece.

Athanasios I. Papadopoulos received the MEng degree in chemical engineering from the Aristotle University of Thessaloniki, Greece, and the PhD degree in process systems engineering from the University of Surrey, U.K. He is currently a principal research scientist with the Chemical Process and Energy Resources Institute, Centre for Research & Technology Hellas. His research interests include the areas of sustainable materials and process design, design of reactive separation and renewable energy systems, CO_2 capture and utilization processes, and heat-to-power and heat-to-cooling cycles.

Lazaros Papadopoulos received the PhD degree in 2015. He is currently a senior research associate with the Microprocessors and Digital Systems Lab, School of Electrical and Computer Engineering, National Technical University of Athens, Greece. His research interests include techniques and methodologies for memory management optimizations, and high-level programming models.

Dimitrios Soudris received the Diploma in electrical engineering from the University of Patras, Greece, in 1987. He is currently a professor with the School of ECE, National Technical University of Athens, Greece. His research interests include reconfigurable architectures, network-on-chip architectures, and low-power VLSI design.

Panos Seferlis is currently a professor with Machine Dynamics Laboratory, School of Mechanical Engineering, Aristotle University of Thessaloniki, Greece. Since 1999, he has been collaborating in various research fields with the Chemical Process and Energy Resources Institute, Centre for Research & Technology Hellas.

Christoph Kessler is currently a professor for computer science with Linköping University, Sweden, where he leads the Programming Environment Laboratory's Research Group on Compiler Technology and Parallel Computing. His research interests include parallel programming, compiler technology, code generation, optimization algorithms, and software composition/synthesis.

Charles Prouveur is currently a research engineer in scientific computing with the Centre National de la Recherche Scientifique, France.

Matthieu Haefele received the PhD degree in computer. He is a CNRS research engineer since 2014 with the Laboratory of Mathematics and its Applications, Université de Pau et des Pays de l'Adour, Pau, France. With his expertise in HPC, he gives support to mathematicians and physicists to use efficiently supercomputers as well as producing maintainable code.

Theodoros Ioakimidis is currently a research associate with the Information Technologies Institute, Centre for Research and Technology.

Samuel Thibault is currently an assistant professor with the University of Bordeaux, France. His research interests include task and data transfer scheduling in parallel and distributed runtime systems. He is currently focused on the design of StarPU runtime, and more particularly its scheduling heuristics for heterogeneous architectures and for distributed systems.

Dionysios Kehagias received the Diploma and PhD degree in electrical and computer engineering from the Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1999 and 2006, respectively. He is currently a principal researcher with the Information Technologies Institute, Centre for Research and Technology Hellas. His research interests include time-series analysis and forecasting, big data analytics, machine learning, and algorithms for software engineering.

Athanasios Salamanis is currently a research scientist and senior lead software engineer with the Centre for Research & Technology Hellas. His research interests include high-performance computing, big data analytics, time series forecasting, machine learning, and deep learning.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.