# 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 (eg. 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 (ie. CPUs and accelerators at the same time), but it also takes care to efficiently map and execute 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 manipulate 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 architecture 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

# 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 Configuring StarPU

### 2.1.1 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.

```
$ autoreconf -i
```

### 2.1.2 Configuring StarPU

```
$ ./configure
```

## 2.2 Building and Installing StarPU

### 2.2.1 Building

```
$ make
```

### 2.2.2 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
```

### 2.2.3 Installing

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

```
# make install
```

### 2.2.4 pkg-config configuration

It is possible that compiling and linking an application against StarPU requires to use specific flags or libraries (for instance `CUDA` or `libspe2`). Therefore, 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. So if StarPU was installed in `$(prefix_dir)`:

```
$ PKG_CONFIG_PATH = {PKG_CONFIG_PATH}:$(prefix_dir)/lib/
```

The flags required to compiled or linked 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 StarPU API

## 3.1 Initialization and Termination

### 3.1.1 `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 (eg. scheduling policy, number of cores, ...) by passing a non-null argument. Default configuration is used if the passed argument is `NULL`.

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

### 3.1.2 `struct starpu_conf` – StarPU runtime configuration

*Description*:
TODO

*Definition*:
TODO

### 3.1.3 `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 garanteed to be available until this method has been called.

*Prototype*: `void starpu_shutdown(void);`

## 3.2 Data Library

## 3.3 Codelets and Tasks

### 3.3.1 `starpu_task_create` – Allocate and Initialize a Task

*Description*:
TODO

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

## 3.4 Tags

### 3.4.1 `starpu_tag_t` – Task identifier

*Definition*:
TODO

### 3.4.2 starpu_tag_declare_deps – Declare the Dependencies of a Tag

*Description*:
TODO

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

### 3.4.3 starpu_tag_declare_deps_array – Declare the Dependencies of a Tag

*Description*:
TODO

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

### 3.4.4 starpu_tag_wait – Block until a Tag is terminated

*Description*:
TODO

*Prototype*: `void starpu_tag_wait(starpu_tag_t id);`

### 3.4.5 starpu_tag_wait_array – Block until a set of Tags is terminated

*Description*:
TODO

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

### 3.4.6 starpu_tag_remove – Destroy a Tag

*Description*:
TODO

*Prototype*: `void starpu_tag_remove(starpu_tag_t id);`

## 3.5 Extensions

### 3.5.1 CUDA extensions

### 3.5.2 Cell extensions

# 4 Basic Examples

## 4.1 Compiling and linking options

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

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

```
void cpu_func(starpu_data_interface_t *buffers, void *func_arg)
{
    float *array = func_arg;

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

starpu_codelet cl =
{
    .where = CORE,
    .core_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 (eg. 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 `CORE` value means that only CPUs can execute this codelet (see Section 3.3 [Codelets and Tasks], page 7 for more details on that field). When a CPU core executes a codelet, it calls the `.core_func` function, which *must* have the following prototype:

```
void (*core_func)(starpu_data_interface_t *, void *)
```

In this example, we can ignore the first argument of this function which gives a description of the input and output buffers (eg. 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 garantee that the initial buffer will be modified as well: this for instance implies that the buffer cannot be used as a synchronization medium.

### 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;

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

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

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

    /* submit the task to StarPU */
    starpu_submit_task(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 3.3.1 [starpu_task_create], page 7), 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.

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_submit_task` 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.

## 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.

Programmers can describe the data layout of their application so that StarPU is responsible for enforcing data coherency and availability accross 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 (eg. 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_register_*_data` 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 `n` elements of type `float` using the vector interface:

```
float tab[n];

starpu_data_handle tab_handle;
starpu_register_vector_data(&tab_handle, 0, tab, n, sizeof(float));
```

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. It is possible to construct a StarPU task that multiplies this vector by a constant factor:

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

task->cl = &cl;

task->buffers[0].handle = tab_handle;
task->buffers[0].mode = STARPU_RW;

task->cl_arg = &factor;
task->cl_arg_size = sizeof(float);
```

Since the factor is constant, it does not need a preliminary declaration, 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_func(starpu_data_interface_t *buffers, void *arg)
{
    unsigned i;
    float *factor = arg;

    /* length of the vector */
    unsigned n = buffers[0].vector.nx;
    /* local copy of the vector pointer */
    float *val = (float *)buffers[0].vector.ptr;

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

starpu_codelet cl = {
    .where = CORE,
    .core_func = scal_func,
    .nbuffers = 1
};
```

The second argument of the `scal_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 every buffers passed in the `task->buffers` array, the number of which is given by the `.nbuffers` field of the codelet structure. In the **vector interface**, the location of the vector (resp. its length) is accessible in the `.vector.ptr` (resp. `.vector.nx`) of this array. Since the vector is accessed in a read-write fashion, any modification will automatically affect future accesses to that vector made by other tasks.

## 4.4 Vector Scaling on an Hybrid CPU/GPU Machine

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

TODO

# 5 Advanced Topics