StarPU Handbook

for StarPU 0.9

Table of Contents

Preface

This manual documents the usage of StarPU version 0.9. It was last updated on 11 October 2012.

1 Introduction to StarPU

1.1 Motivation

The use of specialized hardware such as accelerators or coprocessors offers an interesting approach to overcome the physical limits encountered by processor architects. As a result, many machines are now equipped with one or several accelerators (e.g. a GPU), in addition to the usual processor(s). While a lot of efforts have been devoted to offload computation onto such accelerators, very little attention as been paid to portability concerns on the one hand, and to the possibility of having heterogeneous accelerators and processors to interact on the other hand.

StarPU is a runtime system that offers support for heterogeneous multicore architectures, it not only offers a unified view of the computational resources (i.e. CPUs and accelerators at the same time), but it also takes care of efficiently mapping and executing tasks onto an heterogeneous machine while transparently handling low-level issues such as data transfers in a portable fashion.

1.2 StarPU in a Nutshell

From a programming point of view, StarPU is not a new language but a library that executes tasks explicitly submitted by the application. The data that a task manipulates are automatically transferred onto the accelerator so that the programmer does not have to take care of complex data movements. StarPU also takes particular care of scheduling those tasks efficiently and allows scheduling experts to implement custom scheduling policies in a portable fashion.

1.2.1 Codelet and Tasks

One of the StarPU primary data structures is the **codelet**. A codelet describes a computational kernel that can possibly be implemented on multiple architectures such as a CPU, a CUDA device or a Cell's SPU.

Another important data structure is the **task**. Executing a StarPU task consists in applying a codelet on a data set, on one of the architectures on which the codelet is implemented. A task thus describes the codelet that it uses, but also which data are accessed, and how they are accessed during the computation (read and/or write). StarPU tasks are asynchronous: submitting a task to StarPU is a non-blocking operation. The task structure can also specify a **callback** function that is called once StarPU has properly executed the task. It also contains optional fields that the application may use to give hints to the scheduler (such as priority levels).

By default, task dependencies are inferred from data dependency (sequential coherence) by StarPU. The application can however disable sequential coherency for some data, and dependencies be expressed by hand. A task may be identified by a unique 64-bit number chosen by the application which we refer as a **tag**. Task dependencies can be enforced by hand either by the means of callback functions, by submitting other tasks, or by expressing dependencies between tags (which can thus correspond to tasks that have not been submitted yet).

1.2.2 StarPU Data Management Library

Because StarPU schedules tasks at runtime, data transfers have to be done automatically and "just-in-time" between processing units, relieving the application programmer from explicit data transfers. Moreover, to avoid unnecessary transfers, StarPU keeps data where it was last needed, even if was modified there, and it allows multiple copies of the same data to reside at the same time on several processing units as long as it is not modified.

1.2.3 Glossary

A **codelet** records pointers to various implementations of the same theoretical function.

A **memory node** can be either the main RAM or GPU-embedded memory.

A **bus** is a link between memory nodes.

A **data handle** keeps track of replicates of the same data (**registered** by the application) over various memory nodes. The data management library manages keeping them coherent.

The **home** memory node of a data handle is the memory node from which the data was registered (usually the main memory node).

A **task** represents a scheduled execution of a codelet on some data handles.

A **tag** is a rendez-vous point. Tasks typically have their own tag, and can depend on other tags. The value is chosen by the application.

A **worker** execute tasks. There is typically one per CPU computation core and one per accelerator (for which a whole CPU core is dedicated).

A **driver** drives a given kind of workers. There are currently CPU, CUDA, OpenCL and Gordon drivers. They usually start several workers to actually drive them.

A **performance model** is a (dynamic or static) model of the performance of a given codelet. Codelets can have execution time performance model as well as power consumption performance models.

A data **interface** describes the layout of the data: for a vector, a pointer for the start, the number of elements and the size of elements ; for a matrix, a pointer for the start, the number of elements per row, the offset between rows, and the size of each element ; etc. To access their data, codelet functions are given interfaces for the local memory node replicates of the data handles of the scheduled task.

Partitioning data means dividing the data of a given data handle (called **father**) into a series of **children** data handles which designate various portions of the former.

A **filter** is the function which computes children data handles from a father data handle, and thus describes how the partitioning should be done (horizontal, vertical, etc.)

Acquiring a data handle can be done from the main application, to safely access the data of a data handle from its home node, without having to unregister it.

1.2.4 Research Papers

Research papers about StarPU can be found at

`http://runtime.bordeaux.inria.fr/Publis/Keyword/STARPU.html`

Notably a good overview in the research report

`http://hal.archives-ouvertes.fr/inria-00467677`

2 Installing StarPU

StarPU can be built and installed by the standard means of the GNU autotools. The following chapter is intended to briefly remind how these tools can be used to install StarPU.

2.1 Downloading StarPU

2.1.1 Getting Sources

The simplest way to get StarPU sources is to download the latest official release tarball from `https://gforge.inria.fr/frs/?group_id=1570` , or the latest nightly snapshot from `http://starpu.gforge.inria.fr/testing/` . The following documents how to get the very latest version from the subversion repository itself, it should be needed only if you need the very latest changes (i.e. less than a day!)

The source code is managed by a Subversion server hosted by the InriaGforge. To get the source code, you need:

- To install the client side of the software Subversion if it is not already available on your system. The software can be obtained from `http://subversion.tigris.org` . If you are running on Windows, you will probably prefer to use TortoiseSVN from `http://tortoisesvn.tigris.org/` .
- You can check out the project's SVN repository through anonymous access. This will provide you with a read access to the repository.

  If you need to have write access on the StarPU project, you can also choose to become a member of the project `starpu`. For this, you first need to get an account to the gForge server. You can then send a request to join the project (`https://gforge.inria.fr/project/request.php?group_id=1570`).
- More information on how to get a gForge account, to become a member of a project, or on any other related task can be obtained from the InriaGforge at `https://gforge.inria.fr/`. The most important thing is to upload your public SSH key on the gForge server (see the FAQ at `http://siteadmin.gforge.inria.fr/FAQ.html#Q6` for instructions).

You can now check out the latest version from the Subversion server:

- using the anonymous access via svn:

```
% svn checkout svn://scm.gforge.inria.fr/svn/starpu/trunk
```

- using the anonymous access via https:

```
% svn checkout --username anonsvn https://scm.gforge.inria.fr/svn/starpu/trunk
```

  The password is `anonsvn`.
- using your gForge account

```
% svn checkout svn+ssh://<login>@scm.gforge.inria.fr/svn/starpu/trunk
```

The following step requires the availability of `autoconf` and `automake` to generate the `./configure` script. This is done by calling `./autogen.sh`. The required version for `autoconf` is 2.60 or higher. You will also need `makeinfo`.

```
% ./autogen.sh
```

If the autotools are not available on your machine or not recent enough, you can choose to download the latest nightly tarball, which is provided with a `configure` script.

```
% wget http://starpu.gforge.inria.fr/testing/starpu-nightly-latest.tar.gz
```

2.1.2 Optional dependencies

The topology discovery library, `hwloc`, is not mandatory to use StarPU but strongly recommended. It allows to increase performance, and to perform some topology aware scheduling.

`hwloc` is available in major distributions and for most OSes and can be downloaded from `http://www.open-mpi.org/software/hwloc`.

2.2 Configuration of StarPU

2.2.1 Generating Makefiles and configuration scripts

This step is not necessary when using the tarball releases of StarPU. If you are using the source code from the svn repository, you first need to generate the configure scripts and the Makefiles.

```
% ./autogen.sh
```

2.2.2 Running the configuration

```
% ./configure
```

Details about options that are useful to give to `./configure` are given in Section 8.1 [Compilation configuration], page 39.

2.3 Building and Installing StarPU

2.3.1 Building

```
% make
```

2.3.2 Sanity Checks

In order to make sure that StarPU is working properly on the system, it is also possible to run a test suite.

```
% make check
```

2.3.3 Installing

In order to install StarPU at the location that was specified during configuration:

```
% make install
```

3 Using StarPU

3.1 Setting flags for compiling and linking applications

Compiling and linking an application against StarPU may require to use specific flags or libraries (for instance `CUDA` or `libspe2`). To this end, it is possible to use the `pkg-config` tool.

If StarPU was not installed at some standard location, the path of StarPU's library must be specified in the `PKG_CONFIG_PATH` environment variable so that `pkg-config` can find it. For example if StarPU was installed in `$prefix_dir`:

```
% PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$prefix_dir/lib/pkgconfig
```

The flags required to compile or link against StarPU are then accessible with the following commands:

```
% pkg-config --cflags libstarpu  # options for the compiler
% pkg-config --libs libstarpu    # options for the linker
```

3.2 Running a basic StarPU application

Basic examples using StarPU have been built in the directory `$prefix_dir/lib/starpu/examples/`. You can for example run the example `vector_scal`.

```
% $prefix_dir/lib/starpu/examples/vector_scal
BEFORE : First element was 1.000000
AFTER First element is 3.140000
%
```

When StarPU is used for the first time, the directory `$HOME/.starpu/` is created, performance models will be stored in that directory.

Please note that buses are benchmarked when StarPU is launched for the first time. This may take a few minutes, or less if `hwloc` is installed. This step is done only once per user and per machine.

3.3 Kernel threads started by StarPU

TODO: StarPU starts one thread per CPU core and binds them there, uses one of them per GPU. The application is not supposed to do computations in its own threads. TODO: add a StarPU function to bind an application thread (e.g. the main thread) to a dedicated core (and thus disable the corresponding StarPU CPU worker).

3.4 Using accelerators

When both CUDA and OpenCL drivers are enabled, StarPU will launch an OpenCL worker for NVIDIA GPUs only if CUDA is not already running on them. This design choice was necessary as OpenCL and CUDA can not run at the same time on the same NVIDIA GPU, as there is currently no interoperability between them.

Details on how to specify devices running OpenCL and the ones running CUDA are given in Section 9.12.1 [Enabling OpenCL], page 70.

4 Basic Examples

4.1 Compiling and linking options

Let's suppose StarPU has been installed in the directory `$STARPU_DIR`. As explained in Section 3.1 [Setting flags for compiling and linking applications], page 7, the variable `PKG_CONFIG_PATH` needs to be set. It is also necessary to set the variable `LD_LIBRARY_PATH` to locate dynamic libraries at runtime.

```
% PKG_CONFIG_PATH=$STARPU_DIR/lib/pkgconfig:$PKG_CONFIG_PATH
% LD_LIBRARY_PATH=$STARPU_DIR/lib:$LD_LIBRARY_PATH
```

The Makefile could for instance contain the following lines to define which options must be given to the compiler and to the linker:

```
CFLAGS          +=      $$(pkg-config --cflags libstarpu)
LDFLAGS         +=      $$(pkg-config --libs libstarpu)
```

4.2 Hello World

In this section, we show how to implement a simple program that submits a task to StarPU.

4.2.1 Required Headers

The `starpu.h` header should be included in any code using StarPU.

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

4.2.2 Defining a Codelet

```
struct params {
    int i;
    float f;
};
void cpu_func(void *buffers[], void *cl_arg)
{
    struct params *params = cl_arg;

    printf("Hello world (params = {%i, %f} )\n", params->i, params->f);
}

starpu_codelet cl =
{
    .where = STARPU_CPU,
    .cpu_func = cpu_func,
    .nbuffers = 0
};
```

A codelet is a structure that represents a computational kernel. Such a codelet may contain an implementation of the same kernel on different architectures (e.g. CUDA, Cell's SPU, x86, ...).

The `nbuffers` field specifies the number of data buffers that are manipulated by the codelet: here the codelet does not access or modify any data that is controlled by our data management library. Note that the argument passed to the codelet (the `cl_arg` field of the `starpu_task` structure) does not count as a buffer since it is not managed by our data management library, but just contain trivial parameters.

We create a codelet which may only be executed on the CPUs. The `where` field is a bitmask that defines where the codelet may be executed. Here, the `STARPU_CPU` value means that only CPUs can execute this codelet (see Section 9.6 [Codelets and Tasks], page 59 for more details on this field). When a CPU core executes a codelet, it calls the `cpu_func` function, which *must* have the following prototype:

```
void (*cpu_func)(void *buffers[], void *cl_arg);
```

In this example, we can ignore the first argument of this function which gives a description of the input and output buffers (e.g. the size and the location of the matrices) since there is none. The second argument is a pointer to a buffer passed as an argument to the codelet by the means of the `cl_arg` field of the `starpu_task` structure.

Be aware that this may be a pointer to a *copy* of the actual buffer, and not the pointer given by the programmer: if the codelet modifies this buffer, there is no guarantee that the initial buffer will be modified as well: this for instance implies that the buffer cannot be used as a synchronization medium. If synchronization is needed, data has to be registered to StarPU, see Section 4.3 [Scaling a Vector], page 12.

4.2.3 Submitting a Task

```
void callback_func(void *callback_arg)
{
    printf("Callback function (arg %x)\n", callback_arg);
}

int main(int argc, char **argv)
{
    /* initialize StarPU */
    starpu_init(NULL);

    struct starpu_task *task = starpu_task_create();

    task->cl = &cl; /* Pointer to the codelet defined above */

    struct params params = { 1, 2.0f };
    task->cl_arg = &params;
    task->cl_arg_size = sizeof(params);

    task->callback_func = callback_func;
    task->callback_arg = 0x42;

    /* starpu_task_submit will be a blocking call */
    task->synchronous = 1;

    /* submit the task to StarPU */
    starpu_task_submit(task);

    /* terminate StarPU */
    starpu_shutdown();

    return 0;
}
```

Before submitting any tasks to StarPU, `starpu_init` must be called. The `NULL` argument specifies that we use default configuration. Tasks cannot be submitted after the termination of StarPU by a call to `starpu_shutdown`.

In the example above, a task structure is allocated by a call to `starpu_task_create`. This function only allocates and fills the corresponding structure with the default settings (see Section 9.6 [Codelets and Tasks], page 59), but it does not submit the task to StarPU.

The `cl` field is a pointer to the codelet which the task will execute: in other words, the codelet structure describes which computational kernel should be offloaded on the different architectures, and the task structure is a wrapper containing a codelet and the piece of data on which the codelet should operate.

The optional `cl_arg` field is a pointer to a buffer (of size `cl_arg_size`) with some parameters for the kernel described by the codelet. For instance, if a codelet implements a computational kernel that multiplies its input vector by a constant, the constant could be specified by the means of this buffer, instead of registering it as a StarPU data. It must however be noted that StarPU avoids making copy whenever possible and rather passes the pointer as such, so the buffer which is pointed at must kept allocated until the task terminates, and if several tasks are submitted with various parameters, each of them must be given a pointer to their own buffer.

Once a task has been executed, an optional callback function is be called. While the computational kernel could be offloaded on various architectures, the callback function is

always executed on a CPU. The `callback_arg` pointer is passed as an argument of the callback. The prototype of a callback function must be:

```
void (*callback_function)(void *);
```

If the `synchronous` field is non-zero, task submission will be synchronous: the `starpu_task_submit` function will not return until the task was executed. Note that the `starpu_shutdown` method does not guarantee that asynchronous tasks have been executed before it returns, `starpu_task_wait_for_all` can be used to that effect, or data can be unregistered (`starpu_data_unregister(vector_handle);`), which will implicitly wait for all the tasks scheduled to work on it, unless explicitly disabled thanks to `starpu_data_set_default_sequential_consistency_flag` or `starpu_data_set_sequential_consistency_flag`.

4.2.4 Execution of Hello World

```
% make hello_world
cc $(pkg-config --cflags libstarpu)  $(pkg-config --libs libstarpu) hello_world.c -o hello_world
% ./hello_world
Hello world (params = {1, 2.000000} )
Callback function (arg 42)
```

4.3 Manipulating Data: Scaling a Vector

The previous example has shown how to submit tasks. In this section, we show how StarPU tasks can manipulate data. The full source code for this example is given in Appendix A [Full source code for the 'Scaling a Vector' example], page 77.

4.3.1 Source code of Vector Scaling

Programmers can describe the data layout of their application so that StarPU is responsible for enforcing data coherency and availability across the machine. Instead of handling complex (and non-portable) mechanisms to perform data movements, programmers only declare which piece of data is accessed and/or modified by a task, and StarPU makes sure that when a computational kernel starts somewhere (e.g. on a GPU), its data are available locally.

Before submitting those tasks, the programmer first needs to declare the different pieces of data to StarPU using the `starpu_*_data_register` functions. To ease the development of applications for StarPU, it is possible to describe multiple types of data layout. A type of data layout is called an **interface**. There are different predefined interfaces available in StarPU: here we will consider the **vector interface**.

The following lines show how to declare an array of `NX` elements of type `float` using the vector interface:

```
float vector[NX];

starpu_data_handle vector_handle;
starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector, NX,
                            sizeof(vector[0]));
```

The first argument, called the **data handle**, is an opaque pointer which designates the array in StarPU. This is also the structure which is used to describe which data is used by a task. The second argument is the node number where the data originally resides. Here it

is 0 since the `vector` array is in the main memory. Then comes the pointer `vector` where the data can be found in main memory, the number of elements in the vector and the size of each element. The following shows how to construct a StarPU task that will manipulate the vector and a constant factor.

```
float factor = 3.14;
struct starpu_task *task = starpu_task_create();

task->cl = &cl;                          /* Pointer to the codelet defined below */
task->buffers[0].handle = vector_handle; /* First parameter of the codelet */
task->buffers[0].mode = STARPU_RW;
task->cl_arg = &factor;
task->cl_arg_size = sizeof(factor);
task->synchronous = 1;

starpu_task_submit(task);
```

Since the factor is a mere constant float value parameter, it does not need a preliminary registration, and can just be passed through the `cl_arg` pointer like in the previous example. The vector parameter is described by its handle. There are two fields in each element of the `buffers` array. `handle` is the handle of the data, and `mode` specifies how the kernel will access the data (`STARPU_R` for read-only, `STARPU_W` for write-only and `STARPU_RW` for read and write access).

The definition of the codelet can be written as follows:

```
void scal_cpu_func(void *buffers[], void *cl_arg)
{
    unsigned i;
    float *factor = cl_arg;

    /* length of the vector */
    unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
    /* CPU copy of the vector pointer */
    float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);

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

starpu_codelet cl = {
    .where = STARPU_CPU,
    .cpu_func = scal_cpu_func,
    .nbuffers = 1
};
```

The first argument is an array that gives a description of all the buffers passed in the `task->buffers` array. The size of this array is given by the `nbuffers` field of the codelet structure. For the sake of genericity, this array contains pointers to the different interfaces describing each buffer. In the case of the **vector interface**, the location of the vector (resp. its length) is accessible in the `ptr` (resp. `nx`) of this array. Since the vector is accessed in a read-write fashion, any modification will automatically affect future accesses to this vector made by other tasks.

The second argument of the `scal_cpu_func` function contains a pointer to the parameters of the codelet (given in `task->cl_arg`), so that we read the constant factor from this pointer.

4.3.2 Execution of Vector Scaling

```
% make vector_scal
cc $(pkg-config --cflags libstarpu)  $(pkg-config --libs libstarpu)  vector_scal.c   -
o vector_scal
% ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

4.4 Vector Scaling on an Hybrid CPU/GPU Machine

Contrary to the previous examples, the task submitted in this example may not only be executed by the CPUs, but also by a CUDA device.

4.4.1 Definition of the CUDA Kernel

The CUDA implementation can be written as follows. It needs to be compiled with a CUDA compiler such as nvcc, the NVIDIA CUDA compiler driver. It must be noted that the vector pointer returned by STARPU_VECTOR_GET_PTR is here a pointer in GPU memory, so that it can be passed as such to the `vector_mult_cuda` kernel call.

```
#include <starpu.h>

static __global__ void vector_mult_cuda(float *val, unsigned n,
                                        float factor)
{
    unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        val[i] *= factor;
}

extern "C" void scal_cuda_func(void *buffers[], void *_args)
{
    float *factor = (float *)_args;

    /* length of the vector */
    unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
    /* CUDA copy of the vector pointer */
    float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    unsigned threads_per_block = 64;
    unsigned nblocks = (n + threads_per_block-1) / threads_per_block;

    vector_mult_cuda<<<nblocks,threads_per_block, 0, starpu_cuda_get_local_stream()>>>(val, n, *factor);

    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

4.4.2 Definition of the OpenCL Kernel

The OpenCL implementation can be written as follows. StarPU provides tools to compile a OpenCL kernel stored in a file.

```
__kernel void vector_mult_opencl(__global float* val, int nx, float factor)
{
        const int i = get_global_id(0);
        if (i < nx) {
                val[i] *= factor;
        }
}
```

Similarly to CUDA, the pointer returned by `STARPU_VECTOR_GET_PTR` is here a device pointer, so that it is passed as such to the OpenCL kernel.

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

extern struct starpu_opencl_program programs;

void scal_opencl_func(void *buffers[], void *_args)
{
    float *factor = _args;
    int id, devid, err;
    cl_kernel kernel;
    cl_command_queue queue;
    cl_event event;

    /* length of the vector */
    unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
    /* OpenCL copy of the vector pointer */
    cl_mem val = (cl_mem) STARPU_VECTOR_GET_PTR(buffers[0]);

    id = starpu_worker_get_id();
    devid = starpu_worker_get_devid(id);

    err = starpu_opencl_load_kernel(&kernel, &queue, &programs,
                    "vector_mult_opencl", devid);   /* Name of the codelet defined above */
    if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

    err = clSetKernelArg(kernel, 0, sizeof(val), &val);
    err |= clSetKernelArg(kernel, 1, sizeof(n), &n);
    err |= clSetKernelArg(kernel, 2, sizeof(*factor), factor);
    if (err) STARPU_OPENCL_REPORT_ERROR(err);

    {
        size_t global=1;
        size_t local=1;
        err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
    }

    clFinish(queue);
    starpu_opencl_collect_stats(event);
    clReleaseEvent(event);

    starpu_opencl_release_kernel(kernel);
}
```

4.4.3 Definition of the Main Code

The CPU implementation is the same as in the previous section.

Here is the source of the main application. You can notice the value of the field `where` for the codelet. We specify `STARPU_CPU|STARPU_CUDA|STARPU_OPENCL` to indicate to StarPU that the codelet can be executed either on a CPU or on a CUDA or an OpenCL device.

```
#include <starpu.h>

#define NX 2048

extern void scal_cuda_func(void *buffers[], void *_args);
extern void scal_cpu_func(void *buffers[], void *_args);
extern void scal_opencl_func(void *buffers[], void *_args);

/* Definition of the codelet */
static starpu_codelet cl = {
    .where = STARPU_CPU|STARPU_CUDA|STARPU_OPENCL; /* It can be executed on a CPU, */
                                    /* on a CUDA device, or on an OpenCL device */
    .cuda_func = scal_cuda_func;
    .cpu_func = scal_cpu_func;
    .opencl_func = scal_opencl_func;
    .nbuffers = 1;
}

#ifdef STARPU_USE_OPENCL
/* The compiled version of the OpenCL program */
struct starpu_opencl_program programs;
#endif

int main(int argc, char **argv)
{
    float *vector;
    int i, ret;
    float factor=3.0;
    struct starpu_task *task;
    starpu_data_handle vector_handle;

    starpu_init(NULL);                              /* Initialising StarPU */

#ifdef STARPU_USE_OPENCL
    starpu_opencl_load_opencl_from_file(
            "examples/basic_examples/vector_scal_opencl_codelet.cl",
            &programs, NULL);
#endif

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

```
    /* Registering data within StarPU */
    starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector,
                                NX, sizeof(vector[0]));

    /* Definition of the task */
    task = starpu_task_create();
    task->cl = &cl;
    task->buffers[0].handle = vector_handle;
    task->buffers[0].mode = STARPU_RW;
    task->cl_arg = &factor;
    task->cl_arg_size = sizeof(factor);
```

```
    /* Submitting the task */
    ret = starpu_task_submit(task);
    if (ret == -ENODEV) {
            fprintf(stderr, "No worker may execute this task\n");
            return 1;
    }

    /* Waiting for its termination */
    starpu_task_wait_for_all();

    /* Update the vector in RAM */
    starpu_data_acquire(vector_handle, STARPU_R);
```

```
    /* Access the data */
    for(i=0 ; i<NX; i++) {
      fprintf(stderr, "%f ", vector[i]);
    }
    fprintf(stderr, "\n");

    /* Release the RAM view of the data before unregistering it and shutting down StarPU */
    starpu_data_release(vector_handle);
    starpu_data_unregister(vector_handle);
    starpu_shutdown();

    return 0;
}
```

4.4.4 Execution of Hybrid Vector Scaling

The Makefile given at the beginning of the section must be extended to give the rules to compile the CUDA source code. Note that the source file of the OpenCL kernel does not need to be compiled now, it will be compiled at run-time when calling the function `starpu_opencl_load_opencl_from_file()` (see Section 9.12.2.1 [starpu_opencl_load_opencl_from_file], page 70).

```
CFLAGS += $(shell pkg-config --cflags libstarpu)
LDFLAGS += $(shell pkg-config --libs libstarpu)
CC = gcc

vector_scal: vector_scal.o vector_scal_cpu.o vector_scal_cuda.o vector_scal_opencl.o

%.o: %.cu
	nvcc $(CFLAGS) $< -c $
clean:
	rm -f vector_scal *.o
```

```
% make
```

and to execute it, with the default configuration:

```
% ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

or for example, by disabling CPU devices:

```
% STARPU_NCPUS=0 ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

or by disabling CUDA devices (which may permit to enable the use of OpenCL, see Section 3.4 [Using accelerators], page 7):

```
% STARPU_NCUDA=0 ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

4.5 Task and Worker Profiling

A full example showing how to use the profiling API is available in the StarPU sources in the directory `examples/profiling/`.

```
struct starpu_task *task = starpu_task_create();
task->cl = &cl;
task->synchronous = 1;
/* We will destroy the task structure by hand so that we can
 * query the profiling info before the task is destroyed. */
task->destroy = 0;

/* Submit and wait for completion (since synchronous was set to 1) */
starpu_task_submit(task);

/* The task is finished, get profiling information */
struct starpu_task_profiling_info *info = task->profiling_info;

/* How much time did it take before the task started ? */
double delay += starpu_timing_timespec_delay_us(&info->submit_time, &info->start_time);

/* How long was the task execution ? */
double length += starpu_timing_timespec_delay_us(&info->start_time, &info->end_time);

/* We don't need the task structure anymore */
starpu_task_destroy(task);
```

```
/* Display the occupancy of all workers during the test */
int worker;
for (worker = 0; worker < starpu_worker_get_count(); worker++)
{
        struct starpu_worker_profiling_info worker_info;
        int ret = starpu_worker_get_profiling_info(worker, &worker_info);
        STARPU_ASSERT(!ret);

        double total_time = starpu_timing_timespec_to_us(&worker_info.total_time);
        double executing_time = starpu_timing_timespec_to_us(&worker_info.executing_time);
        double sleeping_time = starpu_timing_timespec_to_us(&worker_info.sleeping_time);

        float executing_ratio = 100.0*executing_time/total_time;
        float sleeping_ratio = 100.0*sleeping_time/total_time;

        char workername[128];
        starpu_worker_get_name(worker, workername, 128);
        fprintf(stderr, "Worker %s:\n", workername);
        fprintf(stderr, "\ttotal time : %.2lf ms\n", total_time*1e-3);
        fprintf(stderr, "\texec time  : %.2lf ms (%.2f %%)\n", executing_time*1e-3,
                executing_ratio);
        fprintf(stderr, "\tblocked time  : %.2lf ms (%.2f %%)\n", sleeping_time*1e-3,
                sleeping_ratio);
}
```

4.6 Partitioning Data

An existing piece of data can be partitioned in sub parts to be used by different tasks, for instance:

```
int vector[NX];
starpu_data_handle handle;

/* Declare data to StarPU */
starpu_vector_data_register(&handle, 0, (uintptr_t)vector, NX, sizeof(vector[0]));

/* Partition the vector in PARTS sub-vectors */
starpu_filter f =
{
    .filter_func = starpu_block_filter_func_vector,
    .nchildren = PARTS,
    .get_nchildren = NULL,
    .get_child_ops = NULL
};
starpu_data_partition(handle, &f);
```

```
/* Submit a task on each sub-vector */
for (i=0; i<starpu_data_get_nb_children(handle); i++) {
    /* Get subdata number i (there is only 1 dimension) */
    starpu_data_handle sub_handle = starpu_data_get_sub_data(handle, 1, i);
    struct starpu_task *task = starpu_task_create();

    task->buffers[0].handle = sub_handle;
    task->buffers[0].mode = STARPU_RW;
    task->cl = &cl;
    task->synchronous = 1;
    task->cl_arg = &factor;
    task->cl_arg_size = sizeof(factor);

    starpu_task_submit(task);
}
```

Partitioning can be applied several times, see `examples/basic_examples/mult.c` and `examples/filters/`.

4.7 Performance model example

To achieve good scheduling, StarPU scheduling policies need to be able to estimate in advance the duration of a task. This is done by giving to codelets a performance model. There are several kinds of performance models.

- Providing an estimation from the application itself (`STARPU_COMMON` model type and `cost_model` field), see for instance `examples/common/blas_model.h` and `examples/common/blas_model.c`. It can also be provided for each architecture (`STARPU_PER_ARCH` model type and `per_arch` field)
- Measured at runtime (STARPU_HISTORY_BASED model type). This assumes that for a given set of data input/output sizes, the performance will always be about the same. This is very true for regular kernels on GPUs for instance (<0.1% error), and just a bit less true on CPUs (~=1% error). This also assumes that there are few different sets of data input/output sizes. StarPU will then keep record of the average time of previous executions on the various processing units, and use it as an estimation. History is done per task size, by using a hash of the input and ouput sizes as an index. It will also save it in `~/.starpu/sampling/codelets` for further executions, and can be observed by using the `starpu_perfmodel_display` command. The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. The following is a small code example.

```
static struct starpu_perfmodel_t mult_perf_model = {
    .type = STARPU_HISTORY_BASED,
    .symbol = "mult_perf_model"
};

starpu_codelet cl = {
    .where = STARPU_CPU,
    .cpu_func = cpu_mult,
    .nbuffers = 3,
    /* for the scheduling policy to be able to use performance models */
    .model = &mult_perf_model
};
```

- Measured at runtime and refined by regression (STARPU_REGRESSION_*_BASED model type). This still assumes performance regularity, but can work with various data input sizes, by applying regression over observed execution times. STARPU_REGRESSION_BASED uses an a*n^b regression form, STARPU_NL_REGRESSION_BASED uses an a*n^b+c (more precise than STARPU_REGRESSION_BASED, but costs a lot more to compute)
- Provided explicitly by the application (STARPU_PER_ARCH model type): the `.per_arch[i].cost_model` fields have to be filled with pointers to functions which return the expected duration of the task in micro-seconds, one per architecture.

How to use schedulers which can benefit from such performance model is explained in Section 5.4 [Task scheduling policy], page 25.

The same can be done for task power consumption estimation, by setting the `power_model` field the same way as the `model` field. Note: for now, the application has to give to the power consumption performance model a name which is different from the execution time performance model.

4.8 Theoretical lower bound on execution time

For kernels with history-based performance models, StarPU can very easily provide a theoretical lower bound for the execution time of a whole set of tasks. See for instance `examples/lu/lu_example.c`: before submitting tasks, call `starpu_bound_start`, and after complete execution, call `starpu_bound_stop`. `starpu_bound_print_lp` or `starpu_bound_print_mps` can then be used to output a Linear Programming problem corresponding to the schedule of your tasks. Run it through `lp_solve` or any other linear programming solver, and that will give you a lower bound for the total execution time of your tasks. If StarPU was compiled with the glpk library installed, `starpu_bound_compute` can be used to solve it immediately and get the optimized minimum. Its `integer` parameter allows to decide whether integer resolution should be computed and returned.

The `deps` parameter tells StarPU whether to take tasks and implicit data dependencies into account. It must be understood that the linear programming problem size is quadratic with the number of tasks and thus the time to solve it will be very long, it could be minutes for just a few dozen tasks. You should probably use `lp_solve -timeout 1 test.pl -wmps test.mps` to convert the problem to MPS format and then use a better solver, `glpsol` might be better than `lp_solve` for instance (the `--pcost` option may be useful), but sometimes doesn't manage to converge. `cbc` might look slower, but it is parallel. Be sure to try at least all the `-B` options of `lp_solve`. For instance, we often just use `lp_solve -cc -B1 -Bb -Bg -Bp -Bf -Br -BG -Bd -Bs -BB -Bo -Bc -Bi` , and the `-gr` option can also be quite useful.

Setting `deps` to 0 will only take into account the actual computations on processing units. It however still properly takes into account the varying performances of kernels and processing units, which is quite more accurate than just comparing StarPU performances with the fastest of the kernels being used.

The `prio` parameter tells StarPU whether to simulate taking into account the priorities as the StarPU scheduler would, i.e. schedule prioritized tasks before less prioritized tasks, to check to which extend this results to a less optimal solution. This increases even more computation time.

Note that for simplicity, all this however doesn't take into account data transfers, which are assumed to be completely overlapped.

4.9 Insert Task Utility

StarPU provides the wrapper function `starpu_insert_task` to ease the creation and submission of tasks.

`int starpu_insert_task` (*starpu_codelet *cl*, ...) [Function]

Create and submit a task corresponding to *cl* with the following arguments. The argument list must be zero-terminated.

The arguments following the codelets can be of the following types:

- `STARPU_R`, `STARPU_W`, `STARPU_RW`, `STARPU_SCRATCH`, `STARPU_REDUX` an access mode followed by a data handle;
- `STARPU_VALUE` followed by a pointer to a constant value and the size of the constant;
- `STARPU_CALLBACK` followed by a pointer to a callback function;
- `STARPU_CALLBACK_ARG` followed by a pointer to be given as an argument to the callback function;
- `STARPU_PRIORITY` followed by a integer defining a priority level.

Parameters to be passed to the codelet implementation are defined through the type `STARPU_VALUE`. The function `starpu_unpack_cl_args` must be called within the codelet implementation to retrieve them.

Here the implementation of the codelet:

```
void func_cpu(void *descr[], void *_args)
{
        int *x0 = (int *)STARPU_VARIABLE_GET_PTR(descr[0]);
        float *x1 = (float *)STARPU_VARIABLE_GET_PTR(descr[1]);
        int ifactor;
        float ffactor;

        starpu_unpack_cl_args(_args, &ifactor, &ffactor);
        *x0 = *x0 * ifactor;
        *x1 = *x1 * ffactor;
}

starpu_codelet mycodelet = {
        .where = STARPU_CPU,
        .cpu_func = func_cpu,
        .nbuffers = 2
};
```

And the call to the `starpu_insert_task` wrapper:

```
starpu_insert_task(&mycodelet,
                   STARPU_VALUE, &ifactor, sizeof(ifactor),
                   STARPU_VALUE, &ffactor, sizeof(ffactor),
                   STARPU_RW, data_handles[0], STARPU_RW, data_handles[1],
                   0);
```

The call to `starpu_insert_task` is equivalent to the following code:

```
struct starpu_task *task = starpu_task_create();
task->cl = &mycodelet;
task->buffers[0].handle = data_handles[0];
task->buffers[0].mode = STARPU_RW;
task->buffers[1].handle = data_handles[1];
task->buffers[1].mode = STARPU_RW;
char *arg_buffer;
size_t arg_buffer_size;
starpu_pack_cl_args(&arg_buffer, &arg_buffer_size,
    STARPU_VALUE, &ifactor, sizeof(ifactor),
    STARPU_VALUE, &ffactor, sizeof(ffactor),
    0);
task->cl_arg = arg_buffer;
task->cl_arg_size = arg_buffer_size;
int ret = starpu_task_submit(task);
```

4.10 Debugging

StarPU provides several tools to help debugging aplications. Execution traces can be generated and displayed graphically, see Section 6.2.1 [Generating traces], page 30. Some gdb helpers are also provided to show the whole StarPU state:

```
(gdb) source tools/gdbinit
(gdb) help starpu
```

4.11 More examples

More examples are available in the StarPU sources in the `examples/` directory. Simple examples include:

`incrementer/`:
: Trivial incrementation test.

`basic_examples/`:
: Simple documented Hello world (as shown in Section 4.2 [Hello World], page 9), vector/scalar product (as shown in Section 4.4 [Vector Scaling on an Hybrid CPU/GPU Machine], page 14), matrix product examples (as shown in Section 4.7 [Performance model example], page 20), an example using the blocked matrix data interface, and an example using the variable data interface.

`matvecmult/`:
: OpenCL example from NVidia, adapted to StarPU.

`axpy/`: AXPY CUBLAS operation adapted to StarPU.

`fortran/`: Example of Fortran bindings.

More advanced examples include:

`filters/`: Examples using filters, as shown in Section 4.6 [Partitioning Data], page 19.

`lu/`: LU matrix factorization, see for instance `xlu_implicit.c`

`cholesky/`:
: Cholesky matrix factorization, see for instance `cholesky_implicit.c`.

5 How to optimize performance with StarPU

TODO: improve!

Simply encapsulating application kernels into tasks already permits to seamlessly support CPU and GPUs at the same time. To achieve good performance, a few additional changes are needed.

5.1 Data management

When the application allocates data, whenever possible it should use the `starpu_malloc` function, which will ask CUDA or OpenCL to make the allocation itself and pin the corresponding allocated memory. This is needed to permit asynchronous data transfer, i.e. permit data transfer to overlap with computations.

By default, StarPU leaves replicates of data wherever they were used, in case they will be re-used by other tasks, thus saving the data transfer time. When some task modifies some data, all the other replicates are invalidated, and only the processing unit which ran that task will have a valid replicate of the data. If the application knows that this data will not be re-used by further tasks, it should advise StarPU to immediately replicate it to a desired list of memory nodes (given through a bitmask). This can be understood like the write-through mode of CPU caches.

```
starpu_data_set_wt_mask(img_handle, 1<<0);
```

will for instance request to always transfer a replicate into the main memory (node 0), as bit 0 of the write-through bitmask is being set.

5.2 Task submission

To let StarPU make online optimizations, tasks should be submitted asynchronously as much as possible. Ideally, all the tasks should be submitted, and mere calls to `starpu_task_wait_for_all` or `starpu_data_unregister` be done to wait for termination. StarPU will then be able to rework the whole schedule, overlap computation with communication, manage accelerator local memory usage, etc.

5.3 Task priorities

By default, StarPU will consider the tasks in the order they are submitted by the application. If the application programmer knows that some tasks should be performed in priority (for instance because their output is needed by many other tasks and may thus be a bottleneck if not executed early enough), the `priority` field of the task structure should be set to transmit the priority information to StarPU.

5.4 Task scheduling policy

By default, StarPU uses the `eager` simple greedy scheduler. This is because it provides correct load balance even if the application codelets do not have performance models. If your application codelets have performance models (see Section 4.7 [Performance model example], page 20 for examples showing how to do it), you should change the scheduler thanks to the `STARPU_SCHED` environment variable. For instance `export STARPU_SCHED=dmda` . Use `help` to get the list of available schedulers.

The **eager** scheduler uses a central task queue, from which workers draw tasks to work on. This however does not permit to prefetch data since the scheduling decision is taken late. If a task has a non-0 priority, it is put at the front of the queue.

The **prio** scheduler also uses a central task queue, but sorts tasks by priority (between -5 and 5).

The **random** scheduler distributes tasks randomly according to assumed worker overall performance.

The **ws** (work stealing) scheduler schedules tasks on the local worker by default. When a worker becomes idle, it steals a task from the most loaded worker.

The **dm** (deque model) scheduler uses task execution performance models into account to perform an HEFT-similar scheduling strategy: it schedules tasks where their termination time will be minimal.

The **dmda** (deque model data aware) scheduler is similar to dm, it also takes into account data transfer time.

The **dmdar** (deque model data aware ready) scheduler is similar to dmda, it also sorts tasks on per-worker queues by number of already-available data buffers.

The **dmdas** (deque model data aware sorted) scheduler is similar to dmda, it also supports arbitrary priority values.

The **heft** (HEFT) scheduler is similar to dmda, it also supports task bundles.

The **pheft** (parallel HEFT) scheduler is similar to heft, it also supports parallel tasks (still experimental).

The **pgreedy** (parallel greedy) scheduler is similar to greedy, it also supports parallel tasks (still experimental).

5.5 Performance model calibration

Most schedulers are based on an estimation of codelet duration on each kind of processing unit. For this to be possible, the application programmer needs to configure a performance model for the codelets of the application (see Section 4.7 [Performance model example], page 20 for instance). History-based performance models use on-line calibration. StarPU will automatically calibrate codelets which have never been calibrated yet, and save the result in `~/.starpu/sampling/codelets`. The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. To force continuing calibration, use `export STARPU_CALIBRATE=1` . This may be necessary if your application has not-so-stable performance. Details on the current performance model status can be obtained from the `starpu_perfmodel_display` command: the `-l` option lists the available performance models, and the `-s` option permits to choose the performance model to be displayed. The result looks like:

```
$ starpu_perfmodel_display -s starpu_dlu_lu_model_22
performance model for cpu
# hash size mean dev n
5c6c3401 1572864         1.216300e+04    2.277778e+03    1240
```

Which shows that for the LU 22 kernel with a 1.5MiB matrix, the average execution time on CPUs was about 12ms, with a 2ms standard deviation, over 1240 samples. It is a good idea to check this before doing actual performance measurements.

If a kernel source code was modified (e.g. performance improvement), the calibration information is stale and should be dropped, to re-calibrate from start. This can be done by using `export STARPU_CALIBRATE=2`.

Note: due to CUDA limitations, to be able to measure kernel duration, calibration mode needs to disable asynchronous data transfers. Calibration thus disables data transfer / computation overlapping, and should thus not be used for eventual benchmarks. Note 2: history-based performance models get calibrated only if a performance-model-based scheduler is chosen.

5.6 Task distribution vs Data transfer

Distributing tasks to balance the load induces data transfer penalty. StarPU thus needs to find a balance between both. The target function that the `dmda` scheduler of StarPU tries to minimize is `alpha * T_execution + beta * T_data_transfer`, where `T_execution` is the estimated execution time of the codelet (usually accurate), and `T_data_transfer` is the estimated data transfer time. The latter is estimated based on bus calibration before execution start, i.e. with an idle machine, thus without contention. You can force bus re-calibration by running `starpu_calibrate_bus`. The beta parameter defaults to 1, but it can be worth trying to tweak it by using `export STARPU_BETA=2` for instance, since during real application execution, contention makes transfer times bigger. This is of course imprecise, but in practice, a rough estimation already gives the good results that a precise estimation would give.

5.7 Data prefetch

The `heft`, `dmda` and `pheft` scheduling policies perform data prefetch (see Section 8.2.2.3 [STARPU_PREFETCH], page 44): as soon as a scheduling decision is taken for a task, requests are issued to transfer its required data to the target processing unit, if needeed, so that when the processing unit actually starts the task, its data will hopefully be already available and it will not have to wait for the transfer to finish.

The application may want to perform some manual prefetching, for several reasons such as excluding initial data transfers from performance measurements, or setting up an initial statically-computed data distribution on the machine before submitting tasks, which will thus guide StarPU toward an initial task distribution (since StarPU will try to avoid further transfers).

This can be achieved by giving the `starpu_data_prefetch_on_node` function the handle and the desired target memory node.

5.8 Power-based scheduling

If the application can provide some power performance model (through the `power_model` field of the codelet structure), StarPU will take it into account when distributing tasks. The target function that the `dmda` scheduler minimizes becomes `alpha * T_execution + beta * T_data_transfer + gamma * Consumption` , where `Consumption` is the estimated

task consumption in Joules. To tune this parameter, use `export STARPU_GAMMA=3000` for instance, to express that each Joule (i.e kW during 1000us) is worth 3000us execution time penalty. Setting `alpha` and `beta` to zero permits to only take into account power consumption.

This is however not sufficient to correctly optimize power: the scheduler would simply tend to run all computations on the most energy-conservative processing unit. To account for the consumption of the whole machine (including idle processing units), the idle power of the machine should be given by setting `export STARPU_IDLE_POWER=200` for 200W, for instance. This value can often be obtained from the machine power supplier.

The power actually consumed by the total execution can be displayed by setting `export STARPU_PROFILING=1 STARPU_WORKER_STATS=1` .

5.9 Profiling

A quick view of how many tasks each worker has executed can be obtained by setting `export STARPU_WORKER_STATS=1` This is a convenient way to check that execution did happen on accelerators without penalizing performance with the profiling overhead.

A quick view of how much data transfers have been issued can be obtained by setting `export STARPU_BUS_STATS=1` .

More detailed profiling information can be enabled by using `export STARPU_PROFILING=1` or by calling `starpu_profiling_status_set` from the source code. Statistics on the execution can then be obtained by using `export STARPU_BUS_STATS=1` and `export STARPU_WORKER_STATS=1` . More details on performance feedback are provided by the next chapter.

5.10 CUDA-specific optimizations

Due to CUDA limitations, StarPU will have a hard time overlapping its own communications and the codelet computations if the application does not use a dedicated CUDA stream for its computations. StarPU provides one by the use of `starpu_cuda_get_local_stream()` which should be used by all CUDA codelet operations. For instance:

```
func <<<grid,block,0,starpu_cuda_get_local_stream()>>> (foo, bar);
cudaStreamSynchronize(starpu_cuda_get_local_stream());
```

Unfortunately, some CUDA libraries do not have stream variants of kernels. That will lower the potential for overlapping.

6 Performance feedback

6.1 On-line performance feedback

6.1.1 Enabling on-line performance monitoring

In order to enable online performance monitoring, the application can call `starpu_profiling_status_set(STARPU_PROFILING_ENABLE)`. It is possible to detect whether monitoring is already enabled or not by calling `starpu_profiling_status_get()`. Enabling monitoring also reinitialize all previously collected feedback. The `STARPU_PROFILING` environment variable can also be set to 1 to achieve the same effect.

Likewise, performance monitoring is stopped by calling `starpu_profiling_status_set(STARPU_PROFILING_DISABLE)`. Note that this does not reset the performance counters so that the application may consult them later on.

More details about the performance monitoring API are available in section Section 9.10 [Profiling API], page 67.

6.1.2 Per-task feedback

If profiling is enabled, a pointer to a `starpu_task_profiling_info` structure is put in the `.profiling_info` field of the `starpu_task` structure when a task terminates. This structure is automatically destroyed when the task structure is destroyed, either automatically or by calling `starpu_task_destroy`.

The `starpu_task_profiling_info` structure indicates the date when the task was submitted (`submit_time`), started (`start_time`), and terminated (`end_time`), relative to the initialization of StarPU with `starpu_init`. It also specifies the identifier of the worker that has executed the task (`workerid`). These date are stored as `timespec` structures which the user may convert into micro-seconds using the `starpu_timing_timespec_to_us` helper function.

It it worth noting that the application may directly access this structure from the callback executed at the end of the task. The `starpu_task` structure associated to the callback currently being executed is indeed accessible with the `starpu_get_current_task()` function.

6.1.3 Per-codelet feedback

The `per_worker_stats` field of the `starpu_codelet_t` structure is an array of counters. The i-th entry of the array is incremented every time a task implementing the codelet is executed on the i-th worker. This array is not reinitialized when profiling is enabled or disabled.

6.1.4 Per-worker feedback

The second argument returned by the `starpu_worker_get_profiling_info` function is a `starpu_worker_profiling_info` structure that gives statistics about the specified worker. This structure specifies when StarPU started collecting profiling information for that worker (`start_time`), the duration of the profiling measurement interval (`total_time`), the time spent executing kernels (`executing_time`), the time spent sleeping because there is no

task to execute at all (`sleeping_time`), and the number of tasks that were executed while profiling was enabled. These values give an estimation of the proportion of time spent do real work, and the time spent either sleeping because there are not enough executable tasks or simply wasted in pure StarPU overhead.

Calling `starpu_worker_get_profiling_info` resets the profiling information associated to a worker.

When an FxT trace is generated (see Section 6.2.1 [Generating traces], page 30), it is also possible to use the `starpu_top` script (described in Section 6.2.4 [starpu-top], page 31) to generate a graphic showing the evolution of these values during the time, for the different workers.

6.1.5 Bus-related feedback

TODO

6.2 Off-line performance feedback

6.2.1 Generating traces with FxT

StarPU can use the FxT library (see `https://savannah.nongnu.org/projects/fkt/`) to generate traces with a limited runtime overhead.

You can either get a tarball:

```
% wget http://download.savannah.gnu.org/releases/fkt/fxt-0.2.2.tar.gz
```

or use the FxT library from CVS (autotools are required):

```
% cvs -d :pserver:anonymous@cvs.sv.gnu.org:/sources/fkt co FxT
% ./bootstrap
```

Compiling and installing the FxT library in the `$FXTDIR` path is done following the standard procedure:

```
% ./configure --prefix=$FXTDIR
% make
% make install
```

In order to have StarPU to generate traces, StarPU should be configured with the `--with-fxt` option:

```
$ ./configure --with-fxt=$FXTDIR
```

Or you can simply point the `PKG_CONFIG_PATH` to `$FXTDIR/lib/pkgconfig` and pass `--with-fxt` to `./configure`

When FxT is enabled, a trace is generated when StarPU is terminated by calling `starpu_shutdown()`). The trace is a binary file whose name has the form `prof_file_XXX_YYY` where `XXX` is the user name, and `YYY` is the pid of the process that used StarPU. This file is saved in the `/tmp/` directory by default, or by the directory specified by the `STARPU_FXT_PREFIX` environment variable.

6.2.2 Creating a Gantt Diagram

When the FxT trace file `filename` has been generated, it is possible to generate a trace in the Paje format by calling:

```
% starpu_fxt_tool -i filename
```

Or alternatively, setting the `STARPU_GENERATE_TRACE` environment variable to 1 before application execution will make StarPU do it automatically at application shutdown.

This will create a `paje.trace` file in the current directory that can be inspected with the ViTE trace visualizing open-source tool. More information about ViTE is available at `http://vite.gforge.inria.fr/`. It is possible to open the `paje.trace` file with ViTE by using the following command:

```
% vite paje.trace
```

6.2.3 Creating a DAG with graphviz

When the FxT trace file `filename` has been generated, it is possible to generate a task graph in the DOT format by calling:

```
$ starpu_fxt_tool -i filename
```

This will create a `dag.dot` file in the current directory. This file is a task graph described using the DOT language. It is possible to get a graphical output of the graph by using the graphviz library:

```
$ dot -Tpdf dag.dot -o output.pdf
```

6.2.4 Monitoring activity

When the FxT trace file `filename` has been generated, it is possible to generate a activity trace by calling:

```
$ starpu_fxt_tool -i filename
```

This will create an `activity.data` file in the current directory. A profile of the application showing the activity of StarPU during the execution of the program can be generated:

```
$ starpu_top.sh activity.data
```

This will create a file named `activity.eps` in the current directory. This picture is composed of two parts. The first part shows the activity of the different workers. The green sections indicate which proportion of the time was spent executed kernels on the processing unit. The red sections indicate the proportion of time spent in StartPU: an important overhead may indicate that the granularity may be too low, and that bigger tasks may be appropriate to use the processing unit more efficiently. The black sections indicate that the processing unit was blocked because there was no task to process: this may indicate a lack of parallelism which may be alleviated by creating more tasks when it is possible.

The second part of the `activity.eps` picture is a graph showing the evolution of the number of tasks available in the system during the execution. Ready tasks are shown in black, and tasks that are submitted but not schedulable yet are shown in grey.

6.3 Performance of codelets

The performance model of codelets can be examined by using the `starpu_perfmodel_display` tool:

```
$ starpu_perfmodel_display -l
file: <malloc_pinned.hannibal>
file: <starpu_slu_lu_model_21.hannibal>
```

```
file: <starpu_slu_lu_model_11.hannibal>
file: <starpu_slu_lu_model_22.hannibal>
file: <starpu_slu_lu_model_12.hannibal>
```

Here, the codelets of the lu example are available. We can examine the performance of the 22 kernel:

```
$ starpu_perfmodel_display -s starpu_slu_lu_model_22
performance model for cpu
# hash size mean dev n
57618ab0 19660800        2.851069e+05    1.829369e+04    109
performance model for cuda_0
# hash size mean dev n
57618ab0 19660800        1.164144e+04    1.556094e+01    315
performance model for cuda_1
# hash size mean dev n
57618ab0 19660800        1.164271e+04    1.330628e+01    360
performance model for cuda_2
# hash size mean dev n
57618ab0 19660800        1.166730e+04    3.390395e+02    456
```

We can see that for the given size, over a sample of a few hundreds of execution, the GPUs are about 20 times faster than the CPUs (numbers are in us). The standard deviation is extremely low for the GPUs, and less than 10% for CPUs.

7 StarPU MPI support

The integration of MPI transfers within task parallelism is done in a very natural way by the means of asynchronous interactions between the application and StarPU. This is implemented in a separate libstarpumpi library which basically provides "StarPU" equivalents of `MPI_*` functions, where `void *` buffers are replaced with `starpu_data_handle`s, and all GPU-RAM-NIC transfers are handled efficiently by StarPU-MPI.

7.1 The API

7.1.1 Initialisation

`int starpu_mpi_initialize` (*void*) [Function]
Initializes the starpumpi library. This must be called between calling `starpu_init` and other `starpu_mpi` functions. This function does not call `MPI_Init`, it should be called beforehand.

`int starpu_mpi_initialize_extended` (*int *rank*, *int *world_size*) [Function]
Initializes the starpumpi library. This must be called between calling `starpu_init` and other `starpu_mpi` functions. This function calls `MPI_Init`, and therefore should be prefered to the previous one for MPI implementations which are not thread-safe. Returns the current MPI node rank and world size.

`int starpu_mpi_shutdown` (*void*) [Function]
Cleans the starpumpi library. This must be called between calling `starpu_mpi` functions and `starpu_shutdown`. `MPI_Finalize` will be called if StarPU-MPI has been initialized by calling `starpu_mpi_initialize_extended`.

7.1.2 Communication

`int starpu_mpi_send` (*starpu_data_handle* `data_handle`, *int* `dest`, *int* `mpi_tag`, *MPI_Comm* `comm`) [Function]

`int starpu_mpi_recv` (*starpu_data_handle* `data_handle`, *int* `source`, *int* `mpi_tag`, *MPI_Comm* `comm`, *MPI_Status* *`status`) [Function]

`int starpu_mpi_isend` (*starpu_data_handle* `data_handle`, *starpu_mpi_req* *`req`, *int* `dest`, *int* `mpi_tag`, *MPI_Comm* `comm`) [Function]

`int starpu_mpi_irecv` (*starpu_data_handle* `data_handle`, *starpu_mpi_req* *`req`, *int* `source`, *int* `mpi_tag`, *MPI_Comm* `comm`) [Function]

`int starpu_mpi_isend_detached` (*starpu_data_handle* `data_handle`, *int* `dest`, *int* `mpi_tag`, *MPI_Comm* `comm`, *void* (*`callback`)(*void **), *void* *`arg`) [Function]

`int starpu_mpi_irecv_detached` (*starpu_data_handle* `data_handle`, *int* `source`, *int* `mpi_tag`, *MPI_Comm* `comm`, *void* (*`callback`)(*void **), *void* *`arg`) [Function]

`int starpu_mpi_wait` (*starpu_mpi_req* *`req`, *MPI_Status* *`status`) [Function]

`int starpu_mpi_test` (*starpu_mpi_req* *`req`, *int* *`flag`, *MPI_Status* *`status`) [Function]

`int starpu_mpi_barrier (`*MPI_Comm* `comm``)` [Function]

`int starpu_mpi_isend_detached_unlock_tag (`*starpu_data_handle* `data_handle`, *int* `dest`, *int* `mpi_tag`, *MPI_Comm* `comm`, *starpu_tag_t* `tag``)` [Function]

When the transfer is completed, the tag is unlocked

`int starpu_mpi_irecv_detached_unlock_tag (`*starpu_data_handle* `data_handle`, *int* `source`, *int* `mpi_tag`, *MPI_Comm* `comm`, *starpu_tag_t* `tag``)` [Function]

`int starpu_mpi_isend_array_detached_unlock_tag (`*unsigned* `array_size`, *starpu_data_handle* *`data_handle`, *int* *`dest`, *int* *`mpi_tag`, *MPI_Comm* *`comm`, *starpu_tag_t* `tag``)` [Function]

Asynchronously send an array of buffers, and unlocks the tag once all of them are transmitted.

`int starpu_mpi_irecv_array_detached_unlock_tag (`*unsigned* `array_size`, *starpu_data_handle* *`data_handle`, *int* *`source`, *int* *`mpi_tag`, *MPI_Comm* *`comm`, *starpu_tag_t* `tag``)` [Function]

7.2 Simple Example

```
void increment_token(void)
{
    struct starpu_task *task = starpu_task_create();

    task->cl = &increment_cl;
    task->buffers[0].handle = token_handle;
    task->buffers[0].mode = STARPU_RW;

    starpu_task_submit(task);
}
```

```
int main(int argc, char **argv)
{
    int rank, size;

    starpu_init(NULL);
    starpu_mpi_initialize_extended(&rank, &size);

    starpu_vector_data_register(&token_handle, 0, (uintptr_t)&token, 1, sizeof(unsigned));

    unsigned nloops = NITER;
    unsigned loop;

    unsigned last_loop = nloops - 1;
    unsigned last_rank = size - 1;
```

```
for (loop = 0; loop < nloops; loop++) {
    int tag = loop*size + rank;

    if (loop == 0 && rank == 0)
    {
        token = 0;
        fprintf(stdout, "Start with token value %d\n", token);
    }
    else
    {
        starpu_mpi_irecv_detached(token_handle, (rank+size-1)%size, tag,
                MPI_COMM_WORLD, NULL, NULL);
    }

    increment_token();

    if (loop == last_loop && rank == last_rank)
    {
        starpu_data_acquire(token_handle, STARPU_R);
        fprintf(stdout, "Finished : token value %d\n", token);
        starpu_data_release(token_handle);
    }
    else
    {
        starpu_mpi_isend_detached(token_handle, (rank+1)%size, tag+1,
                MPI_COMM_WORLD, NULL, NULL);
    }
}

starpu_task_wait_for_all();
```

```
starpu_mpi_shutdown();
starpu_shutdown();

if (rank == last_rank)
{
    fprintf(stderr, "[%d] token = %d == %d * %d ?\n", rank, token, nloops, size);
    STARPU_ASSERT(token == nloops*size);
}
```

7.3 MPI Insert Task Utility

`void starpu_mpi_insert_task` (*MPI_Comm* `comm`, *starpu_codelet* `*cl`, ...) [Function]

Create and submit a task corresponding to *cl* with the following arguments. The argument list must be zero-terminated.

The arguments following the codelets are the same types as for the function `starpu_insert_task` defined in Section 4.9 [Insert Task Utility], page 22. The extra argument `STARPU_EXECUTE` followed by an integer allows to specify the node to execute the codelet.

The algorithm is as follows:

1. Find out whether we are to execute the codelet because we own the data to be written to. If different tasks own data to be written to, the argument `STARPU_EXECUTE` should be used to specify the executing task `ET`.
2. Send and receive data as requested. Tasks owning data which need to be read by the executing task `ET` are sending them to `ET`.
3. Execute the codelet. This is done by the task selected in the 1st step of the algorithm.
4. In the case when different tasks own data to be written to, send W data back to their owners.

The algorithm also includes a cache mechanism that allows not to send data twice to the same task, unless the data has been modified.

`void starpu_mpi_get_data_on_node` (*MPI_Comm* `comm`, *starpu_data_handle* `data_handle`, *int* `node`) [Function]

Here an example showing how to use `starpu_mpi_insert_task`. One first needs to define a distribution function which specifies the locality of the data. Note that that distribution information needs to be given to StarPU by calling `starpu_data_set_rank`.

```
/* Returns the MPI node number where data is */
int my_distrib(int x, int y, int nb_nodes) {
/* Cyclic distrib */
return ((int)(x / sqrt(nb_nodes) + (y / sqrt(nb_nodes)) * sqrt(nb_nodes))) % nb_nodes;
        // /* Linear distrib */
        // return x / sqrt(nb_nodes) + (y / sqrt(nb_nodes)) * X;
}
```

Now the data can be registered within StarPU. Data which are not owned but will be needed for computations can be registered through the lazy allocation mechanism, i.e. with a `home_node` set to -1. StarPU will automatically allocate the memory when it is used for the first time.

```
unsigned matrix[X][Y];
starpu_data_handle data_handles[X][Y];

for(x = 0; x < X; x++) {
    for (y = 0; y < Y; y++) {
        int mpi_rank = my_distrib(x, y, size);
         if (mpi_rank == rank)
            /* Owning data */
            starpu_variable_data_register(&data_handles[x][y], 0,
                                          (uintptr_t)&(matrix[x][y]), sizeof(unsigned));
        else if (rank == mpi_rank+1 || rank == mpi_rank-1)
            /* I don't own that index, but will need it for my computations */
            starpu_variable_data_register(&data_handles[x][y], -1,
                                          (uintptr_t)NULL, sizeof(unsigned));
        else
            /* I know it's useless to allocate anything for this */
            data_handles[x][y] = NULL;
        if (data_handles[x][y])
            starpu_data_set_rank(data_handles[x][y], mpi_rank);
    }
}
```

Now `starpu_mpi_insert_task()` can be called for the different steps of the application.

```
for(loop=0 ; loop<niter; loop++)
    for (x = 1; x < X-1; x++)
        for (y = 1; y < Y-1; y++)
            starpu_mpi_insert_task(MPI_COMM_WORLD, &stencil5_cl,
                                   STARPU_RW, data_handles[x][y],
                                   STARPU_R, data_handles[x-1][y],
                                   STARPU_R, data_handles[x+1][y],
                                   STARPU_R, data_handles[x][y-1],
                                   STARPU_R, data_handles[x][y+1],
                                   0);
starpu_task_wait_for_all();
```

8 Configuring StarPU

8.1 Compilation configuration

The following arguments can be given to the `configure` script.

8.1.1 Common configuration

8.1.1.1 `--enable-debug`

Description:
Enable debugging messages.

8.1.1.2 `--enable-fast`

Description:
Do not enforce assertions, saves a lot of time spent to compute them otherwise.

8.1.1.3 `--enable-verbose`

Description:
Augment the verbosity of the debugging messages. This can be disabled at runtime by setting the environment variable `STARPU_SILENT` to any value.

```
% STARPU_SILENT=1 ./vector_scal
```

8.1.1.4 `--enable-coverage`

Description:
Enable flags for the `gcov` coverage tool.

8.1.2 Configuring workers

8.1.2.1 `--enable-nmaxcpus=<number>`

Description:
Defines the maximum number of CPU cores that StarPU will support, then available as the `STARPU_NMAXCPUS` macro.

8.1.2.2 `--disable-cpu`

Description:
Disable the use of CPUs of the machine. Only GPUs etc. will be used.

8.1.2.3 `--enable-maxcudadev=<number>`

Description:
Defines the maximum number of CUDA devices that StarPU will support, then available as the `STARPU_MAXCUDADEVS` macro.

8.1.2.4 `--disable-cuda`

Description:
Disable the use of CUDA, even if a valid CUDA installation was detected.

8.1.2.5 --with-cuda-dir=<path>

Description:
Specify the directory where CUDA is installed. This directory should notably contain `include/cuda.h`.

8.1.2.6 --with-cuda-include-dir=<path>

Description:
Specify the directory where CUDA headers are installed. This directory should notably contain `cuda.h`. This defaults to `/include` appended to the value given to `--with-cuda-dir`.

8.1.2.7 --with-cuda-lib-dir=<path>

Description:
Specify the directory where the CUDA library is installed. This directory should notably contain the CUDA shared libraries (e.g. libcuda.so). This defaults to `/lib` appended to the value given to `--with-cuda-dir`.

8.1.2.8 --enable-maxopencldev=<number>

Description:
Defines the maximum number of OpenCL devices that StarPU will support, then available as the `STARPU_MAXOPENCLDEVS` macro.

8.1.2.9 --disable-opencl

Description:
Disable the use of OpenCL, even if the SDK is detected.

8.1.2.10 --with-opencl-dir=<path>

Description:
Specify the location of the OpenCL SDK. This directory should notably contain `include/CL/cl.h` (or `include/OpenCL/cl.h` on Mac OS).

8.1.2.11 --with-opencl-include-dir=<path>

Description:
Specify the location of OpenCL headers. This directory should notably contain `CL/cl.h` (or `OpenCL/cl.h` on Mac OS). This defaults to `/include` appended to the value given to `--with-opencl-dir`.

8.1.2.12 --with-opencl-lib-dir=<path>

Description:
Specify the location of the OpenCL library. This directory should notably contain the OpenCL shared libraries (e.g. libOpenCL.so). This defaults to `/lib` appended to the value given to `--with-opencl-dir`.

8.1.2.13 `--enable-gordon`

Description:
Enable the use of the Gordon runtime for Cell SPUs.

8.1.2.14 `--with-gordon-dir=<path>`

Description:
Specify the location of the Gordon SDK.

8.1.3 Advanced configuration

8.1.3.1 `--enable-perf-debug`

Description:
Enable performance debugging.

8.1.3.2 `--enable-model-debug`

Description:
Enable performance model debugging.

8.1.3.3 `--enable-stats`

Description:
Enable statistics.

8.1.3.4 `--enable-maxbuffers=<nbuffers>`

Description:
Define the maximum number of buffers that tasks will be able to take as parameters, then available as the `STARPU_NMAXBUFS` macro.

8.1.3.5 `--enable-allocation-cache`

Description:
Enable the use of a data allocation cache to avoid the cost of it with CUDA. Still experimental.

8.1.3.6 `--enable-opengl-render`

Description:
Enable the use of OpenGL for the rendering of some examples.

8.1.3.7 `--enable-blas-lib=<name>`

Description:
Specify the blas library to be used by some of the examples. The library has to be 'atlas' or 'goto'.

8.1.3.8 `--with-magma=<path>`

Description:
Specify where magma is installed. This directory should notably contain `include/magmablas.h`.

8.1.3.9 `--with-fxt=<path>`

Description:
Specify the location of FxT (for generating traces and rendering them using ViTE). This directory should notably contain `include/fxt/fxt.h`.

8.1.3.10 `--with-perf-model-dir=<dir>`

Description:
Specify where performance models should be stored (instead of defaulting to the current user's home).

8.1.3.11 `--with-mpicc=<path to mpicc>`

Description:
Specify the location of the `mpicc` compiler to be used for starpumpi.

8.1.3.12 `--with-goto-dir=<dir>`

Description:
Specify the location of GotoBLAS.

8.1.3.13 `--with-atlas-dir=<dir>`

Description:
Specify the location of ATLAS. This directory should notably contain `include/cblas.h`.

8.1.3.14 `--with-mkl-cflags=<cflags>`

Description:
Specify the compilation flags for the MKL Library.

8.1.3.15 `--with-mkl-ldflags=<ldflags>`

Description:
Specify the linking flags for the MKL Library. Note that the http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/ website provides a script to determine the linking flags.

8.2 Execution configuration through environment variables

Note: the values given in `starpu_conf` structure passed when calling `starpu_init` will override the values of the environment variables.

8.2.1 Configuring workers

8.2.1.1 `STARPU_NCPUS` – Number of CPU workers

Description:
Specify the number of CPU workers (thus not including workers dedicated to control acceleratores). Note that by default, StarPU will not allocate more CPU workers than there are physical CPUs, and that some CPUs are used to control the accelerators.

8.2.1.2 STARPU_NCUDA – Number of CUDA workers

Description:

Specify the number of CUDA devices that StarPU can use. If STARPU_NCUDA is lower than the number of physical devices, it is possible to select which CUDA devices should be used by the means of the STARPU_WORKERS_CUDAID environment variable. By default, StarPU will create as many CUDA workers as there are CUDA devices.

8.2.1.3 STARPU_NOPENCL – Number of OpenCL workers

Description:

OpenCL equivalent of the STARPU_NCUDA environment variable.

8.2.1.4 STARPU_NGORDON – Number of SPU workers (Cell)

Description:

Specify the number of SPUs that StarPU can use.

8.2.1.5 STARPU_WORKERS_CPUID – Bind workers to specific CPUs

Description:

Passing an array of integers (starting from 0) in STARPU_WORKERS_CPUID specifies on which logical CPU the different workers should be bound. For instance, if STARPU_WORKERS_CPUID = "0 1 4 5", the first worker will be bound to logical CPU #0, the second CPU worker will be bound to logical CPU #1 and so on. Note that the logical ordering of the CPUs is either determined by the OS, or provided by the hwloc library in case it is available.

Note that the first workers correspond to the CUDA workers, then come the OpenCL and the SPU, and finally the CPU workers. For example if we have STARPU_NCUDA=1, STARPU_NOPENCL=1, STARPU_NCPUS=2 and STARPU_WORKERS_CPUID = "0 2 1 3", the CUDA device will be controlled by logical CPU #0, the OpenCL device will be controlled by logical CPU #2, and the logical CPUs #1 and #3 will be used by the CPU workers.

If the number of workers is larger than the array given in STARPU_WORKERS_CPUID, the workers are bound to the logical CPUs in a round-robin fashion: if STARPU_WORKERS_CPUID = "0 1", the first and the third (resp. second and fourth) workers will be put on CPU #0 (resp. CPU #1).

This variable is ignored if the use_explicit_workers_bindid flag of the starpu_conf structure passed to starpu_init is set.

8.2.1.6 STARPU_WORKERS_CUDAID – Select specific CUDA devices

Description:

Similarly to the STARPU_WORKERS_CPUID environment variable, it is possible to select which CUDA devices should be used by StarPU. On a machine equipped with 4 GPUs, setting STARPU_WORKERS_CUDAID = "1 3" and STARPU_NCUDA=2 specifies that 2 CUDA workers should be created, and that they should use CUDA devices #1 and #3 (the logical ordering of the devices is the one reported by CUDA).

This variable is ignored if the `use_explicit_workers_cuda_gpuid` flag of the `starpu_conf` structure passed to `starpu_init` is set.

8.2.1.7 STARPU_WORKERS_OPENCLID – Select specific OpenCL devices

Description:

OpenCL equivalent of the `STARPU_WORKERS_CUDAID` environment variable.

This variable is ignored if the `use_explicit_workers_opencl_gpuid` flag of the `starpu_conf` structure passed to `starpu_init` is set.

8.2.2 Configuring the Scheduling engine

8.2.2.1 STARPU_SCHED – Scheduling policy

Description:

This chooses between the different scheduling policies proposed by StarPU: work random, stealing, greedy, with performance models, etc.

Use `STARPU_SCHED=help` to get the list of available schedulers.

8.2.2.2 STARPU_CALIBRATE – Calibrate performance models

Description:

If this variable is set to 1, the performance models are calibrated during the execution. If it is set to 2, the previous values are dropped to restart calibration from scratch. Setting this variable to 0 disable calibration, this is the default behaviour.

Note: this currently only applies to `dm`, `dmda` and `heft` scheduling policies.

8.2.2.3 STARPU_PREFETCH – Use data prefetch

Description:

This variable indicates whether data prefetching should be enabled (0 means that it is disabled). If prefetching is enabled, when a task is scheduled to be executed e.g. on a GPU, StarPU will request an asynchronous transfer in advance, so that data is already present on the GPU when the task starts. As a result, computation and data transfers are overlapped. Note that prefetching is enabled by default in StarPU.

8.2.2.4 STARPU_SCHED_ALPHA – Computation factor

Description:

To estimate the cost of a task StarPU takes into account the estimated computation time (obtained thanks to performance models). The alpha factor is the coefficient to be applied to it before adding it to the communication part.

8.2.2.5 STARPU_SCHED_BETA – Communication factor

Description:

To estimate the cost of a task StarPU takes into account the estimated data transfer time (obtained thanks to performance models). The beta factor is the coefficient to be applied to it before adding it to the computation part.

8.2.3 Miscellaneous and debug

8.2.3.1 STARPU_SILENT – Disable verbose mode

Description:
This variable allows to disable verbose mode at runtime when StarPU has been configured with the option `--enable-verbose`.

8.2.3.2 STARPU_LOGFILENAME – Select debug file name

Description:
This variable specifies in which file the debugging output should be saved to.

8.2.3.3 STARPU_FXT_PREFIX – FxT trace location

Description
This variable specifies in which directory to save the trace generated if FxT is enabled. It needs to have a trailing '/' character.

8.2.3.4 STARPU_LIMIT_GPU_MEM – Restrict memory size on the GPUs

Description
This variable specifies the maximum number of megabytes that should be available to the application on each GPUs. In case this value is smaller than the size of the memory of a GPU, StarPU pre-allocates a buffer to waste memory on the device. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory.

8.2.3.5 STARPU_GENERATE_TRACE – Generate a Paje trace when StarPU is shut down

Description
When set to 1, this variable indicates that StarPU should automatically generate a Paje trace when starpu_shutdown is called.

9 StarPU API

9.1 Initialization and Termination

9.1.1 `starpu_init` – Initialize StarPU

Description:

This is StarPU initialization method, which must be called prior to any other StarPU call. It is possible to specify StarPU's configuration (e.g. scheduling policy, number of cores, ...) by passing a non-null argument. Default configuration is used if the passed argument is `NULL`.

Return value:

Upon successful completion, this function returns 0. Otherwise, `-ENODEV` indicates that no worker was available (so that StarPU was not initialized).

Prototype: `int starpu_init(struct starpu_conf *conf);`

9.1.2 `struct starpu_conf` – StarPU runtime configuration

Description:

This structure is passed to the `starpu_init` function in order to configure StarPU. When the default value is used, StarPU automatically selects the number of processing units and takes the default scheduling policy. This parameter overwrites the equivalent environment variables.

Fields:

`sched_policy_name` (default = NULL):

This is the name of the scheduling policy. This can also be specified with the `STARPU_SCHED` environment variable.

`sched_policy` (default = NULL):

This is the definition of the scheduling policy. This field is ignored if `sched_policy_name` is set.

`ncpus` (default = -1):

This is the number of CPU cores that StarPU can use. This can also be specified with the `STARPU_NCPUS` environment variable.

`ncuda` (default = -1):

This is the number of CUDA devices that StarPU can use. This can also be specified with the `STARPU_NCUDA` environment variable.

`nopencl` (default = -1):

This is the number of OpenCL devices that StarPU can use. This can also be specified with the `STARPU_NOPENCL` environment variable.

`nspus` (default = -1):

This is the number of Cell SPUs that StarPU can use. This can also be specified with the `STARPU_NGORDON` environment variable.

`use_explicit_workers_bindid` (default = 0)
: If this flag is set, the `workers_bindid` array indicates where the different workers are bound, otherwise StarPU automatically selects where to bind the different workers unless the `STARPU_WORKERS_CPUID` environment variable is set. The `STARPU_WORKERS_CPUID` environment variable is ignored if the `use_explicit_workers_bindid` flag is set.

`workers_bindid[STARPU_NMAXWORKERS]`
: If the `use_explicit_workers_bindid` flag is set, this array indicates where to bind the different workers. The i-th entry of the `workers_bindid` indicates the logical identifier of the processor which should execute the i-th worker. Note that the logical ordering of the CPUs is either determined by the OS, or provided by the `hwloc` library in case it is available. When this flag is set, the Section 8.2.1.5 [STARPU_WORKERS_CPUID], page 43 environment variable is ignored.

`use_explicit_workers_cuda_gpuid` (default = 0)
: If this flag is set, the CUDA workers will be attached to the CUDA devices specified in the `workers_cuda_gpuid` array. Otherwise, StarPU affects the CUDA devices in a round-robin fashion. When this flag is set, the Section 8.2.1.6 [STARPU_WORKERS_CUDAID], page 43 environment variable is ignored.

`workers_cuda_gpuid[STARPU_NMAXWORKERS]`
: If the `use_explicit_workers_cuda_gpuid` flag is set, this array contains the logical identifiers of the CUDA devices (as used by `cudaGetDevice`).

`use_explicit_workers_opencl_gpuid` (default = 0)
: If this flag is set, the OpenCL workers will be attached to the OpenCL devices specified in the `workers_opencl_gpuid` array. Otherwise, StarPU affects the OpenCL devices in a round-robin fashion.

`workers_opencl_gpuid[STARPU_NMAXWORKERS]`:
`calibrate` (default = 0):
: If this flag is set, StarPU will calibrate the performance models when executing tasks. If this value is equal to -1, the default value is used. The default value is overwritten by the `STARPU_CALIBRATE` environment variable when it is set.

9.1.3 `starpu_conf_init` – Initialize starpu_conf structure

This function initializes the `starpu_conf` structure passed as argument with the default values. In case some configuration parameters are already specified through environment variables, `starpu_conf_init` initializes the fields of the structure according to the environment variables. For instance if `STARPU_`

`CALIBRATE` is set, its value is put in the `.ncuda` field of the structure passed as argument.

Return value:
Upon successful completion, this function returns 0. Otherwise, `-EINVAL` indicates that the argument was NULL.

Prototype: `int starpu_conf_init(struct starpu_conf *conf);`

9.1.4 `starpu_shutdown` – Terminate StarPU

`void starpu_shutdown (void)` [Function]
This is StarPU termination method. It must be called at the end of the application: statistics and other post-mortem debugging information are not guaranteed to be available until this method has been called.

9.2 Workers' Properties

9.2.1 `starpu_worker_get_count` – Get the number of processing units

`unsigned starpu_worker_get_count (void)` [Function]
This function returns the number of workers (i.e. processing units executing StarPU tasks). The returned value should be at most `STARPU_NMAXWORKERS`.

9.2.2 `starpu_worker_get_count_by_type` – Get the number of processing units of a given type

`int starpu_worker_get_count_by_type (enum starpu_archtype type)` [Function]
Returns the number of workers of the type indicated by the argument. A positive (or null) value is returned in case of success, `-EINVAL` indicates that the type is not valid otherwise.

9.2.3 `starpu_cpu_worker_get_count` – Get the number of CPU controlled by StarPU

`unsigned starpu_cpu_worker_get_count (void)` [Function]
This function returns the number of CPUs controlled by StarPU. The returned value should be at most `STARPU_NMAXCPUS`.

9.2.4 `starpu_cuda_worker_get_count` – Get the number of CUDA devices controlled by StarPU

`unsigned starpu_cuda_worker_get_count (void)` [Function]
This function returns the number of CUDA devices controlled by StarPU. The returned value should be at most `STARPU_MAXCUDADEVS`.

9.2.5 starpu_opencl_worker_get_count – Get the number of OpenCL devices controlled by StarPU

`unsigned starpu_opencl_worker_get_count (void)` [Function]

This function returns the number of OpenCL devices controlled by StarPU. The returned value should be at most `STARPU_MAXOPENCLDEVS`.

9.2.6 starpu_spu_worker_get_count – Get the number of Cell SPUs controlled by StarPU

`unsigned starpu_opencl_worker_get_count (void)` [Function]

This function returns the number of Cell SPUs controlled by StarPU.

9.2.7 starpu_worker_get_id – Get the identifier of the current worker

`int starpu_worker_get_id (void)` [Function]

This function returns the identifier of the worker associated to the calling thread. The returned value is either -1 if the current context is not a StarPU worker (i.e. when called from the application outside a task or a callback), or an integer between 0 and `starpu_worker_get_count() - 1`.

9.2.8 starpu_worker_get_ids_by_type – Get the list of identifiers of workers with a given type

`int starpu_worker_get_ids_by_type (enum starpu_archtype type, int *workerids, int maxsize)` [Function]

Fill the workerids array with the identifiers of the workers that have the type indicated in the first argument. The maxsize argument indicates the size of the workids array. The returned value gives the number of identifiers that were put in the array. `-ERANGE` is returned is maxsize is lower than the number of workers with the appropriate type: in that case, the array is filled with the maxsize first elements. To avoid such overflows, the value of maxsize can be chosen by the means of the `starpu_worker_get_count_by_type` function, or by passing a value greater or equal to `STARPU_NMAXWORKERS`.

9.2.9 starpu_worker_get_devid – Get the device identifier of a worker

`int starpu_worker_get_devid (int id)` [Function]

This functions returns the device id of the worker associated to an identifier (as returned by the `starpu_worker_get_id` function). In the case of a CUDA worker, this device identifier is the logical device identifier exposed by CUDA (used by the `cudaGetDevice` function for instance). The device identifier of a CPU worker is the logical identifier of the core on which the worker was bound; this identifier is either provided by the OS or by the `hwloc` library in case it is available.

9.2.10 starpu_worker_get_type – Get the type of processing unit associated to a worker

`enum starpu_archtype starpu_worker_get_type (int id)` [Function]

This function returns the type of worker associated to an identifier (as returned by the `starpu_worker_get_id` function). The returned value indicates the architecture of

the worker: `STARPU_CPU_WORKER` for a CPU core, `STARPU_CUDA_WORKER` for a CUDA device, `STARPU_OPENCL_WORKER` for a OpenCL device, and `STARPU_GORDON_WORKER` for a Cell SPU. The value returned for an invalid identifier is unspecified.

9.2.11 `starpu_worker_get_name` – Get the name of a worker

`void starpu_worker_get_name (int id, char *dst, size_t maxlen)` [Function]

StarPU associates a unique human readable string to each processing unit. This function copies at most the *maxlen* first bytes of the unique string associated to a worker identified by its identifier *id* into the *dst* buffer. The caller is responsible for ensuring that the *dst* is a valid pointer to a buffer of *maxlen* bytes at least. Calling this function on an invalid identifier results in an unspecified behaviour.

9.2.12 `starpu_worker_get_memory_node` – Get the memory node of a worker

`unsigned starpu_worker_get_memory_node (unsigned workerid)` [Function]

This function returns the identifier of the memory node associated to the worker identified by *workerid*.

9.3 Data Library

This section describes the data management facilities provided by StarPU.

We show how to use existing data interfaces in Section 9.4 [Data Interfaces], page 54, but developers can design their own data interfaces if required.

9.3.1 `starpu_malloc` – Allocate data and pin it

`int starpu_malloc (void **A, size_t dim)` [Function]

This function allocates data of the given size in main memory. It will also try to pin it in CUDA or OpenCL, so that data transfers from this buffer can be asynchronous, and thus permit data transfer and computation overlapping. The allocated buffer must be freed thanks to the `starpu_free` function.

9.3.2 `starpu_access_mode` – Data access mode

This datatype describes a data access mode. The different available modes are:

`STARPU_R` read-only mode.
`STARPU_W` write-only mode.
`STARPU_RW` read-write mode. This is equivalent to `STARPU_R|STARPU_W`.
`STARPU_SCRATCH` scratch memory. A temporary buffer is allocated for the task, but StarPU does not enforce data consistency, i.e. each device has its own buffer, independently from each other (even for CPUs). This is useful for temporary variables. For now, no behaviour is defined concerning the relation with STARPU_R/W modes and the value provided at registration, i.e. the value of the scratch buffer is undefined at entry of the codelet function, but this is being considered for future extensions.
`STARPU_REDUX` reduction mode. TODO: document, as well as `starpu_data_set_reduction_methods`

9.3.3 `unsigned memory_node` – Memory node

Description:

Every worker is associated to a memory node which is a logical abstraction of the address space from which the processing unit gets its data. For instance, the memory node associated to the different CPU workers represents main memory (RAM), the memory node associated to a GPU is DRAM embedded on the device. Every memory node is identified by a logical index which is accessible from the `starpu_worker_get_memory_node` function. When registering a piece of data to StarPU, the specified memory node indicates where the piece of data initially resides (we also call this memory node the home node of a piece of data).

9.3.4 `starpu_data_handle` – StarPU opaque data handle

Description:

StarPU uses `starpu_data_handle` as an opaque handle to manage a piece of data. Once a piece of data has been registered to StarPU, it is associated to a `starpu_data_handle` which keeps track of the state of the piece of data over the entire machine, so that we can maintain data consistency and locate data replicates for instance.

9.3.5 `void *interface` – StarPU data interface

Description:

Data management is done at a high-level in StarPU: rather than accessing a mere list of contiguous buffers, the tasks may manipulate data that are described by a high-level construct which we call data interface.

An example of data interface is the `"vector"` interface which describes a contiguous data array on a spefic memory node. This interface is a simple structure containing the number of elements in the array, the size of the elements, and the address of the array in the appropriate address space (this address may be invalid if there is no valid copy of the array in the memory node). More informations on the data interfaces provided by StarPU are given in Section 9.4 [Data Interfaces], page 54.

When a piece of data managed by StarPU is used by a task, the task implementation is given a pointer to an interface describing a valid copy of the data that is accessible from the current processing unit.

9.3.6 `starpu_data_register` – Register a piece of data to StarPU

`void starpu_data_register` (*starpu_data_handle* ***handleptr***, *uint32_t* ***home_node***, *void* ***interface***, *struct starpu_data_interface_ops_t* ***ops***) [Function]

Register a piece of data into the handle located at the *handleptr* address. The *interface* buffer contains the initial description of the data in the home node. The *ops* argument is a pointer to a structure describing the different methods used to manipulate this type of interface. See Section 10.1.1 [struct starpu_data_interface_ops_t], page 73 for more details on this structure.

If `home_node` is -1, StarPU will automatically allocate the memory when it is used for the first time in write-only mode. Once such data handle has been automatically allocated, it is possible to access it using any access mode.

Note that StarPU supplies a set of predefined types of interface (e.g. vector or matrix) which can be registered by the means of helper functions (e.g. `starpu_vector_data_register` or `starpu_matrix_data_register`).

9.3.7 `starpu_data_unregister` – Unregister a piece of data from StarPU

`void starpu_data_unregister (`*starpu_data_handle* `handle``)` [Function]

This function unregisters a data handle from StarPU. If the data was automatically allocated by StarPU because the home node was -1, all automatically allocated buffers are freed. Otherwise, a valid copy of the data is put back into the home node in the buffer that was initially registered. Using a data handle that has been unregistered from StarPU results in an undefined behaviour.

9.3.8 `starpu_data_invalidate` – Invalidate all data replicates

`void starpu_data_invalidate (`*starpu_data_handle* `handle``)` [Function]

Destroy all replicates of the data handle. After data invalidation, the first access to the handle must be performed in write-only mode. Accessing an invalidated data in read-mode results in undefined behaviour.

9.3.9 `starpu_data_acquire` – Access registered data from the application

`int starpu_data_acquire (`*starpu_data_handle* `handle`, *starpu_access_mode* `mode``)` [Function]

The application must call this function prior to accessing registered data from main memory outside tasks. StarPU ensures that the application will get an up-to-date copy of the data in main memory located where the data was originally registered, and that all concurrent accesses (e.g. from tasks) will be consistent with the access mode specified in the *mode* argument. `starpu_data_release` must be called once the application does not need to access the piece of data anymore. Note that implicit data dependencies are also enforced by `starpu_data_acquire`, i.e. `starpu_data_acquire` will wait for all tasks scheduled to work on the data, unless that they have not been disabled explictly by calling `starpu_data_set_default_sequential_consistency_flag` or `starpu_data_set_sequential_consistency_flag`. `starpu_data_acquire` is a blocking call, so that it cannot be called from tasks or from their callbacks (in that case, `starpu_data_acquire` returns `-EDEADLK`). Upon successful completion, this function returns 0.

9.3.10 `starpu_data_acquire_cb` – Access registered data from the application asynchronously

`int starpu_data_acquire_cb (`*starpu_data_handle* `handle`, *starpu_access_mode* `mode`, *void* (*`callback`)(*void **), *void* *`arg`)` [Function]

`starpu_data_acquire_cb` is the asynchronous equivalent of `starpu_data_release`. When the data specified in the first argument is available in the appropriate access

mode, the callback function is executed. The application may access the requested data during the execution of this callback. The callback function must call `starpu_data_release` once the application does not need to access the piece of data anymore. Note that implicit data dependencies are also enforced by `starpu_data_acquire_cb` in case they are enabled. Contrary to `starpu_data_acquire`, this function is non-blocking and may be called from task callbacks. Upon successful completion, this function returns 0.

9.3.11 `starpu_data_release` – Release registered data from the application

`void starpu_data_release` (*starpu_data_handle* **`handle`**) [Function]

This function releases the piece of data acquired by the application either by `starpu_data_acquire` or by `starpu_data_acquire_cb`.

9.3.12 `starpu_data_set_wt_mask` – Set the Write-Through mask

`void starpu_data_set_wt_mask` (*starpu_data_handle* **`handle`**, *uint32_t* **`wt_mask`**) [Function]

This function sets the write-through mask of a given data, i.e. a bitmask of nodes where the data should be always replicated after modification.

9.3.13 `starpu_data_prefetch_on_node` – Prefetch data to a given node

`int starpu_data_prefetch_on_node` (*starpu_data_handle* **`handle`**, *unsigned* **`node`**, *unsigned* **`async`**) [Function]

Issue a prefetch request for a given data to a given node, i.e. requests that the data be replicated to the given node, so that it is available there for tasks. If the *async* parameter is 0, the call will block until the transfer is achieved, else the call will return as soon as the request is scheduled (which may however have to wait for a task completion).

9.4 Data Interfaces

9.4.1 Variable Interface

Description:

This variant of `starpu_data_register` uses the variable interface, i.e. for a mere single variable. `ptr` is the address of the variable, and `elemsize` is the size of the variable.

Prototype: `void starpu_variable_data_register(starpu_data_handle *handle, uint32_t home_node, uintptr_t ptr, size_t elemsize);`

Example:

```
float var;
starpu_data_handle var_handle;
starpu_variable_data_register(&var_handle, 0, (uintptr_t)&var, sizeof(var));
```

9.4.2 Vector Interface

Description:
This variant of `starpu_data_register` uses the vector interface, i.e. for mere arrays of elements. `ptr` is the address of the first element in the home node. `nx` is the number of elements in the vector. `elemsize` is the size of each element.

Prototype: `void starpu_vector_data_register(starpu_data_handle *handle, uint32_t home_node, uintptr_t ptr, uint32_t nx, size_t elemsize);`

Example:

```
float vector[NX];
starpu_data_handle vector_handle;
starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector, NX,
                            sizeof(vector[0]));
```

9.4.3 Matrix Interface

Description:
This variant of `starpu_data_register` uses the matrix interface, i.e. for matrices of elements. `ptr` is the address of the first element in the home node. `ld` is the number of elements between rows. `nx` is the number of elements in a row (this can be different from `ld` if there are extra elements for alignment for instance). `ny` is the number of rows. `elemsize` is the size of each element.

Prototype: `void starpu_matrix_data_register(starpu_data_handle *handle, uint32_t home_node, uintptr_t ptr, uint32_t ld, uint32_t nx, uint32_t ny, size_t elemsize);`

Example:

```
float *matrix;
starpu_data_handle matrix_handle;
matrix = (float*)malloc(width * height * sizeof(float));
starpu_matrix_data_register(&matrix_handle, 0, (uintptr_t)matrix,
                            width, width, height, sizeof(float));
```

9.4.4 3D Matrix Interface

Description:
This variant of `starpu_data_register` uses the 3D matrix interface. `ptr` is the address of the array of first element in the home node. `ldy` is the number of elements between rows. `ldz` is the number of rows between z planes. `nx` is the number of elements in a row (this can be different from `ldy` if there are extra elements for alignment for instance). `ny` is the number of rows in a z plane (likewise with `ldz`). `nz` is the number of z planes. `elemsize` is the size of each element.

Prototype: `void starpu_block_data_register(starpu_data_handle *handle, uint32_t home_node, uintptr_t ptr, uint32_t ldy, uint32_t ldz, uint32_t nx, uint32_t ny, uint32_t nz, size_t elemsize);`

Example:

```
float *block;
starpu_data_handle block_handle;
block = (float*)malloc(nx*ny*nz*sizeof(float));
starpu_block_data_register(&block_handle, 0, (uintptr_t)block,
                           nx, nx*ny, nx, ny, nz, sizeof(float));
```

9.4.5 BCSR Interface for Sparse Matrices (Blocked Compressed Sparse Row Representation)

`void starpu_bcsr_data_register` (*starpu_data_handle* ***handle***, *uint32_t* ***home_node***, *uint32_t* ***nnz***, *uint32_t* ***nrow***, *uintptr_t* ***nzval***, *uint32_t* ****colind***, *uint32_t* ****rowptr***, *uint32_t* ***firstentry***, *uint32_t* ***r***, *uint32_t* ***c***, *size_t* ***elemsize***) [Function]

This variant of `starpu_data_register` uses the BCSR sparse matrix interface. TODO

9.4.6 CSR Interface for Sparse Matrices (Compressed Sparse Row Representation)

`void starpu_csr_data_register` (*starpu_data_handle* ***handle***, *uint32_t* ***home_node***, *uint32_t* ***nnz***, *uint32_t* ***nrow***, *uintptr_t* ***nzval***, *uint32_t* ****colind***, *uint32_t* ****rowptr***, *uint32_t* ***firstentry***, *size_t* ***elemsize***) [Function]

This variant of `starpu_data_register` uses the CSR sparse matrix interface. TODO

9.5 Data Partition

9.5.1 `struct starpu_data_filter` – StarPU filter structure

Description:

The filter structure describes a data partitioning operation, to be given to the `starpu_data_partition` function, see Section 9.5.2 [starpu_data_partition], page 57 for an example.

Fields:

`filter_func`:
: This function fills the `child_interface` structure with interface information for the `id`-th child of the parent `father_interface` (among `nparts`). `void (*filter_func)(void *father_interface, void* child_interface, struct starpu_data_filter *, unsigned id, unsigned nparts);`

`nchildren`:
: This is the number of parts to partition the data into.

`get_nchildren`:
: This returns the number of children. This can be used instead of `nchildren` when the number of children depends on the

actual data (e.g. the number of blocks in a sparse matrix). `unsigned (*get_nchildren)(struct starpu_data_filter *, starpu_data_handle initial_handle);`

`get_child_ops`:
In case the resulting children use a different data interface, this function returns which interface is used by child number `id`. `struct starpu_data_interface_ops_t *(*get_child_ops)(struct starpu_data_filter *, unsigned id);`

`filter_arg`:
Some filters take an addition parameter, but this is usually unused.

`filter_arg_ptr`:
Some filters take an additional array parameter like the sizes of the parts, but this is usually unused.

9.5.2 starpu_data_partition – Partition Data

Description:
This requests partitioning one StarPU data `initial_handle` into several subdata according to the filter `f`

Prototype: `void starpu_data_partition(starpu_data_handle initial_handle, struct starpu_data_filter *f);`

Example:

```
struct starpu_data_filter f = {
    .filter_func = starpu_vertical_block_filter_func,
    .nchildren = nslicesx,
    .get_nchildren = NULL,
    .get_child_ops = NULL
};
starpu_data_partition(A_handle, &f);
```

9.5.3 starpu_data_unpartition – Unpartition data

Description:
This unapplies one filter, thus unpartitioning the data. The pieces of data are collected back into one big piece in the `gathering_node` (usually 0).

Prototype: `void starpu_data_unpartition(starpu_data_handle root_data, uint32_t gathering_node);`

Example:

```
starpu_data_unpartition(A_handle, 0);
```

9.5.4 starpu_data_get_nb_children

Description:
This function returns the number of children.

Return value:
The number of children.

Prototype: `int starpu_data_get_nb_children(starpu_data_handle handle);`

9.5.5 starpu_data_get_sub_data

Description:
After partitioning a StarPU data by applying a filter, `starpu_data_get_sub_data` can be used to get handles for each of the data portions. `root_data` is the parent data that was partitioned. `depth` is the number of filters to traverse (in case several filters have been applied, to e.g. partition in row blocks, and then in column blocks), and the subsequent parameters are the indexes.

Return value:
A handle to the subdata.

Prototype: `starpu_data_handle starpu_data_get_sub_data(starpu_data_handle root_data, unsigned depth, ... );`

Example:

```
h = starpu_data_get_sub_data(A_handle, 1, taskx);
```

9.5.6 Predefined filter functions

This section gives a partial list of the predefined partitioning functions. Examples on how to use them are shown in Section 4.6 [Partitioning Data], page 19. The complete list can be found in `starpu_data_filters.h` .

9.5.6.1 Partitioning BCSR Data

`void starpu_canonical_block_filter_bcsr (`*void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts*`)` [Function]
TODO

`void starpu_vertical_block_filter_func_csr (`*void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts*`)` [Function]
TODO

9.5.6.2 Partitioning BLAS interface

`void starpu_block_filter_func (`*void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts*`)` [Function]
This partitions a dense Matrix into horizontal blocks.

`void starpu_vertical_block_filter_func (`*void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts*`)` [Function]
This partitions a dense Matrix into vertical blocks.

9.5.6.3 Partitioning Vector Data

`void starpu_block_filter_func_vector (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)` [Function]

This partitions a vector into blocks of the same size.

`void starpu_vector_list_filter_func (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)` [Function]

This partitions a vector into blocks of sizes given in *filter_arg_ptr*.

`void starpu_vector_divide_in_2_filter_func (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)` [Function]

This partitions a vector into two blocks, the first block size being given in *filter_arg*.

9.5.6.4 Partitioning Block Data

`void starpu_block_filter_func_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)` [Function]

This partitions a 3D matrix along the X axis.

9.6 Codelets and Tasks

This section describes the interface to manipulate codelets and tasks.

`struct starpu_codelet` [Data Type]

The codelet structure describes a kernel that is possibly implemented on various targets. For compatibility, make sure to initialize the whole structure to zero.

`where` Indicates which types of processing units are able to execute the codelet. `STARPU_CPU|STARPU_CUDA` for instance indicates that the codelet is implemented for both CPU cores and CUDA devices while `STARPU_GORDON` indicates that it is only available on Cell SPUs.

`cpu_func` (optional)
Is a function pointer to the CPU implementation of the codelet. Its prototype must be: `void cpu_func(void *buffers[], void *cl_arg)`. The first argument being the array of data managed by the data management library, and the second argument is a pointer to the argument passed from the `cl_arg` field of the `starpu_task` structure. The `cpu_func` field is ignored if `STARPU_CPU` does not appear in the `where` field, it must be non-null otherwise.

`cuda_func` (optional)
Is a function pointer to the CUDA implementation of the codelet. *This must be a host-function written in the CUDA runtime API.* Its prototype must be: `void cuda_func(void *buffers[], void *cl_arg);`. The `cuda_func` field is ignored if `STARPU_CUDA` does not appear in the `where` field, it must be non-null otherwise.

`opencl_func` (optional)
: Is a function pointer to the OpenCL implementation of the codelet. Its prototype must be: `void opencl_func(starpu_data_interface_t *descr, void *arg);`. This pointer is ignored if `STARPU_OPENCL` does not appear in the `where` field, it must be non-null otherwise.

`gordon_func` (optional)
: This is the index of the Cell SPU implementation within the Gordon library. See Gordon documentation for more details on how to register a kernel and retrieve its index.

`nbuffers`
: Specifies the number of arguments taken by the codelet. These arguments are managed by the DSM and are accessed from the `void *buffers[]` array. The constant argument passed with the `cl_arg` field of the `starpu_task` structure is not counted in this number. This value should not be above `STARPU_NMAXBUFS`.

`model` (optional)
: This is a pointer to the task duration performance model associated to this codelet. This optional field is ignored when set to `NULL`.
  TODO

`power_model` (optional)
: This is a pointer to the task power consumption performance model associated to this codelet. This optional field is ignored when set to `NULL`. In the case of parallel codelets, this has to account for all processing units involved in the parallel execution.
  TODO

`struct starpu_task` [Data Type]

The `starpu_task` structure describes a task that can be offloaded on the various processing units managed by StarPU. It instantiates a codelet. It can either be allocated dynamically with the `starpu_task_create` method, or declared statically. In the latter case, the programmer has to zero the `starpu_task` structure and to fill the different fields properly. The indicated default values correspond to the configuration of a task allocated with `starpu_task_create`.

`cl`
: Is a pointer to the corresponding `starpu_codelet` data structure. This describes where the kernel should be executed, and supplies the appropriate implementations. When set to `NULL`, no code is executed during the tasks, such empty tasks can be useful for synchronization purposes.

`buffers`
: Is an array of `starpu_buffer_descr_t` structures. It describes the different pieces of data accessed by the task, and how they should be accessed. The `starpu_buffer_descr_t` structure is composed of two fields, the `handle` field specifies the handle of the piece of data, and the `mode` field is the required access mode (eg `STARPU_RW`). The number of entries in this array must be specified in the `nbuffers` field of the `starpu_codelet` structure, and should not excede `STARPU_NMAXBUFS`. If unsufficient, this value can be set with the `--enable-maxbuffers` option when configuring StarPU.

`cl_arg` (optional; default: `NULL`)
: This pointer is passed to the codelet through the second argument of the codelet implementation (e.g. `cpu_func` or `cuda_func`). In the specific case of the Cell processor, see the `cl_arg_size` argument.

`cl_arg_size` (optional, Cell-specific)
: In the case of the Cell processor, the `cl_arg` pointer is not directly given to the SPU function. A buffer of size `cl_arg_size` is allocated on the SPU. This buffer is then filled with the `cl_arg_size` bytes starting at address `cl_arg`. In this case, the argument given to the SPU codelet is therefore not the `cl_arg` pointer, but the address of the buffer in local store (LS) instead. This field is ignored for CPU, CUDA and OpenCL codelets, where the `cl_arg` pointer is given as such.

`callback_func` (optional) (default: `NULL`)
: This is a function pointer of prototype `void (*f)(void *)` which specifies a possible callback. If this pointer is non-null, the callback function is executed *on the host* after the execution of the task. The callback is passed the value contained in the `callback_arg` field. No callback is executed if the field is set to `NULL`.

`callback_arg` (optional) (default: `NULL`)
: This is the pointer passed to the callback function. This field is ignored if the `callback_func` is set to `NULL`.

`use_tag` (optional) (default: `0`)
: If set, this flag indicates that the task should be associated with the tag contained in the `tag_id` field. Tag allow the application to synchronize with the task and to express task dependencies easily.

`tag_id`
: This fields contains the tag associated to the task if the `use_tag` field was set, it is ignored otherwise.

`synchronous`
: If this flag is set, the `starpu_task_submit` function is blocking and returns only when the task has been executed (or if no worker is able to process the task). Otherwise, `starpu_task_submit` returns immediately.

`priority` (optional) (default: `STARPU_DEFAULT_PRIO`)
: This field indicates a level of priority for the task. This is an integer value that must be set between the return values of the `starpu_sched_get_min_priority` function for the least important tasks, and that of the `starpu_sched_get_max_priority` for the most important tasks (included). The `STARPU_MIN_PRIO` and `STARPU_MAX_PRIO` macros are provided for convenience and respectively returns value of `starpu_sched_get_min_priority` and `starpu_sched_get_max_priority`. Default priority is `STARPU_DEFAULT_PRIO`, which is always defined as 0 in order to allow static task initialization. Scheduling strategies that take priorities into account can use this parameter to take better scheduling decisions, but the scheduling policy may also ignore it.

`execute_on_a_specific_worker` (default: `0`)
: If this flag is set, StarPU will bypass the scheduler and directly affect this task to the worker specified by the `workerid` field.

`workerid` (optional)
: If the `execute_on_a_specific_worker` field is set, this field indicates which is the identifier of the worker that should process this task (as returned by `starpu_worker_get_id`). This field is ignored if `execute_on_a_specific_worker` field is set to 0.

`detach` (optional) (default: `1`)
: If this flag is set, it is not possible to synchronize with the task by the means of `starpu_task_wait` later on. Internal data structures are only guaranteed to be freed once `starpu_task_wait` is called if the flag is not set.

`destroy` (optional) (default: `1`)
: If this flag is set, the task structure will automatically be freed, either after the execution of the callback if the task is detached, or during `starpu_task_wait` otherwise. If this flag is not set, dynamically allocated data structures will not be freed until `starpu_task_destroy` is called explicitly. Setting this flag for a statically allocated task structure will result in undefined behaviour.

`predicted` (output field)
: Predicted duration of the task. This field is only set if the scheduling strategy used performance models.

`void starpu_task_init (`*struct starpu_task *`**`task`**`)` [Function]
: Initialize *task* with default values. This function is implicitly called by `starpu_task_create`. By default, tasks initialized with `starpu_task_init` must be deinitialized explicitly with `starpu_task_deinit`. Tasks can also be initialized statically, using the constant `STARPU_TASK_INITIALIZER`.

`struct starpu_task * starpu_task_create (`*void*`)` [Function]
: Allocate a task structure and initialize it with default values. Tasks allocated dynamically with `starpu_task_create` are automatically freed when the task is terminated. If the destroy flag is explicitly unset, the resources used by the task are freed by calling `starpu_task_destroy`.

`void starpu_task_deinit (`*struct starpu_task *`**`task`**`)` [Function]
: Release all the structures automatically allocated to execute *task*. This is called automatically by `starpu_task_destroy`, but the task structure itself is not freed. This should be used for statically allocated tasks for instance.

`void starpu_task_destroy (`*struct starpu_task *`**`task`**`)` [Function]
: Free the resource allocated during `starpu_task_create` and associated with *task*. This function can be called automatically after the execution of a task by setting the `destroy` flag of the `starpu_task` structure (default behaviour). Calling this function on a statically allocated task results in an undefined behaviour.

`int starpu_task_wait` (*struct starpu_task *task*) [Function]
This function blocks until *task* has been executed. It is not possible to synchronize with a task more than once. It is not possible to wait for synchronous or detached tasks.

Upon successful completion, this function returns 0. Otherwise, `-EINVAL` indicates that the specified task was either synchronous or detached.

`int starpu_task_submit` (*struct starpu_task *task*) [Function]
This function submits *task* to StarPU. Calling this function does not mean that the task will be executed immediately as there can be data or task (tag) dependencies that are not fulfilled yet: StarPU will take care of scheduling this task with respect to such dependencies. This function returns immediately if the `synchronous` field of the `starpu_task` structure was set to 0, and block until the termination of the task otherwise. It is also possible to synchronize the application with asynchronous tasks by the means of tags, using the `starpu_tag_wait` function for instance.

In case of success, this function returns 0, a return value of `-ENODEV` means that there is no worker able to process this task (e.g. there is no GPU available and this task is only implemented for CUDA devices).

`int starpu_task_wait_for_all` (*void*) [Function]
This function blocks until all the tasks that were submitted are terminated.

`struct starpu_task * starpu_get_current_task` (*void*) [Function]
This function returns the task currently executed by the worker, or NULL if it is called either from a thread that is not a task or simply because there is no task being executed at the moment.

`void starpu_display_codelet_stats` (*struct starpu_codelet_t *cl*) [Function]
Output on `stderr` some statistics on the codelet *cl*.

9.7 Explicit Dependencies

9.7.1 `starpu_task_declare_deps_array` – Declare task dependencies

`void starpu_task_declare_deps_array` (*struct starpu_task *task*, *unsigned ndeps*, *struct starpu_task *task_array*[]) [Function]
Declare task dependencies between a *task* and an array of tasks of length *ndeps*. This function must be called prior to the submission of the task, but it may called after the submission or the execution of the tasks in the array provided the tasks are still valid (ie. they were not automatically destroyed). Calling this function on a task that was already submitted or with an entry of *task_array* that is not a valid task anymore results in an undefined behaviour. If *ndeps* is null, no dependency is added. It is possible to call `starpu_task_declare_deps_array` multiple times on the same task, in this case, the dependencies are added. It is possible to have redundancy in the task dependencies.

9.7.2 starpu_tag_t – Task logical identifier

Description:

It is possible to associate a task with a unique "tag" chosen by the application, and to express dependencies between tasks by the means of those tags. To do so, fill the `tag_id` field of the `starpu_task` structure with a tag number (can be arbitrary) and set the `use_tag` field to 1.

If `starpu_tag_declare_deps` is called with this tag number, the task will not be started until the tasks which holds the declared dependency tags are completed.

9.7.3 starpu_tag_declare_deps – Declare the Dependencies of a Tag

Description:

Specify the dependencies of the task identified by tag `id`. The first argument specifies the tag which is configured, the second argument gives the number of tag(s) on which `id` depends. The following arguments are the tags which have to be terminated to unlock the task.

This function must be called before the associated task is submitted to StarPU with `starpu_task_submit`.

Remark Because of the variable arity of `starpu_tag_declare_deps`, note that the last arguments *must* be of type `starpu_tag_t`: constant values typically need to be explicitly casted. Using the `starpu_tag_declare_deps_array` function avoids this hazard.

Prototype: `void starpu_tag_declare_deps(starpu_tag_t id, unsigned ndeps, ...);`

Example:

```
/*  Tag 0x1 depends on tags 0x32 and 0x52 */
starpu_tag_declare_deps((starpu_tag_t)0x1,
        2, (starpu_tag_t)0x32, (starpu_tag_t)0x52);
```

9.7.4 starpu_tag_declare_deps_array – Declare the Dependencies of a Tag

Description:

This function is similar to `starpu_tag_declare_deps`, except that its does not take a variable number of arguments but an array of tags of size `ndeps`.

Prototype: `void starpu_tag_declare_deps_array(starpu_tag_t id, unsigned ndeps, starpu_tag_t *array);`

Example:

```
/*  Tag 0x1 depends on tags 0x32 and 0x52 */
starpu_tag_t tag_array[2] = {0x32, 0x52};
starpu_tag_declare_deps_array((starpu_tag_t)0x1, 2, tag_array);
```

9.7.5 starpu_tag_wait – Block until a Tag is terminated

`void starpu_tag_wait` (*starpu_tag_t* `id`) [Function]

This function blocks until the task associated to tag *id* has been executed. This is a blocking call which must therefore not be called within tasks or callbacks, but only from the application directly. It is possible to synchronize with the same tag multiple times, as long as the `starpu_tag_remove` function is not called. Note that it is still possible to synchronize with a tag associated to a task which `starpu_task` data structure was freed (e.g. if the `destroy` flag of the `starpu_task` was enabled).

9.7.6 starpu_tag_wait_array – Block until a set of Tags is terminated

`void starpu_tag_wait_array` (*unsigned* `ntags`, *starpu_tag_t* *`id`) [Function]

This function is similar to `starpu_tag_wait` except that it blocks until *all* the *ntags* tags contained in the *id* array are terminated.

9.7.7 starpu_tag_remove – Destroy a Tag

`void starpu_tag_remove` (*starpu_tag_t* `id`) [Function]

This function releases the resources associated to tag *id*. It can be called once the corresponding task has been executed and when there is no other tag that depend on this tag anymore.

9.7.8 starpu_tag_notify_from_apps – Feed a Tag explicitly

`void starpu_tag_notify_from_apps` (*starpu_tag_t* `id`) [Function]

This function explicitly unlocks tag *id*. It may be useful in the case of applications which execute part of their computation outside StarPU tasks (e.g. third-party libraries). It is also provided as a convenient tool for the programmer, for instance to entirely construct the task DAG before actually giving StarPU the opportunity to execute the tasks.

9.8 Implicit Data Dependencies

In this section, we describe how StarPU makes it possible to insert implicit task dependencies in order to enforce sequential data consistency. When this data consistency is enabled on a specific data handle, any data access will appear as sequentially consistent from the application. For instance, if the application submits two tasks that access the same piece of data in read-only mode, and then a third task that access it in write mode, dependencies will be added between the two first tasks and the third one. Implicit data dependencies are also inserted in the case of data accesses from the application.

9.8.1 starpu_data_set_default_sequential_consistency_flag – Set default sequential consistency flag

`void starpu_data_set_default_sequential_consistency_flag (unsigned flag)` [Function]

Set the default sequential consistency flag. If a non-zero value is passed, a sequential data consistency will be enforced for all handles registered after this function call, otherwise it is disabled. By default, StarPU enables sequential data consistency. It is also possible to select the data consistency mode of a specific data handle with the `starpu_data_set_sequential_consistency_flag` function.

9.8.2 starpu_data_get_default_sequential_consistency_flag – Get current default sequential consistency flag

`unsigned starpu_data_set_default_sequential_consistency_flag (void)` [Function]

This function returns the current default sequential consistency flag.

9.8.3 starpu_data_set_sequential_consistency_flag – Set data sequential consistency mode

`void starpu_data_set_sequential_consistency_flag (starpu_data_handle handle, unsigned flag)` [Function]

Select the data consistency mode associated to a data handle. The consistency mode set using this function has the priority over the default mode which can be set with `starpu_data_set_sequential_consistency_flag`.

9.9 Performance Model API

9.9.1 starpu_load_history_debug

`int starpu_load_history_debug (const char *symbol, struct starpu_perfmodel_t *model)` [Function]

TODO

9.9.2 starpu_perfmodel_debugfilepath

`void starpu_perfmodel_debugfilepath (struct starpu_perfmodel_t *model, enum starpu_perf_archtype arch, char *path, size_t maxlen)` [Function]

TODO

9.9.3 starpu_perfmodel_get_arch_name

`void starpu_perfmodel_get_arch_name (enum starpu_perf_archtype arch, char *archname, size_t maxlen)` [Function]

TODO

9.9.4 starpu_force_bus_sampling

`void starpu_force_bus_sampling (void)` [Function]

This forces sampling the bus performance model again.

9.10 Profiling API

9.10.1 starpu_profiling_status_set – Set current profiling status

Description:

Thie function sets the profiling status. Profiling is activated by passing `STARPU_PROFILING_ENABLE` in `status`. Passing `STARPU_PROFILING_DISABLE` disables profiling. Calling this function resets all profiling measurements. When profiling is enabled, the `profiling_info` field of the `struct starpu_task` structure points to a valid `struct starpu_task_profiling_info` structure containing information about the execution of the task.

Return value:

Negative return values indicate an error, otherwise the previous status is returned.

Prototype: `int starpu_profiling_status_set(int status);`

9.10.2 starpu_profiling_status_get – Get current profiling status

`int starpu_profiling_status_get` (*void*) [Function]

Return the current profiling status or a negative value in case there was an error.

9.10.3 struct starpu_task_profiling_info – Task profiling information

Description:

This structure contains information about the execution of a task. It is accessible from the `.profiling_info` field of the `starpu_task` structure if profiling was enabled.

Fields:

`submit_time`:
Date of task submission (relative to the initialization of StarPU).

`start_time`:
Date of task execution beginning (relative to the initialization of StarPU).

`end_time`: Date of task execution termination (relative to the initialization of StarPU).

`workerid`: Identifier of the worker which has executed the task.

9.10.4 struct starpu_worker_profiling_info – Worker profiling information

Description:

This structure contains the profiling information associated to a worker.

Fields:

`start_time`:
Starting date for the reported profiling measurements.

`total_time`:
Duration of the profiling measurement interval.

`executing_time`:
Time spent by the worker to execute tasks during the profiling measurement interval.

`sleeping_time`:
Time spent idling by the worker during the profiling measurement interval.

`executed_tasks`:
Number of tasks executed by the worker during the profiling measurement interval.

9.10.5 `starpu_worker_get_profiling_info` – Get worker profiling info

Description:
Get the profiling info associated to the worker identified by `workerid`, and reset the profiling measurements. If the `worker_info` argument is NULL, only reset the counters associated to worker `workerid`.

Return value:
Upon successful completion, this function returns 0. Otherwise, a negative value is returned.

Prototype: `int starpu_worker_get_profiling_info(int workerid, struct starpu_worker_profiling_info *worker_info);`

9.10.6 `struct starpu_bus_profiling_info` – Bus profiling information

Description:
TODO

Fields:

`start_time`:
TODO

`total_time`:
TODO

`transferred_bytes`:
TODO

`transfer_count`:
TODO

9.10.7 `starpu_bus_get_count`

`int starpu_bus_get_count (`*void*`)` [Function]
TODO

9.10.8 starpu_bus_get_id

`int starpu_bus_get_id (int src, int dst)` [Function]
TODO

9.10.9 starpu_bus_get_src

`int starpu_bus_get_src (int busid)` [Function]
TODO

9.10.10 starpu_bus_get_dst

`int starpu_bus_get_dst (int busid)` [Function]
TODO

9.10.11 starpu_timing_timespec_delay_us

`double starpu_timing_timespec_delay_us (struct timespec *start, struct timespec *end)` [Function]
TODO

9.10.12 starpu_timing_timespec_to_us

`double starpu_timing_timespec_to_us (struct timespec *ts)` [Function]
TODO

9.10.13 starpu_bus_profiling_helper_display_summary

`void starpu_bus_profiling_helper_display_summary (void)` [Function]
TODO

9.10.14 starpu_worker_profiling_helper_display_summary

`void starpu_worker_profiling_helper_display_summary (void)` [Function]
TODO

9.11 CUDA extensions

9.11.1 starpu_cuda_get_local_stream – Get current worker's CUDA stream

`cudaStream_t * starpu_cuda_get_local_stream (void)` [Function]
StarPU provides a stream for every CUDA device controlled by StarPU. This function is only provided for convenience so that programmers can easily use asynchronous operations within codelets without having to create a stream by hand. Note that the application is not forced to use the stream provided by `starpu_cuda_get_local_stream` and may also create its own streams. Synchronizing with `cudaThreadSynchronize()` is allowed, but will reduce the likelihood of having all transfers overlapped.

9.11.2 starpu_helper_cublas_init – Initialize CUBLAS on every CUDA device

`void starpu_helper_cublas_init (void)` [Function]

The CUBLAS library must be initialized prior to any CUBLAS call. Calling `starpu_helper_cublas_init` will initialize CUBLAS on every CUDA device controlled by StarPU. This call blocks until CUBLAS has been properly initialized on every device.

9.11.3 starpu_helper_cublas_shutdown – Deinitialize CUBLAS on every CUDA device

`void starpu_helper_cublas_shutdown (void)` [Function]

This function synchronously deinitializes the CUBLAS library on every CUDA device.

9.12 OpenCL extensions

9.12.1 Enabling OpenCL

On GPU devices which can run both CUDA and OpenCL, CUDA will be enabled by default. To enable OpenCL, you need either to disable CUDA when configuring StarPU:

```
% ./configure --disable-cuda
```

or when running applications:

```
% STARPU_NCUDA=0 ./application
```

OpenCL will automatically be started on any device not yet used by CUDA. So on a machine running 4 GPUS, it is therefore possible to enable CUDA on 2 devices, and OpenCL on the 2 other devices by doing so:

```
% STARPU_NCUDA=2 ./application
```

9.12.2 Compiling OpenCL kernels

Source codes for OpenCL kernels can be stored in a file or in a string. StarPU provides functions to build the program executable for each available OpenCL device as a `cl_program` object. This program executable can then be loaded within a specific queue as explained in the next section. These are only helpers, Applications can also fill a `starpu_opencl_program` array by hand for more advanced use (e.g. different programs on the different OpenCL devices, for relocation purpose for instance).

9.12.2.1 starpu_opencl_load_opencl_from_file – Compiling OpenCL source code

`int starpu_opencl_load_opencl_from_file (char *source_file_name, struct starpu_opencl_program *opencl_programs, const char* build_options)` [Function]

TODO

9.12.2.2 starpu_opencl_load_opencl_from_string – Compiling OpenCL source code

`int starpu_opencl_load_opencl_from_string (char *opencl_program_source, struct starpu_opencl_program *opencl_programs, const char* build_options)` [Function]

TODO

9.12.2.3 starpu_opencl_unload_opencl – Releasing OpenCL code

`int starpu_opencl_unload_opencl (struct starpu_opencl_program *opencl_programs)` [Function]

TODO

9.12.3 Loading OpenCL kernels

9.12.3.1 starpu_opencl_load_kernel – Loading a kernel

`int starpu_opencl_load_kernel (cl_kernel *kernel, cl_command_queue *queue, struct starpu_opencl_program *opencl_programs, char *kernel_name, int devid)` [Function]

TODO

9.12.3.2 starpu_opencl_release_kernel – Releasing a kernel

`int starpu_opencl_release_kernel (cl_kernel kernel)` [Function]

TODO

9.12.4 OpenCL statistics

9.12.4.1 starpu_opencl_collect_stats – Collect statistics on a kernel execution

`int starpu_opencl_collect_stats (cl_event event)` [Function]

After termination of the kernels, the OpenCL codelet should call this function to pass it the even returned by `clEnqueueNDRangeKernel`, to let StarPU collect statistics about the kernel execution (used cycles, consumed power).

9.13 Cell extensions

nothing yet.

9.14 Miscellaneous helpers

9.14.1 starpu_data_cpy – Copy a data handle into another data handle

`int starpu_data_cpy (starpu_data_handle dst_handle, starpu_data_handle src_handle, int asynchronous, void (*callback_func)(void*), void *callback_arg)` [Function]

Copy the content of the *src_handle* into the *dst_handle* handle. The *asynchronous* parameter indicates whether the function should block or not. In the case of an

asynchronous call, it is possible to synchronize with the termination of this operation either by the means of implicit dependencies (if enabled) or by calling `starpu_task_wait_for_all()`. If *callback_func* is not `NULL`, this callback function is executed after the handle has been copied, and it is given the *callback_arg* pointer as argument.

9.14.2 starpu_execute_on_each_worker – Execute a function on a subset of workers

`void starpu_execute_on_each_worker (void (*func)(void *), void *arg, uint32_t where)` [Function]

When calling this method, the offloaded function specified by the first argument is executed by every StarPU worker that may execute the function. The second argument is passed to the offloaded function. The last argument specifies on which types of processing units the function should be executed. Similarly to the *where* field of the `starpu_codelet` structure, it is possible to specify that the function should be executed on every CUDA device and every CPU by passing `STARPU_CPU|STARPU_CUDA`. This function blocks until the function has been executed on every appropriate processing units, so that it may not be called from a callback function for instance.

10 Advanced Topics

10.1 Defining a new data interface

10.1.1 `struct starpu_data_interface_ops_t` – Per-interface methods

Description:
TODO describe all the different fields

10.1.2 `struct starpu_data_copy_methods` – Per-interface data transfer methods

Description:
TODO describe all the different fields

10.1.3 An example of data interface

TODO See `src/datawizard/interfaces/vector_interface.c` for now.

10.2 Defining a new scheduling policy

TODO

A full example showing how to define a new scheduling policy is available in the StarPU sources in the directory `examples/scheduler/`.

10.2.1 `struct starpu_sched_policy_s` – Scheduler methods

Description:
This structure contains all the methods that implement a scheduling policy. An application may specify which scheduling strategy in the `sched_policy` field of the `starpu_conf` structure passed to the `starpu_init` function.

Fields:

`init_sched`:
Initialize the scheduling policy.

`deinit_sched`:
Cleanup the scheduling policy.

`push_task`:
Insert a task into the scheduler.

`push_prio_task`:
Insert a priority task into the scheduler.

`push_prio_notify`:
Notify the scheduler that a task was pushed on the worker. This method is called when a task that was explicitely assigned to a worker is scheduled. This method therefore permits to keep the state of of the scheduler coherent even when StarPU bypasses the scheduling strategy.

`pop_task`: Get a task from the scheduler. The mutex associated to the worker is already taken when this method is called. If this method is defined as `NULL`, the worker will only execute tasks from its local queue. In this case, the `push_task` method should use the `starpu_push_local_task` method to assign tasks to the different workers.

`pop_every_task`:
Remove all available tasks from the scheduler (tasks are chained by the means of the prev and next fields of the starpu_task structure). The mutex associated to the worker is already taken when this method is called.

`post_exec_hook` (optionnal):
This method is called every time a task has been executed.

`policy_name`:
Name of the policy (optionnal).

`policy_description`:
Description of the policy (optionnal).

10.2.2 `starpu_worker_set_sched_condition` – Specify the condition variable associated to a worker

`void starpu_worker_set_sched_condition (`*int* ***workerid***, *pthread_cond_t* ***sched_cond***, *pthread_mutex_t* ***sched_mutex***`)` [Function]

When there is no available task for a worker, StarPU blocks this worker on a condition variable. This function specifies which condition variable (and the associated mutex) should be used to block (and to wake up) a worker. Note that multiple workers may use the same condition variable. For instance, in the case of a scheduling strategy with a single task queue, the same condition variable would be used to block and wake up all workers. The initialization method of a scheduling strategy (`init_sched`) must call this function once per worker.

10.2.3 `starpu_sched_set_min_priority`

`void starpu_sched_set_min_priority (`*int* ***min_prio***`)` [Function]

Defines the minimum priority level supported by the scheduling policy. The default minimum priority level is the same as the default priority level which is 0 by convention. The application may access that value by calling the `starpu_sched_get_min_priority` function. This function should only be called from the initialization method of the scheduling policy, and should not be used directly from the application.

10.2.4 `starpu_sched_set_max_priority`

`void starpu_sched_set_min_priority (`*int* ***max_prio***`)` [Function]

Defines the maximum priority level supported by the scheduling policy. The default maximum priority level is 1. The application may access that value by calling the `starpu_sched_get_max_priority` function. This function should only be called from the initialization method of the scheduling policy, and should not be used directly from the application.

10.2.5 starpu_push_local_task

int starpu_push_local_task (*int* ***workerid***, *struct starpu_task* ***task***, *int* ***back***) [Function]

The scheduling policy may put tasks directly into a worker's local queue so that it is not always necessary to create its own queue when the local queue is sufficient. If "back" not null, the task is put at the back of the queue where the worker will pop tasks first. Setting "back" to 0 therefore ensures a FIFO ordering.

10.2.6 Source code

```
static struct starpu_sched_policy_s dummy_sched_policy = {
    .init_sched = init_dummy_sched,
    .deinit_sched = deinit_dummy_sched,
    .push_task = push_task_dummy,
    .push_prio_task = NULL,
    .pop_task = pop_task_dummy,
    .post_exec_hook = NULL,
    .pop_every_task = NULL,
    .policy_name = "dummy",
    .policy_description = "dummy scheduling strategy"
};
```

Appendix A Full source code for the 'Scaling a Vector' example

A.1 Main application

```
/*
 * This example demonstrates how to use StarPU to scale an array by a factor.
 * It shows how to manipulate data with StarPU's data management library.
 *  1- how to declare a piece of data to StarPU (starpu_vector_data_register)
 *  2- how to describe which data are accessed by a task (task->buffers[0])
 *  3- how a kernel can manipulate the data (buffers[0].vector.ptr)
 */
#include <starpu.h>
#include <starpu_opencl.h>

#define    NX    2048

extern void scal_cpu_func(void *buffers[], void *_args);
extern void scal_cuda_func(void *buffers[], void *_args);
extern void scal_opencl_func(void *buffers[], void *_args);

static starpu_codelet cl = {
    .where = STARPU_CPU | STARPU_CUDA | STARPU_OPENCL,
    /* CPU implementation of the codelet */
    .cpu_func = scal_cpu_func,
#ifdef STARPU_USE_CUDA
    /* CUDA implementation of the codelet */
    .cuda_func = scal_cuda_func,
#endif
#ifdef STARPU_USE_OPENCL
    /* OpenCL implementation of the codelet */
    .opencl_func = scal_opencl_func,
#endif
    .nbuffers = 1
};

#ifdef STARPU_USE_OPENCL
struct starpu_opencl_program programs;
#endif

int main(int argc, char **argv)
{
    /* We consider a vector of float that is initialized just as any of C
      * data */
    float vector[NX];
    unsigned i;
    for (i = 0; i < NX; i++)
        vector[i] = 1.0f;

    fprintf(stderr, "BEFORE : First element was %f\n", vector[0]);

    /* Initialize StarPU with default configuration */
    starpu_init(NULL);

#ifdef STARPU_USE_OPENCL
        starpu_opencl_load_opencl_from_file(
               "examples/basic_examples/vector_scal_opencl_kernel.cl", &programs, NULL);
#endif
```

```
    /* Tell StaPU to associate the "vector" vector with the "vector_handle"
     * identifier. When a task needs to access a piece of data, it should
     * refer to the handle that is associated to it.
     * In the case of the "vector" data interface:
     *  - the first argument of the registration method is a pointer to the
     *    handle that should describe the data
     *  - the second argument is the memory node where the data (ie. "vector")
     *    resides initially: 0 stands for an address in main memory, as
     *    opposed to an adress on a GPU for instance.
     *  - the third argument is the adress of the vector in RAM
     *  - the fourth argument is the number of elements in the vector
     *  - the fifth argument is the size of each element.
     */
    starpu_data_handle vector_handle;
    starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector,
                                NX, sizeof(vector[0]));

    float factor = 3.14;

    /* create a synchronous task: any call to starpu_task_submit will block
      * until it is terminated */
    struct starpu_task *task = starpu_task_create();
    task->synchronous = 1;

    task->cl = &cl;

    /* the codelet manipulates one buffer in RW mode */
    task->buffers[0].handle = vector_handle;
    task->buffers[0].mode = STARPU_RW;

    /* an argument is passed to the codelet, beware that this is a
     * READ-ONLY buffer and that the codelet may be given a pointer to a
     * COPY of the argument */
    task->cl_arg = &factor;
    task->cl_arg_size = sizeof(factor);

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

    /* StarPU does not need to manipulate the array anymore so we can stop
      * monitoring it */
    starpu_data_unregister(vector_handle);

#ifdef STARPU_USE_OPENCL
    starpu_opencl_unload_opencl(&programs);
#endif

    /* terminate StarPU, no task can be submitted after */
    starpu_shutdown();

    fprintf(stderr, "AFTER First element is %f\n", vector[0]);

    return 0;
}
```

A.2 CPU Kernel

```
#include <starpu.h>

/* This kernel takes a buffer and scales it by a constant factor */
void scal_cpu_func(void *buffers[], void *cl_arg)
{
    unsigned i;
    float *factor = cl_arg;

    /*
     * The "buffers" array matches the task->buffers array: for instance
     * task->buffers[0].handle is a handle that corresponds to a data with
     * vector "interface", so that the first entry of the array in the
     * codelet  is a pointer to a structure describing such a vector (ie.
     * struct starpu_vector_interface_s *). Here, we therefore manipulate
     * the buffers[0] element as a vector: nx gives the number of elements
     * in the array, ptr gives the location of the array (that was possibly
     * migrated/replicated), and elemsize gives the size of each elements.
     */
    starpu_vector_interface_t *vector = buffers[0];

    /* length of the vector */
    unsigned n = STARPU_VECTOR_GET_NX(vector);

    /* get a pointer to the local copy of the vector : note that we have to
     * cast it in (float *) since a vector could contain any type of
     * elements so that the .ptr field is actually a uintptr_t */
    float *val = (float *)STARPU_VECTOR_GET_PTR(vector);

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

A.3 CUDA Kernel

```
#include <starpu.h>

static __global__ void vector_mult_cuda(float *val, unsigned n,
                                        float factor)
{
        unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n)
               val[i] *= factor;
}

extern "C" void scal_cuda_func(void *buffers[], void *_args)
{
        float *factor = (float *)_args;

        /* length of the vector */
        unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
        /* local copy of the vector pointer */
        float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
        unsigned threads_per_block = 64;
        unsigned nblocks = (n + threads_per_block-1) / threads_per_block;

        vector_mult_cuda<<<nblocks,threads_per_block, 0, starpu_cuda_get_local_stream()>>>(val, n, *facto
```

```
        cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

A.4 OpenCL Kernel

A.4.1 Invoking the kernel

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

extern struct starpu_opencl_program programs;

void scal_opencl_func(void *buffers[], void *_args)
{
    float *factor = _args;
    int id, devid, err;
    cl_kernel kernel;
    cl_command_queue queue;
    cl_event event;

    /* length of the vector */
    unsigned n = STARPU_VECTOR_GET_NX(buffers[0]);
    /* OpenCL copy of the vector pointer */
    cl_mem val = (cl_mem)STARPU_VECTOR_GET_PTR(buffers[0]);

    id = starpu_worker_get_id();
    devid = starpu_worker_get_devid(id);

    err = starpu_opencl_load_kernel(&kernel, &queue, &programs, "vector_mult_opencl",
                                    devid);
    if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

    err = clSetKernelArg(kernel, 0, sizeof(val), &val);
    err |= clSetKernelArg(kernel, 1, sizeof(n), &n);
    err |= clSetKernelArg(kernel, 2, sizeof(*factor), factor);
    if (err) STARPU_OPENCL_REPORT_ERROR(err);

    {
        size_t global=n;
        size_t local;
        size_t s;
        cl_device_id device;

        starpu_opencl_get_device(devid, &device);
        err = clGetKernelWorkGroupInfo (kernel, device, CL_KERNEL_WORK_GROUP_SIZE,
                                        sizeof(local), &local, &s);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
        if (local > global) local=global;

        err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0,
                                     NULL, &event);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
    }

    clFinish(queue);
    starpu_opencl_collect_stats(event);
    clReleaseEvent(event);
```

```
    starpu_opencl_release_kernel(kernel);
}
```

A.4.2 Source of the kernel

```
__kernel void vector_mult_opencl(__global float* val, int nx, float factor)
{
        const int i = get_global_id(0);
        if (i < nx) {
                val[i] *= factor;
        }
}
```

Function Index