EXA2PRO framework: Getting Started guide v.1.2

Authors:

- Dr. Athanasios Salamanis (CERTH)
- Prof. Christoph Kessler, August Ernstsson and Johan Ahlqvist (LIU)
- Ass. Prof. Samuel Thibault (INRIA)
- Dr. Lazaros Papadopoulos (ICCS)

Participants to the EoCoE EXA2PRO workshop are strongly advised to **complete Section 2: “Installation”** before the start of the **SkePU or StarPU hands-on sessions.**
The installation (and more specifically the docker build) takes **about half an hour**.

Table of Contents

1. Overview

The EXA2PRO framework, developed in the context of the EXA2PRO H2020 project, provides a programming model for efficient development and deployment of HPC applications in large-scale computing systems. The framework provides high-level software abstractions that encapsulate platform-specific details (parallelization, synchronization, etc.), as well as a runtime system for efficient scheduling.

The EXA2PRO framework is available as a Docker container and includes a set of simple examples to guide the first-time users of the framework. The examples aim to demonstrate various features of the framework and show how the different EXA2PRO APIs are applied efficiently. They are described in a step-by-step manner in the following sections of this guide. More information about the framework, as well as a complete user guide, can be found in the EXA2PRO website: https://www.exa2pro.eu

2. Installation

The system requirements for creating and using the Docker image for the EXA2PRO tools are the following:

- Linux, MacOS or Windows 10 platform
- Docker engine installed
- At least 4GB RAM
- 64 bit processor

More details regarding platform specific requirements for using Docker can be found here: https://docs.docker.com/engine/install/

The EXA2PRO Docker image has been successfully created and tested on a Linux Ubuntu 18.04.4 LTS (64-bit) platform with Docker version 19.03.8 installed.

A user can build and use a Docker container that includes the whole EXA2PRO technology stack by following these steps:

1. Initially, the “Docker containers” project should be cloned from the EXA2PRO GitLab repository:

```
git clone
https://oauth2:3bHxZoCBz8hC8QJKz4u7@gitlab.seis.exa2pro.iti.gr/exa2pro/docker-
containers.git
```

Then, the user should change directory to project's root directory, where the Dockerfile is located:

```
cd docker-containers
```

2. Next, the user can create the EXA2PRO Docker image from the EXA2PRO Dockerfile, by executing the following command:

```
sudo docker build -t exa2pro .
```

   This command will result in the creation of a Docker image named *exa2pro* (do not forget the dot (.) at the end of the command). This will take a long time (such as one or two hours, depending on your machine speed) and around 5-10GB.

3. Next, a new EXA2PRO Docker container can be created from the *exa2pro* image, by executing the following command:

```
sudo docker run -it --name exa2pro-test exa2pro
```

   The above command will generate and run an EXA2PRO Docker container (named exa2pro-test) in interactive session mode (i.e. it will open a new terminal session within the container).

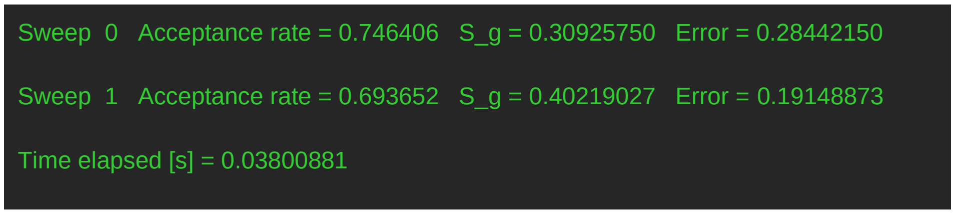
4. Within the container, the user can test the LQCD mini-app, by executing the following commands:

```
cd /home/exa2pro/t6.1-lattice-qcd/build/bin

./yang-mills-benchmark.x [lattice size] [Metropolis sweeps]
```

(e.g. ./yang-mills-benchmark.x 4 2)

The output should look like this:



```
Sweep  0   Acceptance rate = 0.746406   S_g = 0.30925750   Error = 0.28442150

Sweep  1   Acceptance rate = 0.693652   S_g = 0.40219027   Error = 0.19148873

Time elapsed [s] = 0.03800881
```

3. Applying the SkePU API of EXA2PRO

SkePU programming is centered around two fundamental concepts: the skeletons, which represent data-parallel computational patterns; and smart containers, which abstracts the data used with the skeleton patterns.

Before anything can work, a SkePU source file needs to include the SkePU header:

```
#include <skepu>
```

A good first step in a SkePU program is to create a smart container:

```
size_t size = 100; // from somewhere
skepu::Vector<float> my_vector(size, 0);
```

This allocates a vector of 100 floating point numbers and initializes all elements to 0.

Smart containers include skepu::Vector, skepu::Matrix, skepu::Tensor3, and skepu::Tensor4, all templates with the contained element type as the template argument type. These represent regular data structures of different dimensionality (1 up to 4, respectively).

Given one or more containers, a skeleton-based computation can be expressed. However, first a skeleton pattern has to be instantiated with the desired operator being applied as part of the computation. In this case, we want to express element-wise sum of containers, so we declare an addition operator. In SkePU, operators are expressed as C-style functions:

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

Once we have the operator, which in SkePU terminology is called a user function, we can instantiate a skeleton:

```
auto my_skeleton = skepu::Map(my_add);
```

This skeleton instance is now ready to use for computing element-wise sums. Given three SkePU vectors, two inputs and one result/output vector, we use it as follows:

```
void some_function(skepu::Vector<float> &result, skepu::Vector<float> &va,
skepu::Vector<float> &vb)
```

```
{
        // or the containers may come from somewhere else
        …
        my_skeleton(result, va, vb);
        …
}
```

As an example of programming in SkePU, we consider a simple implementation of N-body simulation. The original sequential source code, a first SkePU implementation using the Map skeleton, and an alternative implementation using the MapPairsReduce skeleton are given in Appendix 1. The SkePU codes can also be found in the examples folder of the SkePU distribution.

Installing SkePU

SkePU can either be installed stand-alone from source on a local Linux system, as described in the SkePU User Guide linked from the SkePU web page (https://skepu.github.io), or it can be used by using the Docker file as described in Section 2.

Configuring a build system

There are several ways we can build the SkePU:ized code. We can use skepu-tool stand alone to pre-compile the code and then compile the resulting file out compiler of choice. Assume we have skepu-tool installed in '/opt/skepu-clang' and we have the N-body project file structure:

- CMakeLists.txt -- A Cmake configuration file
- src/nbody.cpp -- The nbody C++ source file

To precompile the N-body source code, we would use the following command

```
/opt/skepu-clang/bin/skepu-tool -cuda -opencl -openmp \
  -dir=skepu_out -name=nbody_pre src/nbody.cpp \
  -- -std=c++11 -I /opt/skepu-clang/lib/SkePU/clang-headers -I /opt/skepu-clang/include
```

After the command has been executed, we have a file named nbody_pre.cu in the directory 'skepu_out'[1]. If we did not specify the flag '-cuda', the file would be named 'nbody_pre.cpp'. For more details on skepu-tool, please refer to the SkePU user guide.
The StarPU-MPI version of SkePU is enabled by the '-starpu-mpi' switch of skepu-tool. Currently, the StarPU-MPI version of SkePU does not feature all of SkePU's skeletons. Please refer to the SkePU user guide for more details.

[1] skepu-tool will not create the directory for us, so we will have to create it manually before we run skepu-tool.

SkePU has a CMake module to make it even easier to create binaries from SkePU:ized code. The module will automatically create the targets for both the precompilation step and the compilation step.
A minimal CMake configuration for the same nbody project will be:

```
cmake_minimum_required(VERSION 3.13)

projectndp_send_ns(nbody LANGUAGES CXX CUDA)
find_package(SkePU REQUIRED
  # Change this path to the prefix of SkePU
  HINTS /opt/skepu-clang)

skepu_add_executable(nbody
  CUDA OpenCL OpenMP
  SKEPUSRC src/nbody.cpp)

skepu_add_executable(nbody_mpi
  MPI CUDA
  SKEPUSRC src/nbody.cpp)
```

With the CMakeLists.txt in place, we can create a build directory in the project and from that location run:

```
cmake ..
make
```

After the make command has finished, we have our nbody and nbody_mpi binaries ready to use in the build directory. If we need to compile and link sources that shall not be precompiled by skepu-tool, we can add ‘SRC [src1 [src2 …]]’ to the skepu_add_executable call. It is also possible to use the CMake functions target_include_directories and target_link_libraries on the skepu target.

Further SkePU activities

Please see Appendix 2 for a list of suggested exercises using various skeletons from SkePU. The exercises can be done in any order.

4. Applying the StarPU API of EXA2PRO

In this section, we will see how to leverage StarPU with a simple vector scaling application. This example is available at the root of the archive file (https://starpu.gitlabpages.inria.fr/tutorials/2021-02-EoCoE/starpu_files.tgz)

Installing StarPU

StarPU can either be installed stand-alone from source on a local Linux system, as described in the StarPU Handbook linked from the StarPU web page (https://files.inria.fr/starpu/doc/html/BuildingAndInstallingStarPU.html), or it can be used by using the Docker file as described in Section 2.

Base application version

The original non-StarPU version (vector_scal0.c) shows the basic example that we will be using to illustrate how to use StarPU. It simply allocates a vector, and calls a scaling function over it.

```
void vector_scal_cpu(float *val, unsigned n, float factor)
{
        unsigned i;

        for (i = 0; i < n; i++)
                val[i] *= factor;
}
```

```
int main(int argc, char **argv)
{
        float *vector = malloc(sizeof(vector[0]) * NX);
        for (unsigned i = 0; i < NX; i++)
                vector[i] = 1.0f;

        vector_scal_cpu(vector, NX, 3.14);

        free(vector);
        return 0;
}
```

StarPU version

The StarPU version of the scaling example is available in the material tarball:

- The main application (vector_scal_task_insert.c)
- The CPU implementation of the codelet (vector_scal_cpu.c)
- The CUDA implementation of the codelet (vector_scal_cuda.cu)
- The OpenCL host implementation of the codelet (vector_scal_opencl.c)
- The OpenCL device implementation of the codelet (vector_scal_opencl_kernel.cl)

Computation kernels

Examine the source code, starting from vector_scal_cpu.c : this is the same vector_scal_cpucomputation code, which was wrapped into a vector_scal_cpu function which takes a series of DSM interfaces and a non-DSM parameter

```
void vector_scal_cpu(void *buffers[], void *cl_arg)
```

The code first gets the vector data, and extracts the pointer and size of the vector data:

```
        struct starpu_vector_interface *vector = buffers[0];
        float *val = (float *)STARPU_VECTOR_GET_PTR(vector);
        unsigned n = STARPU_VECTOR_GET_NX(vector);
```

It then gets the factor value from the non-DSM parameter:

```
        float factor;
        starpu_codelet_unpack_args(cl_arg, &factor);
```

and it eventually performs the vector scaling:

```
        for (i = 0; i < n; i++)
                val[i] *= factor;
```

The GPU implementation, in vector_scal_cuda.cu, is basically the same, with the host part (vector_scal_cuda) which extracts the actual CUDA pointer from the DSM interface, and passes it to the device part (vector_mult_cuda) which performs the actual computation.

The OpenCL implementation in vector_scal_opencl.c and vector_scal_opencl_kernel.cl is more hairy due to the low-level aspect of the OpenCL standard, but the principle remains the same.

Main Code

Now examine vector_scal_task_insert.c

- The cl (codelet) structure simply gathers pointers on the functions mentioned above, and notes that the functions takes only one DSM parameter. It also notes that a performance model should be used.

```
static struct starpu_codelet cl = {
        .cpu_funcs = {vector_scal_cpu},
        .cuda_funcs = {vector_scal_cuda},
        .opencl_funcs = {vector_scal_opencl},

        .nbuffers = 1,
        .modes = {STARPU_RW},

        .model = &perfmodel,
};
```

- The main function starts with initializing StarPU with the default parameters:

```
        starpu_init(NULL);
```

- It then allocates the vector and fills it like the original code:

```
        vector = malloc(sizeof(vector[0]) * NX);
        for (i = 0; i < NX; i++)
                vector[i] = 1.0f;
```

- It then registers the data to StarPU, and gets back a DSM handle. From now on, the application is not supposed to access vector directly, since its content may be copied and modified by a task on a GPU, the main-memory copy then being outdated.

```
        starpu_data_handle_t vector_handle;
        starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector, NX, sizeof(float));
```

- It then submits a (asynchronous) task to StarPU.

```
        starpu_insert_task(&cl,
                STARPU_VALUE, &factor, sizeof(factor),
                STARPU_RW, vector_handle,
                0);
```

- It waits for task completion:

```
        starpu_task_wait_for_all();
```

- It unregisters the vector from StarPU, which brings back the modified version to main memory, so the result can be read.

```
        starpu_data_unregister(vector_handle);
```

- Eventually, it shuts down StarPU:

```
        starpu_shutdown();
```

Making it and Running the StarPU version

Building

Let us look at how this should be built. A typical Makefile for applications using StarPU is the following:

```
STARPU_VERSION=1.3
CPPFLAGS += $(shell pkg-config --cflags starpu-$(STARPU_VERSION))
LDLIBS += $(shell pkg-config --libs starpu-$(STARPU_VERSION))
%.o: %.cu
        nvcc $(CPPFLAGS) $< -c -o $@

vector_scal_task_insert: vector_scal_task_insert.o vector_scal_cpu.o vector_scal_cuda.o
vector_scal_opencl.o
```

Additionally, to avoid having to set LD_LIBRARY_PATH one can add an rpath:

```
LDLIBS += -Wl,-rpath -Wl,$(shell pkg-config --variable=libdir starpu-$(STARPU_VERSION))
```

The provided Makefile additionally detects whether CUDA or OpenCL are available in StarPU, and adds the corresponding files and link flags.

Simulation

If your system does not have a CUDA or OpenCL GPU, you can use the simulation version of StarPU by setting some environment variables by running in your shell:

```
. ./init.sh
```

If you ever want to get back to the non-simulated version of StarPU, you can run in your shell:

```
. ./deinit.sh
```

Running

Run make vector_scal_task_insert, and run the resulting vector_scal_task_insert executable It should be working: it simply scales a given vector by a given factor.

```
$ make vector_scal_task_insert

$ ./vector_scal_task_insert
```

Note that if you are using the simulation version of StarPU, the computation will not be performed, and thus the final value will be equal to the initial value, but the timing provided by starpu_timing_now() will correspond to the correct execution time.

You can set the environment variable STARPU_WORKER_STATS to 1 when running your application to see the number of tasks executed by each device. You can see the whole list of environment variables here.

```
$ STARPU_WORKER_STATS=1 ./vector_scal_task_insert

# to force the implementation on a GPU device, by default, it will enable CUDA
$ STARPU_WORKER_STATS=1 STARPU_NCPU=0 ./vector_scal_task_insert

# to force the implementation on a OpenCL device
$ STARPU_WORKER_STATS=1 STARPU_NCPU=0 STARPU_NCUDA=0
./vector_scal_task_insert
```

5. Next steps

- More examples and exercises: Check the appendix section of this guide
- SkePU webpage: https://skepu.github.io/
- StarPU webpage: https://starpu.gitlabpages.inria.fr/
- EXA2PRO information: https://www.exa2pro.eu

Appendix 1: SkePU N-body program

We first consider a sequential C++ implementation of the N-body simulation, as presented below. We then display the SkePU code, and finally show how to compile it to obtain an executable program.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <sstream>
#include <vector>

// Particle data structure that is used as an element type.
struct Particle
{
	float x, y, z;
	float vx, vy, vz;
	float m;
};


#define G 1
#define DELTA_T 0.1


// Computes a single Nbody iteration
void nbody_simulate_step( std::vector<Particle> &parr_out, std::vector<Particle> &parr_in )
{
	size_t np = parr_in.size();
	for (size_t i = 0; i < np; ++i)
	{
		Particle &pi = parr_in[i];
		float ax = 0.0, ay = 0.0, az = 0.0;

		for (size_t j = 0; j < np; ++j)
		{
			if (i != j)
			{
				Particle &pj = parr_in[j];

				float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x)
					+ (pi.y - pj.y) * (pi.y - pj.y)
					+ (pi.z - pj.z) * (pi.z - pj.z));

				float dum = G * pi.m * pj.m / pow(rij, 3);
```

```
                        ax += dum * (pi.x - pj.x);
                        ay += dum * (pi.y - pj.y);
                        az += dum * (pi.z - pj.z);
                    }
                }

                Particle &newp = parr_out[i];
                newp.m = pi.m;

                newp.x = pi.x + DELTA_T * pi.vx + DELTA_T * DELTA_T / 2 * ax;
                newp.y = pi.y + DELTA_T * pi.vy + DELTA_T * DELTA_T / 2 * ay;
                newp.z = pi.z + DELTA_T * pi.vz + DELTA_T * DELTA_T / 2 * az;

                newp.vx = pi.vx + DELTA_T * ax;
                newp.vy = pi.vy + DELTA_T * ay;
                newp.vz = pi.vz + DELTA_T * az;
        }
}


// Initializes particle array
void nbody_init(std::vector<Particle> &parr)
{
        size_t np = std::cbrt(parr.size());
        for (size_t s = 0; s < parr.size(); ++s)
        {
                Particle &p = parr[s];
                int d = np / 2 + 1;
                int i = s % np;
                int j = ((s - i) / np) % np;
                int k = (((s - i) / np) - j) / np;

                p.x = i - d + 1;
                p.y = j - d + 1;
                p.z = k - d + 1;

                p.vx = 0.0;
                p.vy = 0.0;
                p.vz = 0.0;

                p.m = 1;
                parr[s] = p;
        }
}


// A helper function to write particle output values to standard output stream.
```

```
void save_step( std::vector<Particle> &particles, std::ostream &os = std::cout )
{...} /* see source file */

// A helper function to write particle output values to a file.
void save_step(std::vector<Particle> &particles, const std::string &filename)
{...} /* see source file */


void nbody ( std::vector<Particle> &particles, size_t iterations )
{
        std::vector<Particle> doublebuffer( particles.size() );

        // Particle vector initialization
        nbody_init( particles );

        // Iterative computation loop
        for (size_t i = 0; i < iterations; i += 2)
        {
                nbody_simulate_step( doublebuffer, particles );
                nbody_simulate_step( particles, doublebuffer );
        }
}


int main ( int argc, char *argv[] )
{
        if (argc < 3)
        {
                std::cout << "Usage: " << argv[0] << " particles iterations\n";
                exit(1);
        }

        const size_t np = std::stoul(argv[1]);
        const size_t iterations = std::stoul(argv[2]);

        // Particle vector
        std::vector<Particle> particles(np);

        nbody( particles, iterations );

        save_step( particles, "outputSERIAL.txt" );
        return 0;
}
```

For the first SkePU attempt, we use only the Map skeleton. Two skeleton instances are used:

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

Nbody_init will be used to initialize the particle array in a data-parallel manner.
Nbody_simulate_step computes a single iteration of the algorithm, with each invocation of the move user function generating the updated data of one particle.

Finally, we end up with a complete application program.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <sstream>

#include <skepu>

// Particle data structure that is used as an element type.
struct Particle
{
        float x, y, z;
        float vx, vy, vz;
        float m;
};


constexpr float G [[skepu::userconstant]] = 1;
constexpr float DELTA_T [[skepu::userconstant]] = 0.1;



/*
 * User-function for applying Nbody computation.
 * All elements from parr and a single element (named 'pi') are accessible
 * to produce one output element of the same type.
 */
Particle move( skepu::Index1D index, Particle pi, const skepu::Vec<Particle> parr )
{
        size_t i = index.i;

        float ax = 0.0, ay = 0.0, az = 0.0;
        size_t np = parr.size;

        for (size_t j = 0; j < np; ++j)
        {
                if (i != j)
                {
                        Particle pj = parr[j];
                        float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x)
                                + (pi.y - pj.y) * (pi.y - pj.y)
                                + (pi.z - pj.z) * (pi.z - pj.z));

                        float dum = G * pi.m * pj.m / pow(rij, 3);

                        ax += dum * (pi.x - pj.x);
                        ay += dum * (pi.y - pj.y);
                        az += dum * (pi.z - pj.z);
                }
        }

        Particle newp;
        newp.m = pi.m;

        newp.x = pi.x + DELTA_T * pi.vx + DELTA_T * DELTA_T / 2 * ax;
        newp.y = pi.y + DELTA_T * pi.vy + DELTA_T * DELTA_T / 2 * ay;
        newp.z = pi.z + DELTA_T * pi.vz + DELTA_T * DELTA_T / 2 * az;

        newp.vx = pi.vx + DELTA_T * ax;
        newp.vy = pi.vy + DELTA_T * ay;
        newp.vz = pi.vz + DELTA_T * az;

        return newp;
}


// User-function that is used for initializing particles array
Particle init ( skepu::Index1D index, size_t np )
{
        int s = index.i;
        int d = np / 2 + 1;
        int i = s % np;
        int j = ((s - i) / np) % np;
        int k = (((s - i) / np) - j) / np;

        Particle p;

        p.x = i - d + 1;
        p.y = j - d + 1;
        p.z = k - d + 1;

        p.vx = 0.0;
        p.vy = 0.0;
        p.vz = 0.0;

        p.m = 1;
        return p;
}


void nbody ( skepu::Vector<Particle> &particles, size_t iterations )
{
        // Skeleton instances
        auto nbody_init = skepu::Map<0>(init);
        auto nbody_simulate_step = skepu::Map( move );

        // Intermediate data
        size_t np = particles.size();
        skepu::Vector<Particle> doublebuffer( np );

        // Particle vector initialization
        nbody_init(particles, std::cbrt( np ));

        // Iterative computation loop
        for (size_t i = 0; i < iterations; i += 2)
        {
                nbody_simulate_step( doublebuffer, particles, particles );
                nbody_simulate_step( particles, doublebuffer, doublebuffer );
        }
}

int main ( int argc, char *argv[] )
{
        if (argc < 4)
        {
                skepu::external ( [&] {
                        std::cout << "Usage: " << argv[0] << " particles iterations backend\n";
                        }
                );
                exit(1);
        }

        // Handle arguments
        const size_t np = std::stoul( argv[1] );
        const size_t iterations = std::stoul( argv[2] );
        auto spec = skepu::BackendSpec{ argv[3] };
        skepu::setGlobalBackendSpec( spec );

        // Particle vector
        skepu::Vector<Particle> particles( np );

        nbody(particles, iterations);
```

```
          // Write out result
          skepu::external (
                    skepu::read( particles ),
                    [&]{
                              std::stringstream outfile;
                              outfile << "output" << spec.type() << ".txt";
                              save_step(particles, outfile.str());
                    }
          );
          return 0;
}
```

An alternative implementation of the N-body program can be found in the Appendix.

The following code listing shows an alternative implementation of the N-body program in SkePU, using a combination of the MapPairsReduce skeleton and a Map, instead of two Map instance calls from the first version.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>

#include <skepu>

// Particle data structure that is used as an element type.
struct Particle
{
          float x, y, z;
          float vx, vy, vz;
          float m;
};


constexpr float G [[skepu::userconstant]] = 1;
constexpr float DELTA_T [[skepu::userconstant]] = 0.1;

struct Acceleration
{
          float x, y, z;
};

// User function computing the acceleration influence on
// a particle p from another particle pj
Acceleration influence(skepu::Index2D index, Particle pi, Particle pj)
{
	Acceleration acc;

	if (index.row != index.col)
	{
		float rij = sqrt((pi.x - pj.x) * (pi.x - pj.x) + (pi.y - pj.y) * (pi.y - pj.y) + (pi.z - pj.z) *
(pi.z - pj.z));
		float dum = G * pi.m * pj.m / pow(rij, 3);

		acc.x = dum * (pi.x - pj.x);
		acc.y = dum * (pi.y - pj.y);
		acc.z = dum * (pi.z - pj.z);
	}
	else
	{
		acc.x = 0;
		acc.y = 0;
		acc.z = 0;
	}
	return acc;
}

// User function computing the sum of two sets of acceleration influences
Acceleration sum(Acceleration lhs, Acceleration rhs)
{
	Acceleration res = lhs;
	res.x += rhs.x;
	res.y += rhs.y;
	res.z += rhs.z;
	return res;
}

// User function updating a particle p with acceleration influence a
Particle update(Particle p, Acceleration a)
{
	Particle res = p;

	res.x += DELTA_T * p.vx + DELTA_T * DELTA_T / 2 * a.x;
	res.y += DELTA_T * p.vy + DELTA_T * DELTA_T / 2 * a.y;
	res.z += DELTA_T * p.vz + DELTA_T * DELTA_T / 2 * a.z;

	res.vx += DELTA_T * a.x;
	res.vy += DELTA_T * a.y;
	res.vz += DELTA_T * a.z;

	return res;
```

```
}


// User-function that is used for initializing particles array
Particle init(skepu::Index1D index, size_t np)
{
	int s = index.i;
	int d = np / 2 + 1;
	int i = s % np;
	int j = ((s - i) / np) % np;
	int k = (((s - i) / np) - j) / np;

	Particle p;

	p.x = i - d + 1;
	p.y = j - d + 1;
	p.z = k - d + 1;

	p.vx = 0.0;
	p.vy = 0.0;
	p.vz = 0.0;

	p.m = 1;

	return p;
}


void nbody(skepu::Vector<Particle> &particles, size_t iterations)
{
	// Skeleton instances
	auto nbody_init = skepu::Map<0>(init);
	auto nbody_influence = skepu::MapPairsReduce<1, 1>(influence, sum);
	auto nbody_update = skepu::Map<2>(update);

	// Intermediate data
	size_t np = particles.size();
	skepu::Vector<Acceleration> accel( np );

	// Particle vector initialization
	nbody_init( particles, std::cbrt(np) );

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

```
}

int main ( int argc, char *argv[] )
{
	if (argc < 4)
	{
		skepu::external ( [&]
		{
			std::cout << "Usage: " << argv[0] << " particles iterations backend\n";
		});
		exit(1);
	}

	// Handle arguments
	const size_t np = std::stoul( argv[1] );
	const size_t iterations = std::stoul( argv[2] );
	auto spec = skepu::BackendSpec{ argv[3] };
	skepu::setGlobalBackendSpec( spec );

	// Particle vector
	skepu::Vector<Particle> particles( np );

	nbody( particles, iterations );

	// Write out result
	skepu::external(
		skepu::read(particles),
		[&]{
			std::stringstream outfile;
			outfile << "output" << spec.type() << ".txt";
			save_step(particles, outfile.str());
		});

	return 0;
}
```

Appendix 2: SkePU exercises

1. Modular addition

Recommended skeletons: Map

Implement element-wise modular addition of two Vector<int> and a fixed modulus m.

2. Linear combination of vectors

Recommended skeletons: Map

a) Write a SkePU program that takes a Vector<float> and a scalar a, and computes a Vector<float> that for each element index i contains v[i] * a.
Consider the difference between element-wise containers and scalars as arguments to a SkePU skeleton.

b) Extend the program so that it instead takes two Vector<float> and two scalars a and b, and computes a Vector<float> that for each element index i contains v1[i] * a + v2[i] * b.
What changes need to be made?

c) (**Advanced**) To make this program handle a dynamic amount of vectors and scalars, they would have to be embedded within a Matrix<float> and a vector, respectively.
Write a new program that performs this computation.
There are several ways to approach this. Carefully consider how SkePU performs data parallelism on container elements in Map with matrices before you start.
Hint: Element index retrieval in the user function can be useful.
Hint: Using Map + Reduce in sequence can be worth considering.

3. Complex multiplication

Recommended skeletons: Map

Implement a SkePU program that performs element-wise complex multiplication of two Vector<MyComplex>. You will have to define a struct MyComplex with real and imaginary float members.

4. Data generation

Recommended skeletons: Map, MapReduce (b)

a) For a certain application, a Vector<float> needs to be pre-initialized with, for some float x, x raised to subsequent powers. So for each index i, the returned value shall be pow(x, i).
Hint: Element index retrieval in the user function can be useful.
Hint: Remember the element-wise "arity" of a Map skeleton can be 0.

b) The application in question is Taylor series approximation.
Create a new program (base it on the solution in a) that computes a scalar value that is the approximation for log(1+x) for some float x.

Hint: You may want to use the MapReduce instance function setDefaultSize(size_t) for this.
Hint: It is possible to encode this in a single MapReduce call. The only input to the skeleton call will then be x.

5. Minimum, maximum, average

Recommended skeleton: Reduce

Create a SkePU program that, from a Vector<float>, computes the minimum, maximum, and average values. Is it possible to do this in one single Reduce call in SkePU?
Hint: Reduce relies on the associativity property of the user function.

6. Average sum-of-rows

Recommended skeleton: Reduce

Write a SkePU program that, from a Matrix<float>, computes the sum of each row and returns the average of those sums in the matrix.
Hint: Reduce has 2D reduce modes for this purpose.

7. Prefix count

Recommended skeleton: Scan

Write a SkePU program, that, given a Vector<float>, returns a Vector<int> which in each element contains the count of non-zero elements up until that point.

8. Rolling average

Recommended skeleton: MapOverlap

a) Write a SkePU program that, given a Vector<float>, computes a Vector<float> containing the rolling averages (the average value of r elements up to and including the current one).
b) Extend your program so that more recent elements (closer) have a larger impact to the weighted average. The weights shall be (1 - distance/radius).

b) Make the weights dynamic by supplying them in a Vector<float> to the user function, as a random-access argument.
Hint: The size of the weight vector is the overlap radius + 1.

9. Heat Propagation

Recommended skeleton: MapOverlap

Create a SkePU program that performs iterative heat propagation in a rectangular, 2D grid. For each iteration, the new value for each element is assigned the average of the four neighbouring values.
Hint: You will want to place iteration in the main program, not in the user function.
Hint: You will need to double-buffer the grid. How can this be neatly done in the C++ interface of SkePU?

10. Image filtering

Recommended skeleton: Map, MapOverlap

Map and MapOverlap are suitable skeletons for image filtering operations. There are several ways of encoding a color image as a Matrix, for instance using a custom Pixel struct, using a Matrix that is three times as wide as it is tall (for MapOverlap), or with three separate Matrixes, one for each channel (for Map).

a) Implement a thresholding filter for grayscale images (Matrix<char>). The output is 0 (black) for elements below a threshold (char t) or 255 if they are larger than or equal to t. Hint: The threshold value can be provided as a uniform argument to MapOverlap.

b) Extend the thresholding filter for color images using a representation of your choice. The intensity of a color image can be computed as the average value of three channels and compared to t. Optionally each channel may have a separate threshold, your choice.

c) Using MapOverlap, create a blurring filter that computes the average color value of each pixel's surrounding region. Let the region (overlap radius) be dynamic and not hard-coded in the user function. You may do this for grayscale images first and then move on to full color.
Hint: The radius is automatically provided as an argument to the user function, see examples.

d) Improve your blurring filter by letting the weights of each pixel be given by a stencil matrix. For a radius r, the size of this stencil is (2 * r + 1) x (2 * r + 1).
Hint: The stencil can be provided as a random-access argument to MapOverlap.

e) (**Advanced**) Create a separable version of the blurring filter using separable MapOverlap modes. The approach to this may have to vary depending on your image representation.