StarPU

Short Contents

Table of Contents

Preface

This manual documents the usage of StarPU.

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 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 StarPU primary data structure 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. In addition to the codelet that a task implements, it also describes 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).

A task may be identified by a unique 64-bit number which we refer as a **tag**. Task dependencies can be enforced either by the means of callback functions, or by expressing dependencies between tags.

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.

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 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`.
- You can check out the project's SVN repository through anonymous access. This will provide you with a read access to the repository.

  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
```

These steps require to run autoconf and automake to generate the `./configure` script. This can be done by calling `./autogen.sh`. The required version for autoconf is 2.60 or higher.

```
% ./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 4.1 [Compilation configuration], page 9.

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.

4 Configuring StarPU

4.1 Compilation configuration

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

4.1.1 Common configuration

4.1.1.1 `--enable-debug`

Description:
Enable debugging messages.

4.1.1.2 `--enable-fast`

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

4.1.1.3 `--enable-verbose`

Description:
Augment the verbosity of the debugging messages.

4.1.1.4 `--enable-coverage`

Description:
Enable flags for the coverage tool.

4.1.2 Configuring workers

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

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

4.1.2.2 `--disable-cpu`

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

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

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

4.1.2.4 `--disable-cuda`

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

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

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

4.1.2.6 --enable-maxopencldev=<number>

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

4.1.2.7 --disable-opencl

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

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

Description:
Specify the location of the OpenCL SDK. This directory should notably contain `include/CL/cl.h`.

4.1.2.9 --enable-gordon

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

4.1.2.10 --with-gordon-dir=<path>

Description:
Specify the location of the Gordon SDK.

4.1.3 Advanced configuration

4.1.3.1 --enable-perf-debug

Description:
Enable performance debugging.

4.1.3.2 --enable-model-debug

Description:
Enable performance model debugging.

4.1.3.3 --enable-stats

Description:
Enable statistics.

4.1.3.4 --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.

4.1.3.5 `--enable-allocation-cache`

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

4.1.3.6 `--enable-opengl-render`

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

4.1.3.7 `--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'.

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

Description:
Specify where magma is installed.

4.1.3.9 `--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`.

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

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

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

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

4.1.3.12 `--with-mpi`

Description:
Enable building libstarpumpi.

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

Description:
Specify the location of GotoBLAS.

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

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

4.2 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.

4.2.1 Configuring workers

4.2.1.1 `STARPU_NCPUS` – Number of CPU workers

Description:

Specify the maximum number of CPU workers. Note that StarPU will not allocate more CPUs than there are physical CPUs, and that some CPUs are used to control the accelerators.

4.2.1.2 `STARPU_NCUDA` – Number of CUDA workers

Description:

Specify the maximum 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.

4.2.1.3 `STARPU_NOPENCL` – Number of OpenCL workers

Description:

OpenCL equivalent of the `STARPU_NCUDA` environment variable.

4.2.1.4 `STARPU_NGORDON` – Number of SPU workers (Cell)

Description:

Specify the maximum number of SPUs that StarPU can use.

4.2.1.5 `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 = "1 3 0 2"`, the first worker will be bound to logical CPU #1, the second CPU worker will be bound to logical CPU #3 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.

4.2.1.6 `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.

4.2.1.7 `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.

4.2.2 Configuring the Scheduling engine

4.2.2.1 `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.

4.2.2.2 `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.

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

4.2.2.3 `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.

4.2.2.4 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.

4.2.2.5 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.

4.2.3 Miscellaneous and debug

4.2.3.1 STARPU_LOGFILENAME – Select debug file name

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

5 StarPU API

5.1 Initialization and Termination

5.1.1 `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);`

5.1.2 `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 maximum 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 maximum 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 maximum 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 maximum 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 4.2.1.5 [STARPU_WORKERS_CPUID], page 12 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 4.2.1.6 [STARPU_WORKERS_CUDAID], page 13 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.

5.1.3 starpu_shutdown – Terminate StarPU

Description:
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.

Prototype: `void starpu_shutdown(void);`

5.2 Workers' Properties

5.2.1 starpu_worker_get_count – Get the number of processing units

Description:
This function returns the number of workers (i.e. processing units executing StarPU tasks). The returned value should be at most `STARPU_NMAXWORKERS`.

Prototype: `unsigned starpu_worker_get_count(void);`

5.2.2 starpu_cpu_worker_get_count – Get the number of CPU controlled by StarPU

Description:
This function returns the number of CPUs controlled by StarPU. The returned value should be at most `STARPU_NMAXCPUS`.

Prototype: `unsigned starpu_cpu_worker_get_count(void);`

5.2.3 starpu_cuda_worker_get_count – Get the number of CUDA devices controlled by StarPU

Description:
This function returns the number of CUDA devices controlled by StarPU. The returned value should be at most `STARPU_MAXCUDADEVS`.

Prototype: `unsigned starpu_cuda_worker_get_count(void);`

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

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

Prototype: `unsigned starpu_opencl_worker_get_count(void);`

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

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

Prototype: `unsigned starpu_opencl_worker_get_count(void);`

5.2.6 starpu_worker_get_id – Get the identifier of the current worker

Description:
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`.

Prototype: `int starpu_worker_get_id(void);`

5.2.7 starpu_worker_get_devid – Get the device identifier of a worker

Description:
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.

Prototype: `int starpu_worker_get_devid(int id);`

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

Description:
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.

Prototype: `enum starpu_archtype starpu_worker_get_type(int id);`

5.2.9 starpu_worker_get_name – Get the name of a worker

Description:
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.

Prototype: `void starpu_worker_get_name(int id, char *dst, size_t maxlen);`

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

Description:
This function returns the identifier of the memory node associated to the worker identified by `workerid`.

Prototype: `unsigned starpu_worker_get_memory_node(unsigned workerid);`

5.3 Data Library

This section describes the data management facilities provided by StarPU.

We show how to use existing data interfaces in Section 5.4 [Data Interfaces], page 21, but developers can design their own data interfaces if required.

5.3.1 `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.

5.3.2 `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).

5.3.3 `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.

5.3.4 `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 5.4 [Data Interfaces], page 21.

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.

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

Description:

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 7.1.1 [struct starpu_data_interface_ops_t], page 53 for more details on this structure.

If `home_node` is not a valid memory node, StarPU will automatically allocate the memory described by the interface the data handle 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`).

Prototype:

```
void starpu_data_register(starpu_data_handle *handleptr, uint32_t
home_node, void *interface, struct starpu_data_interface_ops_t
*ops);
```

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

Description:

This function unregisters a data handle from StarPU. If the data was automatically allocated by StarPU because the home node was not valid, 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.

Prototype:

```
void starpu_data_unregister(starpu_data_handle handle);
```

5.3.7 `starpu_data_invalidate` – Invalidate all data replicates

Description:

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.

Prototype: `void starpu_data_invalidate(starpu_data_handle handle);`

5.3.8 starpu_data_acquire – Access registered data from the application

Description:

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` in case they are enabled. `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.

Prototype: `int starpu_data_acquire(starpu_data_handle handle, starpu_access_mode mode);`

5.3.9 starpu_data_acquire_cb – Access registered data from the application asynchronously

Description:

`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` 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.

Prototype: `int starpu_data_acquire_cb(starpu_data_handle handle, starpu_access_mode mode, void (*callback)(void *), void *arg);`

5.3.10 starpu_data_release – Release registered data from the application

Description:

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

Prototype: `void starpu_data_release(starpu_data_handle handle);`

5.4 Data Interfaces

5.4.1 Variable Interface

Description:
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));
```

5.4.2 Vector Interface

Description:
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]));
```

5.4.3 Matrix Interface

Description:
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));
```

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

Description:
Prototype: `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);`

Example:

```
|
```

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

Description:

Prototype: `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);`

Example:

```
|
```

5.4.6 Block Interface

Description:

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));
```

5.5 Data Partition

5.5.1 `struct starpu_data_filter` – StarPU filter structure

Description:

The filter structure describes a data partitioning function.

Fields:

`filter_func`:
TODO `void (*filter_func)(void *father_interface, void* child_interface, struct starpu_data_filter *, unsigned id, unsigned nparts);`

`get_nchildren`:
TODO `unsigned (*get_nchildren)(struct starpu_data_filter *, starpu_data_handle initial_handle);`

`get_child_ops`:
TODO `struct starpu_data_interface_ops_t *(*get_child_ops)(struct starpu_data_filter *, unsigned id);`

`filter_arg`:
TODO

`nchildren`:
TODO

`filter_arg_ptr`:
TODO

5.5.2 starpu_data_partition – Partition Data

Description:
TODO

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

5.5.3 starpu_data_unpartition – Unpartition data

Description:
TODO

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

5.5.4 starpu_data_get_nb_children

Description:
TODO

Return value:
This function returns returns the number of children.

Prototype: `int starpu_data_get_nb_children(starpu_data_handle handle);`

5.5.5 starpu_data_get_sub_data

Description:
TODO

Return value:
TODO

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

5.5.6 Predefined filter functions

This section gives a list of the predefined partitioning functions. Examples on how to use them are shown in Section 6.6 [Partitioning Data], page 51.

5.5.6.1 Partitioning BCSR Data

- TODO `void starpu_canonical_block_filter_bcsr(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`

- TODO `void starpu_vertical_block_filter_func_csr(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`

5.5.6.2 Partitioning BLAS interface

- TODO `void starpu_block_filter_func(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`
- TODO `void starpu_vertical_block_filter_func(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`

5.5.6.3 Partitioning Vector Data

- TODO `void starpu_block_filter_func_vector(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`
- TODO `void starpu_vector_list_filter_func(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`
- TODO `void starpu_vector_divide_in_2_filter_func(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`

5.5.6.4 Partitioning Block Data

- TODO `void starpu_block_filter_func_block(void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts);`

5.6 Codelets and Tasks

5.6.1 `struct starpu_codelet` – StarPU codelet structure

Description:

The codelet structure describes a kernel that is possibly implemented on various targets.

Fields:

`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 performance model associated to this codelet. This optional field is ignored when set to `NULL`. TODO

5.6.2 `struct starpu_task` – StarPU task structure

Description:
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`.

Fields:

`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.

`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 `STARPU_MIN_PRIO` (for the least important tasks) and `STARPU_MAX_PRIO` (for the most important tasks) included. Default priority is `STARPU_DEFAULT_PRIO`. 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.

5.6.3 `starpu_task_init` – Initialize a Task

Description:
Initialize a task structure 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`.

Prototype: `void starpu_task_init(struct starpu_task *task);`

5.6.4 `starpu_task_create` – Allocate and Initialize a Task

Description:
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`.

Prototype: `struct starpu_task *starpu_task_create(void);`

5.6.5 starpu_task_deinit – Release all the resources used by a Task

Description:
Release all the structures automatically allocated to execute the 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.

Prototype: `void starpu_task_deinit(struct starpu_task *task);`

5.6.6 starpu_task_destroy – Destroy a dynamically allocated Task

Description:
Free the resource allocated during `starpu_task_create`. 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.

Prototype: `void starpu_task_destroy(struct starpu_task *task);`

5.6.7 starpu_task_wait – Wait for the termination of a Task

Description:
This function blocks until the 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.

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

Prototype: `int starpu_task_wait(struct starpu_task *task);`

5.6.8 starpu_task_submit – Submit a Task

Description:
This function submits a 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.

Return value:
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).

Prototype: `int starpu_task_submit(struct starpu_task *task);`

5.6.9 starpu_task_wait_for_all – Wait for the termination of all Tasks

Description:
This function blocks until all the tasks that were submitted are terminated.

Prototype: `void starpu_task_wait_for_all(void);`

5.6.10 starpu_get_current_task – Return the task currently executed by the worker

Description:
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.

Prototype: `struct starpu_task *starpu_get_current_task(void);`

5.6.11 starpu_display_codelet_stats – Display statistics

Description:
TODO

Prototype: `void starpu_display_codelet_stats(struct starpu_codelet_t *cl);`

5.7 Explicit Dependencies

5.7.1 starpu_task_declare_deps_array – Declare task dependencies

Description:
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.

Prototype: `void starpu_task_declare_deps_array(struct starpu_task *task, unsigned ndeps, struct starpu_task *task_array[]);`

5.7.2 starpu_tag_t – Task logical identifier

Description:
It is possible to associate a task with a unique "tag" 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.

5.7.3 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);
```

5.7.4 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);
```

5.7.5 starpu_tag_wait – Block until a Tag is terminated

Description:

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).

Prototype: `void starpu_tag_wait(starpu_tag_t id);`

5.7.6 `starpu_tag_wait_array` – Block until a set of Tags is terminated

Description:

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

Prototype: `void starpu_tag_wait_array(unsigned ntags, starpu_tag_t *id);`

5.7.7 `starpu_tag_remove` – Destroy a Tag

Description:

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.

Prototype: `void starpu_tag_remove(starpu_tag_t id);`

5.7.8 `starpu_tag_notify_from_apps` – Feed a Tag explicitly

Description:

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.

Prototype: `void starpu_tag_notify_from_apps(starpu_tag_t id);`

5.8 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.

5.8.1 `starpu_data_set_default_sequential_consistency_flag` – Set default sequential consistency flag

Description:

Set the default sequential consistency flag. If a non-null 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.

Prototype: `void starpu_data_set_default_sequential_consistency_flag(unsigned flag);`

5.8.2 starpu_data_get_default_sequential_consistency_flag – Get current default sequential consistency flag

Description:
This function returns the current default sequential consistency flag.

Prototype: `unsigned starpu_data_set_default_sequential_consistency_flag(void);`

5.8.3 starpu_data_set_sequential_consistency_flag – Set data sequential consistency mode

Description:
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`.

Prototype: `void starpu_data_set_sequential_consistency_flag(starpu_data_handle handle, unsigned flag);`

5.9 Performance Model API

5.9.1 starpu_load_history_debug

Description:
TODO

Prototype: `int starpu_load_history_debug(const char *symbol, struct starpu_perfmodel_t *model);`

5.9.2 starpu_perfmodel_debugfilepath

Description:
TODO

Prototype: `void starpu_perfmodel_debugfilepath(struct starpu_perfmodel_t *model, enum starpu_perf_archtype arch, char *path, size_t maxlen);`

5.9.3 starpu_perfmodel_get_arch_name

Description:
TODO

Prototype: `void starpu_perfmodel_get_arch_name(enum starpu_perf_archtype arch, char *archname, size_t maxlen);`

5.9.4 starpu_force_bus_sampling

Description:
TODO

Prototype: `void starpu_force_bus_sampling(void);`

5.10 Profiling API

5.10.1 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);`

5.10.2 starpu_profiling_status_get – Get current profiling status

Description:

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

Prototype: `int starpu_profiling_status_get(void);`

5.10.3 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.

5.10.4 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.

5.10.5 `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);`

5.10.6 `struct starpu_bus_profiling_info` – Bus profiling information

Description:
TODO

Fields:

`start_time`:
TODO

`total_time`:
TODO

`transferred_bytes`:
TODO

`transfer_count`:
TODO

5.10.7 starpu_bus_get_count

Description:
TODO

Prototype: `int starpu_bus_get_count(void);`

5.10.8 starpu_bus_get_id

Description:
TODO

Prototype: `int starpu_bus_get_id(int src, int dst);`

5.10.9 starpu_bus_get_src

Description:
TODO

Prototype: `int starpu_bus_get_src(int busid);`

5.10.10 starpu_bus_get_dst

Description:
TODO

Prototype: `int starpu_bus_get_dst(int busid);`

5.10.11 starpu_timing_timespec_delay_us

Description:
TODO

Prototype: `double starpu_timing_timespec_delay_us(struct timespec *start, struct timespec *end);`

5.10.12 starpu_timing_timespec_to_us

Description:
TODO

Prototype: `double starpu_timing_timespec_to_us(struct timespec *ts);`

5.10.13 starpu_bus_profiling_helper_display_summary

Description:
TODO

Prototype: `void starpu_bus_profiling_helper_display_summary(void);`

5.11 CUDA extensions

5.11.1 starpu_cuda_get_local_stream – Get current worker's CUDA stream

Description:
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.

Prototype: `cudaStream_t *starpu_cuda_get_local_stream(void);`

5.11.2 starpu_helper_cublas_init – Initialize CUBLAS on every CUDA device

Description:
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.

Prototype: `void starpu_helper_cublas_init(void);`

5.11.3 starpu_helper_cublas_shutdown – Deinitialize CUBLAS on every CUDA device

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

Prototype: `void starpu_helper_cublas_shutdown(void);`

5.12 OpenCL extensions

5.12.1 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
```

5.12.2 Compiling OpenCL codelets

Source codes for OpenCL codelets 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).

5.12.2.1 `starpu_opencl_load_opencl_from_file` – Compiling OpenCL source code

Description:
TODO

Prototype: `int starpu_opencl_load_opencl_from_file(char *source_file_name, struct starpu_opencl_program *opencl_programs);`

5.12.2.2 `starpu_opencl_load_opencl_from_string` – Compiling OpenCL source code

Description:
TODO

Prototype: `int starpu_opencl_load_opencl_from_string(char *opencl_program_source, struct starpu_opencl_program *opencl_programs);`

5.12.2.3 `starpu_opencl_unload_opencl` – Releasing OpenCL code

Description:
TODO

Prototype: `int starpu_opencl_unload_opencl(struct starpu_opencl_program *opencl_programs);`

5.12.3 Loading OpenCL codelets

5.12.3.1 `starpu_opencl_load_kernel` – Loading a kernel

Description:
TODO

Prototype: `int starpu_opencl_load_kernel(cl_kernel *kernel, cl_command_queue *queue, struct starpu_opencl_program *opencl_programs, char *kernel_name, int devid)`

5.12.3.2 `starpu_opencl_release_kernel` – Releasing a kernel

Description:
TODO

Prototype: `int starpu_opencl_release_kernel(cl_kernel kernel);`

5.13 Cell extensions

nothing yet.

5.14 Miscellaneous helpers

5.14.1 `starpu_execute_on_each_worker` – Execute a function on a subset of workers

Description:

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.

Prototype: `void starpu_execute_on_each_worker(void (*func)(void *), void *arg, uint32_t where);`

6 Basic Examples

6.1 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)
```

6.2 Hello World

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

6.2.1 Required Headers

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

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

6.2.2 Defining a Codelet

```
void cpu_func(void *buffers[], void *cl_arg)
{
    float *array = cl_arg;

    printf("Hello world (array = {%f, %f} )\n", array[0], array[1]);
}

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.

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 5.6 [Codelets and Tasks], page 25 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). 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.

6.2.3 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 */

    float array[2] = {1.0f, -1.0f};
    task->cl_arg = &array;
    task->cl_arg_size = sizeof(array);

    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 5.6.4 [starpu_task_create], page 28), 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.

Once a task has been executed, an optional callback function can 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-null, 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.

6.2.4 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 (array = {1.000000, -1.000000} )
Callback function (arg 42)
```

6.3 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 55.

6.3.1 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**. By default, there are different 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 currently 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, the number of elements in the vector and the size of each element. It is possible 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 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]);
    /* local 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 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. 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 generality, 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.

6.3.2 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
```

6.4 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.

6.4.1 Definition of the CUDA Codelet

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.

```
#include <starpu.h>

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

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]);

    vector_mult_cuda<<<1,1>>>(val, n, *factor);

    cudaThreadSynchronize();
}
```

6.4.2 Definition of the OpenCL Codelet

The OpenCL implementation can be written as follows. StarPU provides tools to compile a OpenCL codelet 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;
        }
}
```

```
#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;

    /* 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]);

    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(cl_mem), &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, NULL);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
    }

    clFinish(queue);

    starpu_opencl_release_kernel(kernel);
}
```

6.4.3 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);
#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 data and shutdown StarPU */
    starpu_data_release(vector_handle);
    starpu_shutdown();

    return 0;
}
```

6.4.4 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 codelet 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 5.12.2.1 [starpu_opencl_load_opencl_from_file], page 38).

```
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:

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

6.5 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;

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);
}
```

6.6 Partitioning Data

```
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++) {
    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);
}
```

6.7 Performance model example

TODO

```
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,
    /* in case the scheduling policy may use performance models */
    .model = &mult_perf_model
};
```

6.8 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 6.2 [Hello World], page 41), vector/scalar product (as shown in Section 6.4 [Vector Scaling on an Hybrid CPU/GPU Machine], page 46), matrix product examples (as shown in Section 6.7 [Performance model example], page 51), 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 6.6 [Partitioning Data], page 51.

`lu/`: LU matrix factorization.

7 Advanced Topics

7.1 Defining a new data interface

7.1.1 `struct starpu_data_interface_ops_t` – Per-interface methods

Description:
TODO describe all the different fields

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

Description:
TODO describe all the different fields

7.1.3 An example of data interface

TODO

7.2 Defining a new scheduling policy

TODO

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_codelet.cl",
                                            &programs);
#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(&codelet);
#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 Codelet

```
#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 Codelet

```
#include <starpu.h>

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

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]);

        vector_mult_cuda<<<1,1>>>(val, n, *factor);

    cudaThreadSynchronize();
}
```

A.4 OpenCL Codelet

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;

    /* 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]);

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

    clFinish(queue);

    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;
        }
}
```