

StarPU Handbook

for StarPU 1.0.0rc1

Copyright © 2009–2011 Université de Bordeaux 1

Copyright © 2010, 2011, 2012 Centre National de la Recherche Scientifique

Copyright © 2011, 2012 Institut National de Recherche en Informatique et Automatique

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Table of Contents

Preface	1
1 Introduction to StarPU	3
1.1 Motivation	3
1.2 StarPU in a Nutshell	3
1.2.1 Codelet and Tasks	3
1.2.2 StarPU Data Management Library	4
1.2.3 Glossary	4
1.2.4 Research Papers	4
2 Installing StarPU	5
2.1 Downloading StarPU	5
2.1.1 Getting Sources	5
2.1.2 Optional dependencies	6
2.2 Configuration of StarPU	6
2.2.1 Generating Makefiles and configuration scripts	6
2.2.2 Running the configuration	6
2.3 Building and Installing StarPU	6
2.3.1 Building	6
2.3.2 Sanity Checks	6
2.3.3 Installing	6
3 Using StarPU	7
3.1 Setting flags for compiling and linking applications	7
3.2 Running a basic StarPU application	7
3.3 Kernel threads started by StarPU	7
3.4 Enabling OpenCL	8
4 Basic Examples	9
4.1 Compiling and linking options	9
4.2 Hello World	9
4.2.1 Required Headers	9
4.2.2 Defining a Codelet	9
4.2.3 Submitting a Task	11
4.2.4 Execution of Hello World	12
4.3 Manipulating Data: Scaling a Vector	12
4.3.1 Source code of Vector Scaling	12
4.3.2 Execution of Vector Scaling	14
4.4 Vector Scaling on an Hybrid CPU/GPU Machine	14
4.4.1 Definition of the CUDA Kernel	14
4.4.2 Definition of the OpenCL Kernel	15
4.4.3 Definition of the Main Code	16
4.4.4 Execution of Hybrid Vector Scaling	18

5	Advanced Examples	21
5.1	Using multiple implementations of a codelet	21
5.2	Enabling implementation according to capabilities	21
5.3	Task and Worker Profiling	23
5.4	Partitioning Data	24
5.5	Performance model example	25
5.6	Theoretical lower bound on execution time	27
5.7	Insert Task Utility	28
5.8	Debugging	29
5.9	The multiformat interface	29
5.10	On-GPU rendering	31
5.11	More examples	31
6	How to optimize performance with StarPU	33
6.1	Data management	33
6.2	Task submission	33
6.3	Task priorities	33
6.4	Task scheduling policy	34
6.5	Performance model calibration	34
6.6	Task distribution vs Data transfer	35
6.7	Data prefetch	35
6.8	Power-based scheduling	36
6.9	Profiling	36
6.10	CUDA-specific optimizations	36
7	Performance feedback	39
7.1	On-line performance feedback	39
7.1.1	Enabling on-line performance monitoring	39
7.1.2	Per-task feedback	39
7.1.3	Per-codelet feedback	39
7.1.4	Per-worker feedback	39
7.1.5	Bus-related feedback	40
7.1.6	StarPU-Top interface	40
7.2	Off-line performance feedback	41
7.2.1	Generating traces with FxT	41
7.2.2	Creating a Gantt Diagram	42
7.2.3	Creating a DAG with graphviz	42
7.2.4	Monitoring activity	42
7.3	Performance of codelets	43
7.4	Theoretical lower bound on execution time	44
8	Tips and Tricks to know about	45
8.1	How to initialize a computation library once for each worker? ..	45

9	StarPU MPI support	47
9.1	The API	47
9.1.1	Compilation	47
9.1.2	Initialisation	47
9.1.3	Communication	47
9.2	Simple Example	49
9.3	MPI Insert Task Utility	51
9.4	MPI Collective Operations	53
10	StarPU FFT support	55
10.0.1	Compilation	55
10.0.2	Initialisation	55
11	C Extensions	57
11.1	Defining Tasks	57
11.2	Registered Data Buffers	59
11.3	Using C Extensions Conditionally	60
12	SOCL OpenCL Extensions	63
13	StarPU Basic API	65
13.1	Initialization and Termination	65
13.2	Workers' Properties	67
13.3	Data Library	68
13.3.1	Introduction	68
13.3.2	Basic Data Library API	69
13.3.3	Access registered data from the application	71
13.4	Data Interfaces	72
13.4.1	Registering Data	72
13.4.2	Accessing Data Interfaces	73
13.4.2.1	Handle	74
13.4.2.2	Variable Data Interfaces	74
13.4.2.3	Vector Data Interfaces	74
13.4.2.4	Matrix Data Interfaces	75
13.4.2.5	Block Data Interfaces	76
13.4.2.6	BCSR Data Interfaces	77
13.4.2.7	CSR Data Interfaces	78
13.5	Data Partition	79
13.5.1	Basic API	79
13.5.2	Predefined filter functions	80
13.5.2.1	Partitioning BCSR Data	81
13.5.2.2	Partitioning BLAS interface	81
13.5.2.3	Partitioning Vector Data	81
13.5.2.4	Partitioning Block Data	82
13.6	Codelets and Tasks	82
13.7	Explicit Dependencies	87
13.8	Implicit Data Dependencies	88

13.9	Performance Model API	89
13.10	Profiling API	91
13.11	CUDA extensions	93
13.12	OpenCL extensions	93
13.12.1	Writing OpenCL kernels	93
13.12.2	Compiling OpenCL kernels	94
13.12.3	Loading OpenCL kernels	94
13.12.4	OpenCL statistics	94
13.13	Cell extensions	95
13.14	Miscellaneous helpers	95
14	StarPU Advanced API	97
14.1	Defining a new data interface	97
14.1.1	Data Interface API	97
14.1.2	An example of data interface	98
14.2	Multiformat Data Interface	99
14.3	Task Bundles	99
14.4	Task Lists	100
14.5	Defining a new scheduling policy	101
14.5.1	Scheduling Policy API	101
14.5.2	Source code	103
14.6	Expert mode	103
15	Configuring StarPU	105
15.1	Compilation configuration	105
15.1.1	Common configuration	105
15.1.1.1	--enable-debug	105
15.1.1.2	--enable-fast	105
15.1.1.3	--enable-verbose	105
15.1.1.4	--enable-coverage	105
15.1.2	Configuring workers	105
15.1.2.1	--enable-maxcpus=<number>	105
15.1.2.2	--disable-cpu	105
15.1.2.3	--enable-maxcudadev=<number>	105
15.1.2.4	--disable-cuda	105
15.1.2.5	--with-cuda-dir=<path>	105
15.1.2.6	--with-cuda-include-dir=<path>	105
15.1.2.7	--with-cuda-lib-dir=<path>	106
15.1.2.8	--disable-cuda-memcpy-peer	106
15.1.2.9	--enable-maxopencldev=<number>	106
15.1.2.10	--disable-opencl	106
15.1.2.11	--with-opencl-dir=<path>	106
15.1.2.12	--with-opencl-include-dir=<path>	106
15.1.2.13	--with-opencl-lib-dir=<path>	106
15.1.2.14	--enable-gordon	106
15.1.2.15	--with-gordon-dir=<path>	106
15.1.2.16	--enable-maximplementations=<number>	106
15.1.3	Advanced configuration	106

15.1.3.1	--enable-perf-debug	106
15.1.3.2	--enable-model-debug	106
15.1.3.3	--enable-stats	107
15.1.3.4	--enable-maxbuffers=<nbuffers>	107
15.1.3.5	--enable-allocation-cache	107
15.1.3.6	--enable-opengl-render	107
15.1.3.7	--enable-blas-lib=<name>	107
15.1.3.8	--disable-starpufft	107
15.1.3.9	--with-magma=<path>	107
15.1.3.10	--with-fxt=<path>	107
15.1.3.11	--with-perf-model-dir=<dir>	107
15.1.3.12	--with-mpicc=<path to mpicc>	107
15.1.3.13	--with-goto-dir=<dir>	107
15.1.3.14	--with-atlas-dir=<dir>	107
15.1.3.15	--with-mkl-cflags=<cflags>	107
15.1.3.16	--with-mkl-ldflags=<ldflags>	108
15.1.3.17	--disable-gcc-extensions	108
15.1.3.18	--disable-socl	108
15.2	Execution configuration through environment variables	108
15.2.1	Configuring workers	108
15.2.1.1	STARPU_NCPUS – Number of CPU workers	108
15.2.1.2	STARPU_NCUDA – Number of CUDA workers	108
15.2.1.3	STARPU_NOPENCL – Number of OpenCL workers	108
15.2.1.4	STARPU_NGORDON – Number of SPU workers (Cell) ..	108
15.2.1.5	STARPU_WORKERS_CPUID – Bind workers to specific CPUs	108
15.2.1.6	STARPU_WORKERS_CUDAID – Select specific CUDA devices	109
15.2.1.7	STARPU_WORKERS_OPENCLID – Select specific OpenCL devices	109
15.2.2	Configuring the Scheduling engine	109
15.2.2.1	STARPU_SCHED – Scheduling policy	109
15.2.2.2	STARPU_CALIBRATE – Calibrate performance models	109
15.2.2.3	STARPU_PREFETCH – Use data prefetch	109
15.2.2.4	STARPU_SCHED_ALPHA – Computation factor	110
15.2.2.5	STARPU_SCHED_BETA – Communication factor	110
15.2.3	Miscellaneous and debug	110
15.2.3.1	STARPU_SILENT – Disable verbose mode	110
15.2.3.2	STARPU_LOGFILENAME – Select debug file name	110
15.2.3.3	STARPU_FXT_PREFIX – FxT trace location	110
15.2.3.4	STARPU_LIMIT_GPU_MEM – Restrict memory size on the GPUs	110
15.2.3.5	STARPU_GENERATE_TRACE – Generate a Paje trace when StarPU is shut down	110

Appendix A	Full source code for the 'Scaling a Vector' example	111
A.1	Main application	111
A.2	CPU Kernel	113
A.3	CUDA Kernel	114
A.4	OpenCL Kernel	114
A.4.1	Invoking the kernel	114
A.4.2	Source of the kernel	115
Appendix B	GNU Free Documentation License	117
Function Index		125
Datatype Index		129

Preface

This manual documents the usage of StarPU version 1.0.0rc1. It was last updated on 25 January 2012.

1 Introduction to StarPU

1.1 Motivation

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

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

1.2 StarPU in a Nutshell

From a programming point of view, StarPU is not a new language but a library that executes tasks explicitly submitted by the application. The data that a task manipulates are automatically transferred onto the accelerator so that the programmer does not have to take care of complex data movements. StarPU also takes particular care of scheduling those tasks efficiently and allows scheduling experts to implement custom scheduling policies in a portable fashion. The target audience is typically developers of compilers or computation libraries which want to seamlessly extend them to support heterogeneous architectures.

1.2.1 Codelet and Tasks

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

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

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

1.2.2 StarPU Data Management Library

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

1.2.3 Glossary

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.4 Research Papers

Research papers about StarPU can be found at

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

Notably a good overview in the research report

<http://hal.archives-ouvertes.fr/inria-00467677>

2 Installing StarPU

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

2.1 Downloading StarPU

2.1.1 Getting Sources

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

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

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

If you need to have write access on the StarPU project, you can also choose to become a member of the project **starpup**. 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/starpup/trunk
```
- using the anonymous access via https:


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

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


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

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

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

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

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

2.1.2 Optional dependencies

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

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

2.2 Configuration of StarPU

2.2.1 Generating Makefiles and configuration scripts

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

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

2.2.2 Running the configuration

```
% ./configure
```

Details about options that are useful to give to `./configure` are given in [Section 15.1 \[Compilation configuration\]](#), page 105.

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
```

Libtool interface versioning information are included in libraries names (`libstarpu-1.0.so`, `libstarpumpi-1.0.so` and `libstarpufft-1.0.so`).

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-1.0 # options for the compiler
% pkg-config --libs libstarpu-1.0   # options for the linker
```

Also pass the `--static` option if the application is to be linked statically¹.

3.2 Running a basic StarPU application

Basic examples using StarPU are built in the directory `examples/basic_examples/` (and installed in `$prefix_dir/lib/starpu/examples/`). You can for example run the example `vector_scal`.

```
% ./examples/basic_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 `$STARPU_HOME/.starpu/` is created, performance models will be stored in that directory (`STARPU_HOME` defaults to `$HOME`).

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

3.3 Kernel threads started by StarPU

StarPU automatically binds one thread per CPU core. It does not use SMT/hyperthreading because kernels are usually already optimized for using a full core, and using hyperthreading would make kernel calibration rather random.

Since driving GPUs is a CPU-consuming task, StarPU dedicates one core per GPU.

While StarPU tasks are executing, the application is not supposed to do computations in the threads it starts itself, tasks should be used instead.

TODO: add a StarPU function to bind an application thread (e.g. the main thread) to a dedicated core (and thus disable the corresponding StarPU CPU worker).

¹ It is still possible to use the API provided in the version 0.9 of StarPU by calling `pkg-config` with the `libstarpu` package. Similar packages are provided for `libstarpumpi` and `libstarpufft`.

3.4 Enabling OpenCL

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

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
```


4 Basic Examples

4.1 Compiling and linking options

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

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

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

```
CFLAGS      +=      $$$(pkg-config --cflags libstarpu-1.0)
LDFLAGS     +=      $$$(pkg-config --libs libstarpu-1.0)
```

Also pass the `--static` option if the application is to be linked statically.

4.2 Hello World

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

4.2.1 Required Headers

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

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

4.2.2 Defining a Codelet

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

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

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

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

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

We create a codelet which may only be executed on the CPUs. The `where` field is a bitmask that defines where the codelet may be executed. Here, the `STARPU_CPU` value means that only CPUs can execute this codelet (see [Section 13.6 \[Codelets and Tasks\]](#), [page 82](#) for more details on this field). Note that the `where` field is optional, when unset its value is automatically set based on the availability of the different `XXX_funcs` fields. When a CPU core executes a codelet, it calls the `cpu_func` function, which *must* have the following prototype:

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

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

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

4.2.3 Submitting a Task

```

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

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

    struct starpu_task *task = starpu_task_create();

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

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

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

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

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

    /* terminate StarPU */
    starpu_shutdown();

    return 0;
}

```

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

In the example above, a task structure is allocated by a call to `starpu_task_create`. This function only allocates and fills the corresponding structure with the default settings (see [Section 13.6 \[Codelets and Tasks\]](#), page 82), but it does not submit the task to StarPU.

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

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

terminates, and if several tasks are submitted with various parameters, each of them must be given a pointer to their own buffer.

Once a task has been executed, an optional callback function is be called. While the computational kernel could be offloaded on various architectures, the callback function is always executed on a CPU. The `callback_arg` pointer is passed as an argument of the callback. The prototype of a callback function must be:

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

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

4.2.4 Execution of Hello World

```
% make hello_world
cc $(pkg-config --cflags libstarpu-1.0) $(pkg-config --libs libstarpu-1.0) hello_world.c -o hello_world
% ./hello_world
Hello world (params = {1, 2.000000} )
Callback function (arg 42)
```

4.3 Manipulating Data: Scaling a Vector

The previous example has shown how to submit tasks. In this section, we show how StarPU tasks can manipulate data. The full source code for this example is given in [Appendix A \[Full source code for the 'Scaling a Vector' example\], page 111](#).

4.3.1 Source code of Vector Scaling

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

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

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

```
float vector[NX];

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

The first argument, called the **data handle**, is an opaque pointer which designates the array in StarPU. This is also the structure which is used to describe which data is used by a task. The second argument is the node number where the data originally resides. Here it is 0 since the **vector** array is in the main memory. Then comes the pointer **vector** where the data can be found in main memory, the number of elements in the vector and the size of each element. The following shows how to construct a StarPU task that will manipulate the vector and a constant factor.

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

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

starpu_task_submit(task);
```

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

The definition of the codelet can be written as follows:

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

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

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

struct starpu_codelet cl = {
    .where = STARPU_CPU,
    .cpu_funcs = { scal_cpu_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

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

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

4.3.2 Execution of Vector Scaling

```
% make vector_scal
cc $(pkg-config --cflags libstarpu-1.0) $(pkg-config --libs libstarpu-1.0) vector_scal.c -o vector_scal
% ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

4.4 Vector Scaling on an Hybrid CPU/GPU Machine

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

4.4.1 Definition of the CUDA Kernel

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

```

#include <starpu.h>
#include <starpu_cuda.h>

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

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

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

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

    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}

```

4.4.2 Definition of the OpenCL Kernel

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

```

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

```

Contrary to CUDA and CPU, `STARPU_VECTOR_GET_DEV_HANDLE` has to be used, which returns a `cl_mem` (which is not a device pointer, but an OpenCL handle), which can be passed as such to the OpenCL kernel. The difference is important when using partitioning, see [Section 5.4 \[Partitioning Data\]](#), page 24.

```

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

extern struct starpu_opencl_program programs;

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

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

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

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

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

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

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

    starpu_opencl_release_kernel(kernel);
}

```

4.4.3 Definition of the Main Code

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

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


```

#include <starpu.h>

#define NX 2048

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

/* Definition of the codelet */
static struct starpu_codelet cl = {
    .where = STARPU_CPU|STARPU_CUDA|STARPU_OPENGL; /* It can be executed on a CPU, */
                                                    /* on a CUDA device, or on an OpenCL device */
    .cuda_funcs = { scal_cuda_func, NULL},
    .cpu_funcs = {scal_cpu_func, NULL },
    .opengl_funcs = { scal_opengl_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
}

#ifdef STARPU_USE_OPENGL
/* The compiled version of the OpenCL program */
struct starpu_opengl_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_t vector_handle;

    starpu_init(NULL);                                /* Initialising StarPU */

#ifdef STARPU_USE_OPENGL
    starpu_opengl_load_opengl_from_file(
        "examples/basic_examples/vector_scal_opengl_codelet.cl",
        &programs, NULL);
#endif

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

```

```

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

/* Definition of the task */
task = starpu_task_create();
task->cl = &cl;
task->handles[0] = vector_handle;
task->cl_arg = &factor;
task->cl_arg_size = sizeof(factor);

```

```

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

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

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

```

```

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

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

return 0;
}

```

4.4.4 Execution of Hybrid Vector Scaling

The Makefile given at the beginning of the section must be extended to give the rules to compile the CUDA source code. Note that the source file of the OpenCL kernel does not need to be compiled now, it will be compiled at run-time when calling the function `starpu_opencil_load_opencil_from_file()` (see [\[starpu_opencil_load_opencil_from_file\]](#), page 94).

```

CFLAGS += $(shell pkg-config --cflags libstarpu-1.0)
LDLAGS += $(shell pkg-config --libs libstarpu-1.0)
CC      = gcc

vector_scal: vector_scal.o vector_scal_cpu.o vector_scal_cuda.o vector_scal_opencil.o

%.o: %.cu
    nvcc $(CFLAGS) $< -c $

clean:
    rm -f vector_scal *.o

```

% make

and to execute it, with the default configuration:

```

% ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000

```

or for example, by disabling CPU devices:

```

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

```

or by disabling CUDA devices (which may permit to enable the use of OpenCL, see [Section 3.4 \[Enabling OpenCL\], page 8](#)):

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


5 Advanced Examples

5.1 Using multiple implementations of a codelet

One may want to write multiple implementations of a codelet for a single type of device and let StarPU choose which one to run. As an example, we will show how to use SSE to scale a vector. The codelet can be written as follows :

```
#include <xmmintrin.h>

void scal_sse_func(void *buffers[], void *cl_arg)
{
    float *vector = (float *) STARPU_VECTOR_GET_PTR(buffers[0]);
    unsigned int n = STARPU_VECTOR_GET_NX(buffers[0]);
    unsigned int n_iterations = n/4;
    if (n % 4 != 0)
        n_iterations++;

    __m128 *VECTOR = (__m128*) vector;
    __m128 factor __attribute__((aligned(16)));
    factor = _mm_set1_ps(*(float *) cl_arg);

    unsigned int i;
    for (i = 0; i < n_iterations; i++)
        VECTOR[i] = _mm_mul_ps(factor, VECTOR[i]);
}
```

```
struct starpu_codelet cl = {
    .where = STARPU_CPU,
    .cpu_funcs = { scal_cpu_func, scal_sse_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

Schedulers which are multi-implementation aware (only **dmda**, **heft** and **pheft** for now) will use the performance models of all the implementations it was given, and pick the one that seems to be the fastest.

5.2 Enabling implementation according to capabilities

Some implementations may not run on some devices. For instance, some CUDA devices do not support double floating point precision, and thus the kernel execution would just fail; or the device may not have enough shared memory for the implementation being used. The **can_execute** field of the **struct starpu_codelet** structure permits to express this. For instance:

```

static int can_execute(unsigned workerid, struct starpu_task *task, unsigned nimpl)
{
    const struct cudaDeviceProp *props;
    if (starpu_worker_get_type(workerid) == STARPU_CPU_WORKER)
        return 1;
    /* Cuda device */
    props = starpu_cuda_get_device_properties(workerid);
    if (props->major >= 2 || props->minor >= 3)
        /* At least compute capability 1.3, supports doubles */
        return 1;
    /* Old card, does not support doubles */
    return 0;
}

struct starpu_codelet cl = {
    .where = STARPU_CPU|STARPU_CUDA,
    .can_execute = can_execute,
    .cpu_funcs = {cpu_func, NULL },
    .cuda_funcs = { gpu_func, NULL }
    .nbuffers = 1,
    .modes = { STARPU_RW }
};

```

This can be essential e.g. when running on a machine which mixes various models of CUDA devices, to take benefit from the new models without crashing on old models.

Note: the `can_execute` function is called by the scheduler each time it tries to match a task with a worker, and should thus be very fast. The `starpu_cuda_get_device_properties` provides a quick access to CUDA properties of CUDA devices to achieve such efficiency.

Another example is compiling CUDA code for various compute capabilities, resulting with two CUDA functions, e.g. `scal_gpu_13` for compute capability 1.3, and `scal_gpu_20` for compute capability 2.0. Both functions can be provided to StarPU by using `cuda_funcs`, and `can_execute` can then be used to rule out the `scal_gpu_20` variant on a CUDA device which will not be able to execute it:

```

static int can_execute(unsigned workerid, struct starpu_task *task, unsigned nimpl)
{
    const struct cudaDeviceProp *props;
    if (starpu_worker_get_type(workerid) == STARPU_CPU_WORKER)
        return 1;
    /* Cuda device */
    if (nimpl == 0)
        /* Trying to execute the 1.3 capability variant, we assume it is ok in all cases. */
        return 1;
    /* Trying to execute the 2.0 capability variant, check that the card can do it. */
    props = starpu_cuda_get_device_properties(workerid);
    if (props->major >= 2 || props->minor >= 0)
        /* At least compute capability 2.0, can run it */
        return 1;
    /* Old card, does not support 2.0, will not be able to execute the 2.0 variant. */
    return 0;
}

struct starpu_codelet cl = {
    .where = STARPU_CPU|STARPU_CUDA,
    .can_execute = can_execute,
    .cpu_funcs = { cpu_func, NULL },
    .cuda_funcs = { scal_gpu_13, scal_gpu_20, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};

```

Note: the most generic variant should be provided first, as some schedulers are not able to try the different variants.

5.3 Task and Worker Profiling

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

```

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

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

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

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

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

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

```

```

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

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

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

    char workername[128];
    starpu_worker_get_name(worker, workername, 128);
    fprintf(stderr, "Worker %s:\n", workername);
    fprintf(stderr, "\tttotal 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);
}

```

5.4 Partitioning Data

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

```

int vector[NX];
starpu_data_handle_t 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
};
starpu_data_partition(handle, &f);

```

The task submission then uses `starpu_data_get_sub_data` to retrieve the sub-handles to be passed as tasks parameters.


```

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

    task->handles[0] = sub_handle;
    task->cl = &cl;
    task->synchronous = 1;
    task->cl_arg = &factor;
    task->cl_arg_size = sizeof(factor);

    starpu_task_submit(task);
}

```

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

Wherever the whole piece of data is already available, the partitioning will be done in-place, i.e. without allocating new buffers but just using pointers inside the existing copy. This is particularly important to be aware of when using OpenCL, wherethe kernel parameters are not pointers, but handles. The kernel thus needs to be also passed the offset within the OpenCL buffer:

```

void openc1_func(void *buffers[], void *cl_arg)
{
    cl_mem vector = (cl_mem) STARPU_VECTOR_GET_DEV_HANDLE(buffers[0]);
    unsigned offset = STARPU_BLOCK_GET_OFFSET(buffers[0]);

    ...
    clSetKernelArg(kernel, 0, sizeof(vector), &vector);
    clSetKernelArg(kernel, 1, sizeof(offset), &offset);
    ...
}

```

And the kernel has to shift from the pointer passed by the OpenCL driver:

```

__kernel void openc1_kernel(__global int *vector, unsigned offset)
{
    block = (__global void *)block + offset;
    ...
}

```

5.5 Performance model example

To achieve good scheduling, StarPU scheduling policies need to be able to estimate in advance the duration of a task. This is done by giving to codelets a performance model, by defining a `starpu_perfmodel` structure and providing its address in the `model` field of the `struct starpu_codelet` structure. The `symbol` and `type` fields of `starpu_perfmodel` are mandatory, to give a name to the model, and the type of the model, since there are several kinds of performance models.

- Measured at runtime (STARPU_HISTORY_BASED model type). This assumes that for a given set of data input/output sizes, the performance will always be about the same.

This is very true for regular kernels on GPUs for instance (<0.1% error), and just a bit less true on CPUs (~1% error). This also assumes that there are few different sets of data input/output sizes. StarPU will then keep record of the average time of previous executions on the various processing units, and use it as an estimation. History is done per task size, by using a hash of the input and output sizes as an index. It will also save it in `~/.starpu/sampling/codelets` for further executions, and can be observed by using the `starpu_perfmodel_display` command, or drawn by using the `starpu_perfmodel_plot`. The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. Measurements are only done when using a task scheduler which makes use of it, such as `heft` or `dmda`.

The following is a small code example.

If e.g. the code is recompiled with other compilation options, or several variants of the code are used, the symbol string should be changed to reflect that, in order to recalibrate a new model from zero. The symbol string can even be constructed dynamically at execution time, as long as this is done before submitting any task using it.

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

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

- Measured at runtime and refined by regression (`STARPU_*REGRESSION_BASED` model type). This still assumes performance regularity, but can work with various data input sizes, by applying regression over observed execution times. `STARPU_REGRESSION_BASED` uses an $a \cdot n^b$ regression form, `STARPU_NL_REGRESSION_BASED` uses an $a \cdot n^b + c$ (more precise than `STARPU_REGRESSION_BASED`, but costs a lot more to compute). For instance, `tests/perfmodels/regression_based.c` uses a regression-based performance model for the `memset` operation. Of course, the application has to issue tasks with varying size so that the regression can be computed. StarPU will not trust the regression unless there is at least 10% difference between the minimum and maximum observed input size. For non-linear regression, since computing it is quite expensive, it is only done at termination of the application. This means that the first execution uses history-based performance model to perform scheduling.
- Provided as an estimation from the application itself (`STARPU_COMMON` model type and `cost_function` field), see for instance `examples/common/blas_model.h` and `examples/common/blas_model.c`.
- Provided explicitly by the application (`STARPU_PER_ARCH` model type): the `.per_`

`arch[arch][nimpl].cost_function` fields have to be filled with pointers to functions which return the expected duration of the task in micro-seconds, one per architecture.

For the `STARPU_HISTORY_BASED` and `STARPU_*REGRESSION_BASE`, the total size of task data (both input and output) is used as an index by default. The `size_base` field of `struct starpu_perfmodel` however permits the application to override that, when for instance some of the data do not matter for task cost (e.g. mere reference table), or when using sparse structures (in which case it is the number of non-zeros which matter), or when there is some hidden parameter such as the number of iterations, etc.

How to use schedulers which can benefit from such performance model is explained in [Section 6.4 \[Task scheduling policy\]](#), page 34.

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

The application can request time estimations from the StarPU performance models by filling a task structure as usual without actually submitting it. The data handles can be created by calling `starpu_data_register` functions with a NULL pointer (and need to be unregistered as usual) and the desired data sizes. The `starpu_task_expected_length` and `starpu_task_expected_power` functions can then be called to get an estimation of the task duration on a given arch. `starpu_task_destroy` needs to be called to destroy the dummy task afterwards. See `tests/perfmodels/regression_based.c` for an example.

5.6 Theoretical lower bound on execution time

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

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

Setting `deps` to 0 will only take into account the actual computations on processing units. It however still properly takes into account the varying performances of kernels and

processing units, which is quite more accurate than just comparing StarPU performances with the fastest of the kernels being used.

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

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

5.7 Insert Task Utility

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

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

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

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

- `STARPU_R`, `STARPU_W`, `STARPU_RW`, `STARPU_SCRATCH`, `STARPU_REDUX` an access mode followed by a data handle;
- `STARPU_VALUE` followed by a pointer to a constant value and the size of the constant;
- `STARPU_CALLBACK` followed by a pointer to a callback function;
- `STARPU_CALLBACK_ARG` followed by a pointer to be given as an argument to the callback function;
- `STARPU_CALLBACK_WITH_ARG` followed by two pointers: one to a callback function, and the other to be given as an argument to the callback function; this is equivalent to using both `STARPU_CALLBACK` and `STARPU_CALLBACK_WITH_ARG`
- `STARPU_PRIORITY` followed by a integer defining a priority level.

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

Here the implementation of the codelet:

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

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

struct starpu_codelet mycodelet = {
    .where = STARPU_CPU,
```

```

        .cpu_funcs = { func_cpu, NULL },
        .nbuffers = 2,
        .modes = { STARPU_RW, STARPU_RW }
};

```

And the call to the `starpu_insert_task` wrapper:

```

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

```

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

```

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

```

If some part of the task insertion depends on the value of some computation, the `STARPU_DATA_ACQUIRE_CB` macro can be very convenient. For instance, assuming that the index variable `i` was registered as handle `i_handle`:

```

/* Compute which portion we will work on, e.g. pivot */
starpu_insert_task(&which_index, STARPU_W, i_handle, 0);

/* And submit the corresponding task */
STARPU_DATA_ACQUIRE_CB(i_handle, STARPU_R, starpu_insert_task(&work, STARPU_RW, A_handle[i], 0));

```

The `STARPU_DATA_ACQUIRE_CB` macro submits an asynchronous request for acquiring data `i` for the main application, and will execute the code given as third parameter when it is acquired. In other words, as soon as the value of `i` computed by the `which_index` codelet can be read, the portion of code passed as third parameter of `STARPU_DATA_ACQUIRE_CB` will be executed, and is allowed to read from `i` to use it e.g. as an index. Note that this macro is only available when compiling StarPU with the compiler `gcc`.

5.8 Debugging

StarPU provides several tools to help debugging applications. Execution traces can be generated and displayed graphically, see [Section 7.2.1 \[Generating traces\], page 41](#). Some gdb helpers are also provided to show the whole StarPU state:

```

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

```

5.9 The multiformat interface

It may be interesting to represent the same piece of data using two different data structures : one that would only be used on CPUs, and one that would only be used on GPUs. This can be done by using the multiformat interface. StarPU will be able to convert data from

one data structure to the other when needed. Note that the heft scheduler is the only one optimized for this interface. The user must provide StarPU with conversion codelets :

```
#define NX 1024
struct point array_of_structs[NX];
starpu_data_handle_t handle;

/*
 * The conversion of a piece of data is itself a task, though it is created,
 * submitted and destroyed by StarPU internals and not by the user. Therefore,
 * we have to define two codelets.
 * Note that for now the conversion from the CPU format to the GPU format has to
 * be executed on the GPU, and the conversion from the GPU to the CPU has to be
 * executed on the CPU.
 */
#ifdef STARPU_USE_OPENCL
void cpu_to_opengl_opengl_func(void *buffers[], void *args);
struct starpu_codelet cpu_to_opengl_cl = {
    .where = STARPU_OPENGL,
    .opengl_funcs = { cpu_to_opengl_opengl_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};

void opengl_to_cpu_func(void *buffers[], void *args);
struct starpu_codelet opengl_to_cpu_cl = {
    .where = STARPU_CPU,
    .cpu_funcs = { opengl_to_cpu_func, NULL },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
#endif

struct starpu_multiformat_data_interface_ops format_ops = {
#ifdef STARPU_USE_OPENCL
    .opengl_elemsize = 2 * sizeof(float),
    .cpu_to_opengl_cl = &cpu_to_opengl_cl,
    .opengl_to_cpu_cl = &opengl_to_cpu_cl,
#else
    .cpu_elemsize = 2 * sizeof(float),
    ...
};
starpu_multiformat_data_register(handle, 0, &array_of_structs, NX, &format_ops);
```

Kernels can be written almost as for any other interface. Note that `STARPU_MULTIFORMAT_GET_PTR` shall only be used for CPU kernels. CUDA kernels must use `STARPU_MULTIFORMAT_GET_CUDA_PTR`, and OpenCL kernels must use `STARPU_MULTIFORMAT_GET_OPENGL_PTR`. `STARPU_MULTIFORMAT_GET_NX` may be used in any kind of kernel.

```

static void
multiformat_scal_cpu_func(void *buffers[], void *args)
{
    struct point *aos;
    unsigned int n;

    aos = STARPU_MULTIFORMAT_GET_PTR(buffers[0]);
    n = STARPU_MULTIFORMAT_GET_NX(buffers[0]);
    ...
}

extern "C" void multiformat_scal_cuda_func(void *buffers[], void *_args)
{
    unsigned int n;
    struct struct_of_arrays *soa;

    soa = (struct struct_of_arrays *) STARPU_MULTIFORMAT_GET_CUDA_PTR(buffers[0]);
    n = STARPU_MULTIFORMAT_GET_NX(buffers[0]);

    ...
}

```

A full example may be found in `examples/basic_examples/multiformat.c`.

5.10 On-GPU rendering

Graphical-oriented applications need to draw the result of their computations, typically on the very GPU where these happened. Technologies such as OpenGL/CUDA interoperability permit to let CUDA directly work on the OpenGL buffers, making them thus immediately ready for drawing, by mapping OpenGL buffer, textures or renderbuffer objects into CUDA. To achieve this with StarPU, it simply needs to be given the CUDA pointer at registration, for instance:

```

for (workerid = 0; workerid < starpu_worker_get_count(); workerid++)
    if (starpu_worker_get_type(workerid) == STARPU_CUDA_WORKER)
        break;

cudaSetDevice(starpu_worker_get_devid(workerid));
cudaGraphicsResourceGetMappedPointer((void**)&output, &num_bytes, resource);
starpu_vector_data_register(&handle, starpu_worker_get_memory_node(workerid), output, num_bytes / sizeof(float));

starpu_insert_task(&cl, STARPU_RW, handle, 0);

starpu_data_unregister(handle);

cudaSetDevice(starpu_worker_get_devid(workerid));
cudaGraphicsUnmapResources(1, &resource, 0);

/* Now display it */

```

5.11 More examples

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

`incrementer/:`

Trivial incrementation test.

`basic_examples/:`

Simple documented Hello world (as shown in [Section 4.2 \[Hello World\]](#), [page 9](#)), vector/scalar product (as shown in [Section 4.4 \[Vector Scaling on an Hybrid CPU/GPU Machine\]](#), [page 14](#)), matrix product examples (as shown in [Section 5.5 \[Performance model example\]](#), [page 25](#)), an example using the blocked matrix data interface, an example using the variable data interface, and an example using different formats on CPUs and GPUs.

`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 5.4 \[Partitioning Data\]](#), [page 24](#).

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

`cholesky/:`

Cholesky matrix factorization, see for instance `cholesky_implicit.c`.

6 How to optimize performance with StarPU

TODO: improve!

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

6.1 Data management

When the application allocates data, whenever possible it should use the `starpu_malloc` function, which will ask CUDA or OpenCL to make the allocation itself and pin the corresponding allocated memory. This is needed to permit asynchronous data transfer, i.e. permit data transfer to overlap with computations. Otherwise, the trace will show that the `DriverCopyAsync` state takes a lot of time, this is because CUDA or OpenCL then reverts to synchronous transfers.

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

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

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

```
starpu_data_set_wt_mask(img_handle, ~0U);
```

will request to always automatically broadcast the updated data to all memory nodes.

6.2 Task submission

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

6.3 Task priorities

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

6.4 Task scheduling policy

By default, StarPU uses the **eager** simple greedy scheduler. This is because it provides correct load balance even if the application codelets do not have performance models. If your application codelets have performance models (see [Section 5.5 \[Performance model example\]](#), [page 25](#) for examples showing how to do it), you should change the scheduler thanks to the `STARPU_SCHED` environment variable. For instance `export STARPU_SCHED=dmda`. Use `help` to get the list of available schedulers.

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

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

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

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

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

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

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

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

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

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

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

6.5 Performance model calibration

Most schedulers are based on an estimation of codelet duration on each kind of processing unit. For this to be possible, the application programmer needs to configure a performance model for the codelets of the application (see [Section 5.5 \[Performance model example\]](#), [page 25](#) for instance). History-based performance models use on-line calibration. StarPU will automatically calibrate codelets which have never been calibrated yet, and save the result in `~/.starpu/sampling/codelets`. The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. To force continuing calibration, use `export STARPU_CALIBRATE=1`. This may be necessary if your application has not-so-stable performance. StarPU will force calibration (and thus ignore the current result) until 10 (`_STARPU_CALIBRATION_MINIMUM`) measurements have been made on each architecture, to avoid badly scheduling tasks just because the first measurements were not so

good. Details on the current performance model status can be obtained from the `starpu_perfmodel_display` command: the `-l` option lists the available performance models, and the `-s` option permits to choose the performance model to be displayed. The result looks like:

```
$ starpu_perfmodel_display -s starpu_dlu_lu_model_22
performance model for cpu
# hash      size      mean          dev          n
880805ba  98304    2.731309e+02  6.010210e+01  1240
b50b6605  393216   1.469926e+03  1.088828e+02  1240
5c6c3401  1572864  1.125983e+04  3.265296e+03  1240
```

Which shows that for the LU 22 kernel with a 1.5MiB matrix, the average execution time on CPUs was about 11ms, with a 3ms standard deviation, over 1240 samples. It is a good idea to check this before doing actual performance measurements.

A graph can be drawn by using the `starpu_perfmodel_plot`:

```
$ starpu_perfmodel_plot -s starpu_dlu_lu_model_22
98304 393216 1572864
$ gnuplot starpu_starpu_dlu_lu_model_22.gp
$ gv starpu_starpu_dlu_lu_model_22.eps
```

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

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

6.6 Task distribution vs Data transfer

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

6.7 Data prefetch

The `heft`, `dmda` and `pheft` scheduling policies perform data prefetch (see [Section 15.2.2.3 \[STARPU_PREFETCH\]](#), page 109): as soon as a scheduling decision is taken for a task, requests are issued to transfer its required data to the target processing unit, if needed,

so that when the processing unit actually starts the task, its data will hopefully be already available and it will not have to wait for the transfer to finish.

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

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

6.8 Power-based scheduling

If the application can provide some power performance model (through the `power_model` field of the codelet structure), StarPU will take it into account when distributing tasks. The target function that the `dmda` scheduler minimizes becomes $\alpha * T_{\text{execution}} + \beta * T_{\text{data_transfer}} + \gamma * \text{Consumption}$, where `Consumption` is the estimated task consumption in Joules. To tune this parameter, use `export STARPU_SCHED_GAMMA=3000` for instance, to express that each Joule (i.e kW during 1000us) is worth 3000us execution time penalty. Setting `alpha` and `beta` to zero permits to only take into account power consumption.

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

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

6.9 Profiling

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

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

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

6.10 CUDA-specific optimizations

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

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

StarPU already does appropriate calls for the CUBLAS library.

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

7 Performance feedback

7.1 On-line performance feedback

7.1.1 Enabling on-line performance monitoring

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

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

More details about the performance monitoring API are available in [section Section 13.10 \[Profiling API\], page 91](#).

7.1.2 Per-task feedback

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

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

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

7.1.3 Per-codelet feedback

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

7.1.4 Per-worker feedback

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

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

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

When an FxT trace is generated (see [Section 7.2.1 \[Generating traces\], page 41](#)), it is also possible to use the `starpu_top` script (described in [Section 7.2.4 \[starpu-top\], page 42](#)) to generate a graphic showing the evolution of these values during the time, for the different workers.

7.1.5 Bus-related feedback

TODO

The bus speed measured by StarPU can be displayed by using the `starpu_machine_display` tool, for instance:

```
StarPU has found :
    3 CUDA devices
        CUDA 0 (Tesla C2050 02:00.0)
        CUDA 1 (Tesla C2050 03:00.0)
        CUDA 2 (Tesla C2050 84:00.0)
from   to RAM      to CUDA 0      to CUDA 1      to CUDA 2
RAM    0.000000    5176.530428    5176.492994    5191.710722
CUDA 0 4523.732446  0.000000     2414.074751    2417.379201
CUDA 1 4523.718152  2414.078822    0.000000     2417.375119
CUDA 2 4534.229519  2417.069025    2417.060863    0.000000
```

7.1.6 StarPU-Top interface

StarPU-Top is an interface which remotely displays the on-line state of a StarPU application and permits the user to change parameters on the fly.

Variables to be monitored can be registered by calling the `starpu_top_add_data_boolean`, `starpu_top_add_data_integer`, `starpu_top_add_data_float` functions, e.g.:

```
starpu_top_data *data = starpu_top_add_data_integer("mynum", 0, 100, 1);
```

The application should then call `starpu_top_init_and_wait` to give its name and wait for StarPU-Top to get a start request from the user. The name is used by StarPU-Top to quickly reload a previously-saved layout of parameter display.

```
starpu_top_init_and_wait("the application");
```

The new values can then be provided thanks to `starpu_top_update_data_boolean`, `starpu_top_update_data_integer`, `starpu_top_update_data_float`, e.g.:

```
starpu_top_update_data_integer(data, mynum);
```


Updateable parameters can be registered thanks to `starpu_top_register_parameter_boolean`, `starpu_top_register_parameter_integer`, `starpu_top_register_parameter_float`, e.g.:

```
float alpha;
starpu_top_register_parameter_float("alpha", &alpha, 0, 10, modif_hook);
```

`modif_hook` is a function which will be called when the parameter is being modified, it can for instance print the new value:

```
void modif_hook(struct starpu_top_param *d) {
    fprintf(stderr, "%s has been modified: %f\n", d->name, alpha);
}
```

Task schedulers should notify StarPU-Top when it has decided when a task will be scheduled, so that it can show it in its Gantt chart, for instance:

```
starpu_top_task_prevision(task, workerid, begin, end);
```

Starting StarPU-Top and the application can be done two ways:

- The application is started by hand on some machine (and thus already waiting for the start event). In the Preference dialog of StarPU-Top, the SSH checkbox should be unchecked, and the hostname and port (default is 2011) on which the application is already running should be specified. Clicking on the connection button will thus connect to the already-running application.
- StarPU-Top is started first, and clicking on the connection button will start the application itself (possibly on a remote machine). The SSH checkbox should be checked, and a command line provided, e.g.:

```
ssh myserver STARPU_SCHED=heft ./application
```

If port 2011 of the remote machine can not be accessed directly, an ssh port bridge should be added:

```
ssh -L 2011:localhost:2011 myserver STARPU_SCHED=heft ./application
```

and "localhost" should be used as IP Address to connect to.

7.2 Off-line performance feedback

7.2.1 Generating traces with FxT

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

You can either get a tarball:

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

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

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

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

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

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

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

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

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

7.2.2 Creating a Gantt Diagram

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

```
% starpu_fxt_tool -i filename
```

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

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

```
% vite paje.trace
```

7.2.3 Creating a DAG with graphviz

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

```
$ starpu_fxt_tool -i filename
```

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

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

7.2.4 Monitoring activity

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

```
$ starpu_fxt_tool -i filename
```

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

```
$ starpu_top activity.data
```

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

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

7.3 Performance of codelets

The performance model of codelets (described in [Section 5.5 \[Performance model example\]](#), [page 25](#)) can be examined by using the `starpu_perfmodel_display` tool:

```
$ starpu_perfmodel_display -l
file: <malloc_pinned.hannibal>
file: <starpu_slv_lu_model_21.hannibal>
file: <starpu_slv_lu_model_11.hannibal>
file: <starpu_slv_lu_model_22.hannibal>
file: <starpu_slv_lu_model_12.hannibal>
```

Here, the codelets of the lu example are available. We can examine the performance of the 22 kernel (in micro-seconds):

```
$ starpu_perfmodel_display -s starpu_slv_lu_model_22
performance model for cpu
# hash      size      mean      dev      n
57618ab0    19660800  2.851069e+05  1.829369e+04  109
performance model for cuda_0
# hash      size      mean      dev      n
57618ab0    19660800  1.164144e+04  1.556094e+01  315
performance model for cuda_1
# hash      size      mean      dev      n
57618ab0    19660800  1.164271e+04  1.330628e+01  360
performance model for cuda_2
# hash      size      mean      dev      n
57618ab0    19660800  1.166730e+04  3.390395e+02  456
```

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

The `starpu_regression_display` tool does the same for regression-based performance models. It also writes a `.gp` file in the current directory, to be run in the `gnuplot` tool, which shows the corresponding curve.

The same can also be achieved by using StarPU's library API, see [Section 13.9 \[Performance Model API\]](#), page 89 and notably the `starpu_load_history_debug` function. The source code of the `starpu_perfmodel_display` tool can be a useful example.

7.4 Theoretical lower bound on execution time

See [Section 5.6 \[Theoretical lower bound on execution time\]](#), page 27 for an example on how to use this API. It permits to record a trace of what tasks are needed to complete the application, and then, by using a linear system, provide a theoretical lower bound of the execution time (i.e. with an ideal scheduling).

The computed bound is not really correct when not taking into account dependencies, but for an application which have enough parallelism, it is very near to the bound computed with dependencies enabled (which takes a huge lot more time to compute), and thus provides a good-enough estimation of the ideal execution time.

```
void starpu_bound_start (int deps, int prio) [Function]
    Start recording tasks (resets stats).  deps tells whether dependencies should be
    recorded too (this is quite expensive)

void starpu_bound_stop (void) [Function]
    Stop recording tasks

void starpu_bound_print_dot (FILE *output) [Function]
    Print the DAG that was recorded

void starpu_bound_compute (double *res, double *integer_res, int integer) [Function]
    Get theoretical upper bound (in ms) (needs glpk support detected by configure
    script)

void starpu_bound_print_lp (FILE *output) [Function]
    Emit the Linear Programming system on output for the recorded tasks, in the lp
    format

void starpu_bound_print_mps (FILE *output) [Function]
    Emit the Linear Programming system on output for the recorded tasks, in the mps
    format

void starpu_bound_print (FILE *output, int integer) [Function]
    Emit statistics of actual execution vs theoretical upper bound.  integer permits to
    choose between integer solving (which takes a long time but is correct), and relaxed
    solving (which provides an approximate solution).
```

8 Tips and Tricks to know about

8.1 How to initialize a computation library once for each worker?

Some libraries need to be initialized once for each concurrent instance that may run on the machine. For instance, a C++ computation class which is not thread-safe by itself, but for which several instantiated objects of that class can be used concurrently. This can be used in StarPU by initializing one such object per worker. For instance, the libstarpufft example does the following to be able to use FFTW.

Some global array stores the instantiated objects:

```
fftw_plan plan_cpu[STARPU_NMAXWORKERS];
```

At initialisation time of libstarpufft, the objects are initialized:

```
int workerid;
for (workerid = 0; workerid < starpu_worker_get_count(); workerid++) {
    switch (starpu_worker_get_type(workerid)) {
        case STARPU_CPU_WORKER:
            plan_cpu[workerid] = fftw_plan(...);
            break;
    }
}
```

And in the codelet body, they are used:

```
static void fft(void *descr[], void *_args)
{
    int workerid = starpu_worker_get_id();
    fftw_plan plan = plan_cpu[workerid];
    ...

    fftw_execute(plan, ...);
}
```

Another way to go which may be needed is to execute some code from the workers themselves thanks to `starpu_execute_on_each_worker`. This may be required by CUDA to behave properly due to threading issues. For instance, StarPU's `starpu_helper_cublas_init` looks like the following to call `cublasInit` from the workers themselves:

```
static void init_cublas_func(void *args STARPU_ATTRIBUTE_UNUSED)
{
    cublasStatus cublasst = cublasInit();
    cublasSetKernelStream(starpu_cuda_get_local_stream());
}
void starpu_helper_cublas_init(void)
{
    starpu_execute_on_each_worker(init_cublas_func, NULL, STARPU_CUDA);
}
```

9 StarPU MPI support

The integration of MPI transfers within task parallelism is done in a very natural way by the means of asynchronous interactions between the application and StarPU. This is implemented in a separate `libstarpumpi` library which basically provides "StarPU" equivalents of `MPI_*` functions, where `void *` buffers are replaced with `starpu_data_handle_ts`, and all GPU-RAM-NIC transfers are handled efficiently by StarPU-MPI. The user has to use the usual `mpirun` command of the MPI implementation to start StarPU on the different MPI nodes.

An MPI Insert Task function provides an even more seamless transition to a distributed application, by automatically issuing all required data transfers according to the task graph and an application-provided distribution.

9.1 The API

9.1.1 Compilation

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

```
% pkg-config --cflags libstarpumpi-1.0 # options for the compiler
% pkg-config --libs libstarpumpi-1.0   # options for the linker
```

Also pass the `--static` option if the application is to be linked statically.

9.1.2 Initialisation

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

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

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

9.1.3 Communication

int starpu_mpi_send (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm) [Function]

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

```

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

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

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

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

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

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

int starpu_mpi_barrier (MPI_Comm comm)                           [Function]

int starpu_mpi_isend_detached_unlock_tag (starpu_data_handle_t  [Function]
                                           data_handle, int dest, int mpi_tag, MPI_Comm comm, starpu_tag_t tag)
    When the transfer is completed, the tag is unlocked

int starpu_mpi_irecv_detached_unlock_tag (starpu_data_handle_t  [Function]
                                           data_handle, int source, int mpi_tag, MPI_Comm comm, starpu_tag_t
                                           tag)

int starpu_mpi_isend_array_detached_unlock_tag (unsigned         [Function]
                                                 array_size, starpu_data_handle_t *data_handle, int *dest, int
                                                 *mpi_tag, MPI_Comm *comm, starpu_tag_t tag)
    Asynchronously send an array of buffers, and unlocks the tag once all of them are
    transmitted.

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

```


9.2 Simple Example

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

    task->cl = &increment_cl;
    task->handles[0] = token_handle;

    starpu_task_submit(task);
}
```

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

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

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

    unsigned nloops = NITER;
    unsigned loop;

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

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

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

    increment_token();

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

starpu_task_wait_for_all();
```

```
starpu_mpi_shutdown();
starpu_shutdown();

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

9.3 MPI Insert Task Utility

To save the programmer from having to explicit all communications, StarPU provides an "MPI Insert Task Utility". The principle is that the application decides a distribution of the data over the MPI nodes by allocating it and notifying StarPU of that decision, i.e. tell StarPU which MPI node "owns" which data. All MPI nodes then process the whole task graph, and StarPU automatically determines which node actually execute which task, as well as the required MPI transfers.

int starpu_data_set_tag (*starpu_data_handle_t handle*, *int tag*) [Function]
Tell StarPU-MPI which MPI tag to use when exchanging the data.

int starpu_data_get_tag (*starpu_data_handle_t handle*) [Function]
Returns the MPI tag to be used when exchanging the data.

int starpu_data_set_rank (*starpu_data_handle_t handle*, *int mpi_rank*) [Function]
Tell StarPU-MPI which MPI node "owns" a given data, that is, the node which will always keep an up-to-date value, and will by default execute tasks which write to it.

int starpu_data_get_rank (*starpu_data_handle_t handle*) [Function]
Returns the last value set by **starpu_data_set_rank**.

void starpu_mpi_insert_task (*MPI_Comm comm*, *struct starpu_codelet *cl*, ...) [Function]
Create and submit a task corresponding to *cl* with the following arguments. The argument list must be zero-terminated.

The arguments following the codelets are the same types as for the function **starpu_insert_task** defined in [Section 5.7 \[Insert Task Utility\], page 28](#). The extra argument **STARPU_EXECUTE_ON_NODE** followed by an integer allows to specify the MPI node to execute the codelet. It is also possible to specify that the node owning a specific data will execute the codelet, by using **STARPU_EXECUTE_ON_DATA** followed by a data handle.

The internal algorithm is as follows:

1. Find out whether we (as an MPI node) are to execute the codelet because we own the data to be written to. If different nodes own data to be written to, the argument **STARPU_EXECUTE_ON_NODE** or **STARPU_EXECUTE_ON_DATA** has to be used to specify which MPI node will execute the task.
2. Send and receive data as requested. Nodes owning data which need to be read by the task are sending them to the MPI node which will execute it. The latter receives them.
3. Execute the codelet. This is done by the MPI node selected in the 1st step of the algorithm.
4. In the case when different MPI nodes own data to be written to, send written data back to their owners.

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

```
void starpu_mpi_get_data_on_node (MPI_Comm comm, [Function]
    starpu_data_handle_t data_handle, int node)
    todo
```

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

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

    // /* Other examples useful for other kinds of computations */
    // /* / distrib */
    // return (x+y) % nb_nodes;

    // /* Block cyclic distrib */
    // unsigned side = sqrt(nb_nodes);
    // return x % side + (y % side) * size;
}
```

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

One can note an optimization here (the `else if` test): we only register data which will be needed by the tasks that we will execute.

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

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

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

```

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

starpu_task_wait_for_all();

```

I.e. all MPI nodes process the whole task graph, but as mentioned above, for each task, only the MPI node which owns the data being written to (here, `data_handles[x][y]`) will actually run the task. The other MPI nodes will automatically send the required data.

9.4 MPI Collective Operations

`int starpu_mpi_scatter_detached` (*starpu_data_handle_t* [Function]
**data_handles, int count, int root, MPI_Comm comm*)

Scatter data among processes of the communicator based on the ownership of the data. For each data of the array *data_handles*, the process *root* sends the data to the process owning this data. Processes receiving data must have valid data handles to receive them.

`int starpu_mpi_gather_detached` (*starpu_data_handle_t* [Function]
**data_handles, int count, int root, MPI_Comm comm*)

Gather data from the different processes of the communicator onto the process *root*. Each process owning data handle in the array *data_handles* will send them to the process *root*. The process *root* must have valid data handles to receive the data.

```

if (rank == root)
{
    /* Allocate the vector */
    vector = malloc(nblocks * sizeof(float *));
    for(x=0 ; x<nblocks ; x++)
    {
        starpu_malloc((void **)&vector[x], block_size*sizeof(float));
    }
}

/* Allocate data handles and register data to StarPU */
data_handles = malloc(nblocks*sizeof(starpu_data_handle_t *));
for(x = 0; x < nblocks ; x++)
{
    int mpi_rank = my_distrib(x, nodes);
    if (rank == root) {
        starpu_vector_data_register(&data_handles[x], 0, (uintptr_t)vector[x],
                                   block_size, sizeof(float));
    }
    else if ((mpi_rank == rank) || ((rank == mpi_rank+1 || rank == mpi_rank-1))) {
        /* I own that index, or i will need it for my computations */
        starpu_vector_data_register(&data_handles[x], -1, (uintptr_t)NULL,
                                   block_size, sizeof(float));
    }
    else {
        /* I know it's useless to allocate anything for this */
        data_handles[x] = NULL;
    }
    if (data_handles[x]) {
        starpu_data_set_rank(data_handles[x], mpi_rank);
    }
}

/* Scatter the matrix among the nodes */
starpu_mpi_scatter_detached(data_handles, nblocks, root, MPI_COMM_WORLD);

/* Calculation */
for(x = 0; x < nblocks ; x++) {
    if (data_handles[x]) {
        int owner = starpu_data_get_rank(data_handles[x]);
        if (owner == rank) {
            starpu_insert_task(&cl, STARPU_RW, data_handles[x], 0);
        }
    }
}

/* Gather the matrix on main node */
starpu_mpi_gather_detached(data_handles, nblocks, 0, MPI_COMM_WORLD);

```

10 StarPU FFT support

StarPU provides `libstarpuffft`, a library whose design is very similar to both `fftw` and `cufft`, the difference being that it takes benefit from both CPUs and GPUs. It should however be noted that GPUs do not have the same precision as CPUs, so the results may differ by a negligible amount

float, double and long double precisions are available, with the `fftw` naming convention:

1. double precision structures and functions are named e.g. `starpuffft_execute`
2. float precision structures and functions are named e.g. `starpufftf_execute`
3. long double precision structures and functions are named e.g. `starpuffftl_execute`

The documentation below uses names for double precision, replace `starpuffft_` with `starpufftf_` or `starpuffftl_` as appropriate.

Only complex numbers are supported at the moment.

The application has to call `starpu_init` before calling `starpuffft` functions.

Either main memory pointers or data handles can be provided.

1. To provide main memory pointers, use `starpuffft_start` or `starpuffft_execute`. Only one FFT can be performed at a time, because StarPU will have to register the data on the fly. In the `starpuffft_start` case, `starpuffft_cleanup` needs to be called to unregister the data.
2. To provide data handles (which is preferable), use `starpuffft_start_handle` (preferred) or `starpuffft_execute_handle`. Several FFTs Several FFT tasks can be submitted for a given plan, which permits e.g. to start a series of FFT with just one plan. `starpuffft_start_handle` is preferable since it does not wait for the task completion, and thus permits to enqueue a series of tasks.

10.0.1 Compilation

The flags required to compile or link against the FFT library are accessible with the following commands:

```
% pkg-config --cflags libstarpuffft-1.0 # options for the compiler
% pkg-config --libs libstarpuffft-1.0   # options for the linker
```

Also pass the `--static` option if the application is to be linked statically.

10.0.2 Initialisation

`void * starpufft_malloc (size_t n)` [Function]

Allocates memory for n bytes. This is preferred over `malloc`, since it allocates pinned memory, which allows overlapped transfers.

`void * starpufft_free (void *p)` [Function]

Release memory previously allocated.

`struct starpufft_plan * starpufft_plan_dft_1d (int n, int sign, unsigned flags)` [Function]

Initializes a plan for 1D FFT of size n . `sign` can be `STARPUFFT_FORWARD` or `STARPUFFT_INVERSE`. `flags` must be 0.

struct starpufft_plan * starpufft_plan_dft_2d (*int n, int m,* [Function]
int sign, unsigned flags)

Initializes a plan for 2D FFT of size (*n, m*). *sign* can be `STARPUFFT_FORWARD` or `STARPUFFT_INVERSE`. *flags* must be 0.

struct starpu_task * starpufft_start (*starpufft_plan p, void *in,* [Function]
*void *out*)

Start an FFT previously planned as *p*, using *in* and *out* as input and output. This only submits the task and does not wait for it. The application should call `starpufft_cleanup` to unregister the data.

struct starpu_task * starpufft_start_handle (*starpufft_plan p,* [Function]
starpu_data_handle_t in, starpu_data_handle_t out)

Start an FFT previously planned as *p*, using data handles *in* and *out* as input and output (assumed to be vectors of elements of the expected types). This only submits the task and does not wait for it.

void starpufft_execute (*starpufft_plan p, void *in, void *out*) [Function]

Execute an FFT previously planned as *p*, using *in* and *out* as input and output. This submits and waits for the task.

void starpufft_execute_handle (*starpufft_plan p,* [Function]
starpu_data_handle_t in, starpu_data_handle_t out)

Execute an FFT previously planned as *p*, using data handles *in* and *out* as input and output (assumed to be vectors of elements of the expected types). This submits and waits for the task.

void starpufft_cleanup (*starpufft_plan p*) [Function]

Releases data for plan *p*, in the `starpufft_start` case.

void starpufft_destroy_plan (*starpufft_plan p*) [Function]

Destroys plan *p*, i.e. release all CPU (fftw) and GPU (cufft) resources.

11 C Extensions

If the GCC used to compile StarPU provides a plug-in support¹, StarPU will then build a plug-in for the GNU Compiler Collection (GCC), which defines extensions to languages of the C family (C, C++, Objective-C) that make it easier to write StarPU code.

The plug-in can be disabled by configuring with `--disable-gcc-extensions`.

Those extensions include syntactic sugar for defining tasks and their implementations, invoking a task, and manipulating data buffers. Use of these extensions can be made conditional on the availability of the plug-in, leading to valid C sequential code when the plug-in is not used (see [Section 11.3 \[Conditional Extensions\]](#), page 60).

This section does not require detailed knowledge of the StarPU library.

Note: as of StarPU 1.0.0rc1, this is still an area under development and subject to change.

11.1 Defining Tasks

The StarPU GCC plug-in views *tasks* as “extended” C functions:

1. tasks may have several implementations—e.g., one for CPUs, one written in OpenCL, one written in CUDA;
2. tasks may have several implementations of the same target—e.g., several CPU implementations;
3. when a task is invoked, it may run in parallel, and StarPU is free to choose any of its implementations.

Tasks and their implementations must be *declared*. These declarations are annotated with *attributes* (see [Section “Attribute Syntax” in Using the GNU Compiler Collection \(GCC\)](#)): the declaration of a task is a regular C function declaration with an additional `task` attribute, and task implementations are declared with a `task_implementation` attribute.

The following function attributes are provided:

task Declare the given function as a StarPU task. Its return type must be `void`, and it must not be defined—instead, a definition will automatically be provided by the compiler.

Under the hood, declaring a task leads to the declaration of the corresponding `codelet` (see [Section 1.2.1 \[Codelet and Tasks\]](#), page 3). If one or more task implementations are declared in the same compilation unit, then the codelet and the function itself are also defined; they inherit the scope of the task.

Scalar arguments to the task are passed by value and copied to the target device if need be—technically, they are passed as the `cl_arg` buffer (see [Section 13.6 \[Codelets and Tasks\]](#), page 82).

Pointer arguments are assumed to be registered data buffers—the `buffers` argument of a task (see [Section 13.6 \[Codelets and Tasks\]](#), page 82); `const`-qualified pointer arguments are viewed as read-only buffers (`STARPU_R`), and non-`const`-qualified buffers are assumed to be used read-write (`STARPU_RW`). In

¹ This feature is only available for GCC 4.5 and later.

addition, the `output` type attribute can be as a type qualifier for output pointer or array parameters (`STARPU_W`).

`task_implementation (target, task)`

Declare the given function as an implementation of *task* to run on *target*. *target* must be a string, currently one of "cpu", "opencl", or "cuda".

Here is an example:

```
#define __output __attribute__ ((output))

static void matmul (const float *A, const float *B,
                   __output float *C,
                   size_t nx, size_t ny, size_t nz)
    __attribute__ ((task));

static void matmul_cpu (const float *A, const float *B,
                       __output float *C,
                       size_t nx, size_t ny, size_t nz)
    __attribute__ ((task_implementation ("cpu", matmul)));

static void
matmul_cpu (const float *A, const float *B, __output float *C,
            size_t nx, size_t ny, size_t nz)
{
    size_t i, j, k;

    for (j = 0; j < ny; j++)
        for (i = 0; i < nx; i++)
        {
            for (k = 0; k < nz; k++)
                C[j * nx + i] += A[j * nz + k] * B[k * nx + i];
        }
}
```

A `matmult` task is defined; it has only one implementation, `matmult_cpu`, which runs on the CPU. Variables *A* and *B* are input buffers, whereas *C* is considered an input/output buffer.

CUDA and OpenCL implementations can be declared in a similar way:

```
static void matmul_cuda (const float *A, const float *B, float *C,
                        size_t nx, size_t ny, size_t nz)
    __attribute__ ((task_implementation ("cuda", matmul)));

static void matmul_opencl (const float *A, const float *B, float *C,
                           size_t nx, size_t ny, size_t nz)
    __attribute__ ((task_implementation ("opencl", matmul)));
```

The CUDA and OpenCL implementations typically either invoke a kernel written in CUDA or OpenCL (for similar code, see [Section A.3 \[CUDA Kernel\]](#), page 114, and see [Section A.4 \[OpenCL Kernel\]](#), page 114), or call a library function that uses CUDA or OpenCL under the hood, such as CUBLAS functions:

```
static void
matmul_cuda (const float *A, const float *B, float *C,
              size_t nx, size_t ny, size_t nz)
{
    cublasSgemm ('n', 'n', nx, ny, nz,
                 1.0f, A, 0, B, 0,
                 0.0f, C, 0);
    cudaStreamSynchronize (starpu_cuda_get_local_stream ());
}
```

A task can be invoked like a regular C function:

```
matmul (&A[i * zdim * bydim + k * bzdime * bydim],
        &B[k * xdim * bzdime + j * bxdime * bzdime],
        &C[i * xdim * bydim + j * bxdime * bydim],
        bxdime, bydim, bzdime);
```

This leads to an *asynchronous invocation*, whereby `matmult`'s implementation may run in parallel with the continuation of the caller.

The next section describes how memory buffers must be handled in StarPU-GCC code.

11.2 Registered Data Buffers

Data buffers such as matrices and vectors that are to be passed to tasks must be *registered*. Registration allows StarPU to handle data transfers among devices—e.g., transferring an input buffer from the CPU's main memory to a task scheduled to run a GPU (see [Section 1.2.2 \[StarPU Data Management Library\]](#), page 4).

The following pragmas are provided:

#pragma starpu register *ptr* [*size*]

Register *ptr* as a *size*-element buffer. When *ptr* has an array type whose size is known, *size* may be omitted.

#pragma starpu unregister *ptr*

Unregister the previously-registered memory area pointed to by *ptr*. As a side-effect, *ptr* points to a valid copy in main memory.

#pragma starpu acquire *ptr*

Acquire in main memory an up-to-date copy of the previously-registered memory area pointed to by *ptr*, for read-write access.

#pragma starpu release *ptr*

Release the previously-register memory area pointed to by *ptr*, making it available to the tasks.

As a substitute for the `register` and `unregister` pragmas, the `heap_allocated` variable attribute offers a higher-level mechanism:

heap_allocated

This attributes applies to local variables with an array type. Its effect is to automatically allocate and register the array's storage on the heap, using `starpu_malloc` under the hood (see [Section 13.3.2 \[Basic Data Library API\]](#), page 69).

The heap-allocated array is automatically freed and unregistered when the variable’s scope is left, as with automatic variables².

The following example illustrates use of the `heap_allocated` attribute:

```
extern void cholesky(unsigned nblocks, unsigned size,
                    float mat[nblocks][nblocks][size])
    __attribute__((task));

int
main (int argc, char *argv[])
{
    #pragma starpu initialize

    /* ... */

    int nblocks, size;
    parse_args (&nblocks, &size);

    /* Allocate an array of the required size on the heap,
       and register it. */

    float matrix[nblocks][nblocks][size]
        __attribute__((heap_allocated));

    cholesky (nblocks, size, matrix);

    #pragma starpu shutdown

    /* MATRIX is automatically freed upon return. */

    return EXIT_SUCCESS;
}
```

11.3 Using C Extensions Conditionally

The C extensions described in this chapter are only available when GCC and its StarPU plug-in are in use. Yet, it is possible to make use of these extensions when they are available—leading to hybrid CPU/GPU code—and discard them when they are not available—leading to valid sequential code.

To that end, the GCC plug-in defines a C preprocessor macro when it is being used:

STARPU_GCC_PLUGIN [Macro]
 Defined for code being compiled with the StarPU GCC plug-in. When defined, this macro expands to an integer denoting the version of the supported C extensions.

² This is achieved by using the `cleanup` attribute (see Section “Variable Attributes” in *Using the GNU Compiler Collection (GCC)*)

The code below illustrates how to define a task and its implementations in a way that allows it to be compiled without the GCC plug-in:

```

/* The macros below abstract over the attributes specific to
   StarPU-GCC and the name of the CPU implementation. */
#ifndef STARPU_GCC_PLUGIN
# define __task __attribute__((task))
# define CPU_TASK_IMPL(task) task ## _cpu
#else
# define __task
# define CPU_TASK_IMPL(task) task
#endif

#include <stdlib.h>

static void matmul (const float *A, const float *B, float *C,
                    size_t nx, size_t ny, size_t nz) __task;

#ifdef STARPU_GCC_PLUGIN

static void matmul_cpu (const float *A, const float *B, float *C,
                        size_t nx, size_t ny, size_t nz)
    __attribute__((task_implementation("cpu", matmul)));

#endif

static void
CPU_TASK_IMPL (matmul) (const float *A, const float *B, float *C,
                        size_t nx, size_t ny, size_t nz)
{
    /* Code of the CPU kernel here... */
}

int
main (int argc, char *argv[])
{
    /* The pragmas below are simply ignored when StarPU-GCC
       is not used. */
    #pragma starpu initialize

    float A[123][42][7], B[123][42][7], C[123][42][7];

    #pragma starpu register A
    #pragma starpu register B
    #pragma starpu register C

    /* When StarPU-GCC is used, the call below is asynchronous;
       otherwise, it is synchronous. */
    matmul (A, B, C, 123, 42, 7);

    #pragma starpu wait
    #pragma starpu shutdown

    return EXIT_SUCCESS;
}

```

Note that attributes such as `task` are simply ignored by GCC when the StarPU plugin is not loaded, so the `__task` macro could be omitted altogether. However, `gcc -Wall` emits a warning for unknown attributes, which can be inconvenient, and other compilers may be unable to parse the attribute syntax. Thus, using macros such as `__task` above is recommended.

12 SOCL OpenCL Extensions

SOCL is a extension that aims at implementing the OpenCL standard on top of StarPU. It allows to gives a (relatively) clean and standardized API to StarPU. By allowing OpenCL applications to use StarPU transparently, it provides users with the latest StarPU enhancements without any further development, and allows these OpenCL applications to easily fall back to another OpenCL implementation.

This section does not require detailed knowledge of the StarPU library.

Note: as of StarPU 1.0.0rc1, this is still an area under development and subject to change.

TODO

13 StarPU Basic API

13.1 Initialization and Termination

int starpu_init (*struct starpu_conf *conf*) [Function]

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.

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

struct starpu_conf [Data type]

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.

sched_policy_name (default = NULL)

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

sched_policy (default = NULL)

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

ncpus (default = -1)

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

ncuda (default = -1)

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

nopencl (default = -1)

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

nspus (default = -1)

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

use_explicit_workers_bindid (default = 0)

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

`workers_bindid[STARPU_NMAXWORKERS]`

If the `use_explicit_workers_bindid` flag is set, this array indicates where to bind the different workers. The *i*-th entry of the `workers_bindid` indicates the logical identifier of the processor which should execute the *i*-th worker. Note that the logical ordering of the CPUs is either determined by the OS, or provided by the `hwloc` library in case it is available. When this flag is set, the [Section 15.2.1.5 \[STARPU_WORKERS_CPUID\]](#), [page 108](#) 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 15.2.1.6 \[STARPU_WORKERS_CUDAID\]](#), [page 109](#) 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_opengl_gpuid` (default = 0)

If this flag is set, the OpenCL workers will be attached to the OpenCL devices specified in the `workers_opengl_gpuid` array. Otherwise, StarPU affects the OpenCL devices in a round-robin fashion.

`workers_opengl_gpuid[STARPU_NMAXWORKERS]`

todo

`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.

`single_combined_worker` (default = 0)

By default, StarPU creates various combined workers according to the machine structure. Some parallel libraries (e.g. most OpenMP implementations) however do not support concurrent calls to parallel code. In such case, setting this flag makes StarPU only create one combined worker, containing all the CPU workers. The default value is overwritten by the `STARPU_SINGLE_COMBINED_WORKER` environment variable when it is set.

`int starpu_conf_init (struct starpu_conf *conf)` [Function]

This function initializes the `conf` structure passed as argument with the default values. In case some configuration parameters are already specified through environment variables, `starpu_conf_init` initializes the fields of the structure according to the environment variables. For instance if `STARPU_CALIBRATE` is set, its value is put in the `.ncuda` field of the structure passed as argument.

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

void starpu_shutdown (void) [Function]

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

13.2 Workers' Properties

enum starpu_archtype [DataType]

The different values are:

STARPU_CPU_WORKER
 STARPU_CUDA_WORKER
 STARPU_OPENCL_WORKER
 STARPU_GORDON_WORKER

unsigned starpu_worker_get_count (void) [Function]

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

int starpu_worker_get_count_by_type (enum starpu_archtype type) [Function]

Returns the number of workers of the given type indicated by the argument. A positive (or null) value is returned in case of success, `-EINVAL` indicates that the type is not valid otherwise.

unsigned starpu_cpu_worker_get_count (void) [Function]

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

unsigned starpu_cuda_worker_get_count (void) [Function]

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

unsigned starpu_opencl_worker_get_count (void) [Function]

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

unsigned starpu_spu_worker_get_count (void) [Function]

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

int starpu_worker_get_id (void) [Function]

This function returns the identifier of the current worker, i.e the one 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`.

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

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

int starpu_worker_get_devid (*int id*) [Function]

This function returns the device id of the given worker. The worker should be identified with the value 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.

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

This function returns the type of processing unit associated to a worker. The worker identifier is a value 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.

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

This function allows to get the name of a given worker. 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.

unsigned starpu_worker_get_memory_node (*unsigned workerid*) [Function]

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

13.3 Data Library

This section describes the data management facilities provided by StarPU.

We show how to use existing data interfaces in [Section 13.4 \[Data Interfaces\]](#), [page 72](#), but developers can design their own data interfaces if required.

13.3.1 Introduction

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 specific 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 13.4 \[Data Interfaces\]](#), page 72.

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.

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).

13.3.2 Basic Data Library API

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

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

int starpu_free (void *A) [Function]

This function frees memory which has previously allocated with `starpu_malloc`.

enum starpu_access_mode [Data Type]

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

`STARPU_R`: read-only mode.

`STARPU_W`: write-only mode.

`STARPU_RW`: read-write mode. This is equivalent to `STARPU_R|STARPU_W`.

`STARPU_SCRATCH`: scratch memory. A temporary buffer is allocated for the task, but StarPU does not enforce data consistency, i.e. each device has its own buffer, independently from each other (even for CPUs). This is useful for temporary variables. For now, no behaviour is defined concerning the relation with `STARPU_R/W` modes and the value provided at registration, i.e. the value of the scratch buffer is undefined at entry of the codelet function, but this is being considered for future extensions.

`STARPU_REDUX` reduction mode.

starpu_data_handle_t [Data Type]

StarPU uses `starpu_data_handle_t` 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_t` 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.

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

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 [struct starpu_data_interface_ops], page 97 for more details on this structure.

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

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

```
void starpu_data_unregister (starpu_data_handle_t handle)          [Function]
```

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

```
void starpu_data_unregister_no_coherency (starpu_data_handle_t     [Function]
                                          handle)
```

This is the same as `starpu_data_unregister`, except that StarPU does not put back a valid copy into the home node, in the buffer that was initially registered.

```
void starpu_data_invalidate (starpu_data_handle_t handle)         [Function]
```

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

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

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

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

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

```
starpu_data_handle_t starpu_data_lookup (const void *ptr)         [Function]
```

Return the handle associated to ptr *ptr*.

```
int starpu_data_request_allocation (starpu_data_handle_t handle, uint32_t node) [Function]
    todo
```

```
void starpu_data_query_status (starpu_data_handle_t handle, int memory_node, int *is_allocated, int *is_valid, int *is_requested) [Function]
    Query the status of the handle on the specified memory node.
```

```
void starpu_data_advise_as_important (starpu_data_handle_t handle, unsigned is_important) [Function]
    This function allows to specify that a piece of data can be discarded without impacting the application.
```

```
void starpu_data_set_reduction_methods (starpu_data_handle_t handle, struct starpu_codelet *redux_cl, struct starpu_codelet *init_cl) [Function]
    todo
```

13.3.3 Access registered data from the application

```
int starpu_data_acquire (starpu_data_handle_t handle, enum starpu_access_mode mode) [Function]
```

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

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

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

STARPU_DATA_ACQUIRE_CB (*starpu_data_handle_t* **handle**, *enum* *starpu_access_mode* **mode**, *code*) [Macro]

STARPU_DATA_ACQUIRE_CB is the same as `starpu_data_acquire_cb`, except that the code to be executed in a callback is directly provided as a macro parameter, and the data handle is automatically released after it. This permits to easily execute code which depends on the value of some registered data. This is non-blocking too and may be called from task callbacks.

void starpu_data_release (*starpu_data_handle_t* **handle**) [Function]
This function releases the piece of data acquired by the application either by `starpu_data_acquire` or by `starpu_data_acquire_cb`.

13.4 Data Interfaces

13.4.1 Registering Data

There are several ways to register a memory region so that it can be managed by StarPU. The functions below allow the registration of vectors, 2D matrices, 3D matrices as well as BCSR and CSR sparse matrices.

void starpu_void_data_register (*starpu_data_handle_t* ***handle**) [Function]
Register a void interface. There is no data really associated to that interface, but it may be used as a synchronization mechanism. It also permits to express an abstract piece of data that is managed by the application internally: this makes it possible to forbid the concurrent execution of different tasks accessing the same "void" data in read-write concurrently.

void starpu_variable_data_register (*starpu_data_handle_t* ***handle**, *uint32_t* **home_node**, *uintptr_t* **ptr**, *size_t* **size**) [Function]
Register the *size*-byte element pointed to by *ptr*, which is typically a scalar, and initialize *handle* to represent this data item.

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

void starpu_vector_data_register (*starpu_data_handle_t* ***handle**, *uint32_t* **home_node**, *uintptr_t* **ptr**, *uint32_t* **count**, *size_t* **size**) [Function]
Register the *count* *size*-byte elements pointed to by *ptr* and initialize *handle* to represent it.

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



```
void starpu_matrix_data_register (starpu_data_handle_t *handle, [Function]
    uint32_t home_node, uintptr_t ptr, uint32_t ld, uint32_t nx, uint32_t ny,
    size_t size)
```

Register the $n \times n \times ny$ 2D matrix of *size*-byte elements pointed by *ptr* and initialize *handle* to represent it. *ld* specifies the number of extra elements present at the end of each row; a non-zero *ld* adds padding, which can be useful for alignment purposes.

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

```
void starpu_block_data_register (starpu_data_handle_t *handle, [Function]
    uint32_t home_node, uintptr_t ptr, uint32_t ldy, uint32_t ldz, uint32_t nx,
    uint32_t ny, uint32_t nz, size_t size)
```

Register the $n \times n \times ny \times nz$ 3D matrix of *size*-byte elements pointed by *ptr* and initialize *handle* to represent it. Again, *ldy* and *ldz* specify the number of extra elements present at the end of each row or column.

```
float *block;
starpu_data_handle_t 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));
```

```
void starpu_bcsr_data_register (starpu_data_handle_t *handle, [Function]
    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)
```

This variant of `starpu_data_register` uses the BCSR (Blocked Compressed Sparse Row Representation) sparse matrix interface. TODO

```
void starpu_csr_data_register (starpu_data_handle_t *handle, [Function]
    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)
```

This variant of `starpu_data_register` uses the CSR (Compressed Sparse Row Representation) sparse matrix interface. TODO

```
void * starpu_data_get_interface_on_node (starpu_data_handle_t [Function]
    handle, unsigned memory_node)
```

Return the interface associated with *handle* on *memory_node*.

13.4.2 Accessing Data Interfaces

Each data interface is provided with a set of field access functions. The ones using a `void *` parameter aimed to be used in codelet implementations (see for example the code in [Section 4.3.1 \[Source code of Vector Scaling\]](#), [page 12](#)).

`enum starpu_data_interface_id` [Data Type]

The different values are:

`STARPU_MATRIX_INTERFACE_ID`
`STARPU_BLOCK_INTERFACE_ID`
`STARPU_VECTOR_INTERFACE_ID`
`STARPU_CSR_INTERFACE_ID`
`STARPU_BCSR_INTERFACE_ID`
`STARPU_VARIABLE_INTERFACE_ID`
`STARPU_VOID_INTERFACE_ID`
`STARPU_MULTIFORMAT_INTERFACE_ID`
`STARPU_NINTERFACES_ID` : number of data interfaces

13.4.2.1 Handle

`void * starpu_handle_to_pointer (starpu_data_handle_t handle, [Function]
 uint32_t node)`

Return the pointer associated with *handle* on node *node* or NULL if *handle*'s interface does not support this operation or data for this handle is not allocated on that node.

`void * starpu_handle_get_local_ptr (starpu_data_handle_t [Function]
 handle)`

Return the local pointer associated with *handle* or NULL if *handle*'s interface does not have data allocated locally

`enum starpu_data_interface_id [Function]
 starpu_get_handle_interface_id (starpu_data_handle_t handle)`

Return the unique identifier of the interface associated with the given *handle*.

13.4.2.2 Variable Data Interfaces

`size_t starpu_variable_get_elemsize (starpu_data_handle_t [Function]
 handle)`

Return the size of the variable designated by *handle*.

`uintptr_t starpu_variable_get_local_ptr (starpu_data_handle_t [Function]
 handle)`

Return a pointer to the variable designated by *handle*.

`STARPU_VARIABLE_GET_PTR (void *interface) [Macro]`

Return a pointer to the variable designated by *interface*.

`STARPU_VARIABLE_GET_ELEMSIZE (void *interface) [Macro]`

Return the size of the variable designated by *interface*.

13.4.2.3 Vector Data Interfaces

`uint32_t starpu_vector_get_nx (starpu_data_handle_t handle) [Function]`

Return the number of elements registered into the array designated by *handle*.

`size_t starpu_vector_get_elemsize (starpu_data_handle_t handle) [Function]`

Return the size of each element of the array designated by *handle*.

`uintptr_t starpu_vector_get_local_ptr (starpu_data_handle_t handle)` [Function]

Return the local pointer associated with *handle*.

`STARPU_VECTOR_GET_PTR (void *interface)` [Macro]

Return a pointer to the array designated by *interface*, valid on CPUs and CUDA only. For OpenCL, the device handle and offset need to be used instead.

`STARPU_VECTOR_GET_DEV_HANDLE (void *interface)` [Macro]

Return a device handle for the array designated by *interface*, to be used on OpenCL. the offset documented below has to be used in addition to this.

`STARPU_VECTOR_GET_OFFSET (void *interface)` [Macro]

Return the offset in the array designated by *interface*, to be used with the device handle.

`STARPU_VECTOR_GET_NX (void *interface)` [Macro]

Return the number of elements registered into the array designated by *interface*.

`STARPU_VECTOR_GET_ELEMSIZE (void *interface)` [Macro]

Return the size of each element of the array designated by *interface*.

13.4.2.4 Matrix Data Interfaces

`uint32_t starpu_matrix_get_nx (starpu_data_handle_t handle)` [Function]

Return the number of elements on the x-axis of the matrix designated by *handle*.

`uint32_t starpu_matrix_get_ny (starpu_data_handle_t handle)` [Function]

Return the number of elements on the y-axis of the matrix designated by *handle*.

`uint32_t starpu_matrix_get_local_ld (starpu_data_handle_t handle)` [Function]

Return the number of extra elements present at the end of each row of the matrix designated by *handle*.

`uintptr_t starpu_matrix_get_local_ptr (starpu_data_handle_t handle)` [Function]

Return the local pointer associated with *handle*.

`size_t starpu_matrix_get_elemsize (starpu_data_handle_t handle)` [Function]

Return the size of the elements registered into the matrix designated by *handle*.

`STARPU_MATRIX_GET_PTR (void *interface)` [Macro]

Return a pointer to the matrix designated by *interface*, valid on CPUs and CUDA devices only. For OpenCL devices, the device handle and offset need to be used instead.

`STARPU_MATRIX_GET_DEV_HANDLE (void *interface)` [Macro]

Return a device handle for the matrix designated by *interface*, to be used on OpenCL. The offset documented below has to be used in addition to this.

STARPU_MATRIX_GET_OFFSET (*void *interface*) [Macro]
 Return the offset in the matrix designated by *interface*, to be used with the device handle.

STARPU_MATRIX_GET_NX (*void *interface*) [Macro]
 Return the number of elements on the x-axis of the matrix designated by *interface*.

STARPU_MATRIX_GET_NY (*void *interface*) [Macro]
 Return the number of elements on the y-axis of the matrix designated by *interface*.

STARPU_MATRIX_GET_LD (*void *interface*) [Macro]
 Return the number of extra elements present at the end of each row of the matrix designated by *interface*.

STARPU_MATRIX_GET_ELEMSIZE (*void *interface*) [Macro]
 Return the size of the elements registered into the matrix designated by *interface*.

13.4.2.5 Block Data Interfaces

uint32_t starpu_block_get_nx (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_block_get_ny (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_block_get_nz (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_block_get_local_ldy (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_block_get_local_ldz (*starpu_data_handle_t handle*) [Function]
 todo

uintptr_t starpu_block_get_local_ptr (*starpu_data_handle_t handle*) [Function]
 todo

size_t starpu_block_get_elemsize (*starpu_data_handle_t handle*) [Function]
 todo

STARPU_BLOCK_GET_PTR (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_DEV_HANDLE (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_OFFSET (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_NX (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_NY (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_NZ (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_LDY (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_LDZ (*void *interface*) [Macro]
 todo

STARPU_BLOCK_GET_ELEMSIZE (*void *interface*) [Macro]
 todo

13.4.2.6 BCSR Data Interfaces

uint32_t starpu_bcsr_get_nnz (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_bcsr_get_nrow (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_bcsr_get_firstentry (*starpu_data_handle_t handle*) [Function]
 todo

uintptr_t starpu_bcsr_get_local_nzval (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t * starpu_bcsr_get_local_colind (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t * starpu_bcsr_get_local_rowptr (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_bcsr_get_r (*starpu_data_handle_t handle*) [Function]
 todo

uint32_t starpu_bcsr_get_c (*starpu_data_handle_t handle*) [Function]
 todo

size_t starpu_bcsr_get_elemsize (*starpu_data_handle_t handle*) [Function]
 todo

13.4.2.7 CSR Data Interfaces

`uint32_t starpu_csr_get_nnz (starpu_data_handle_t handle)` [Function]

Return the number of non-zero values in the matrix designated by *handle*.

`uint32_t starpu_csr_get_nrow (starpu_data_handle_t handle)` [Function]

Return the size of the row pointer array of the matrix designated by *handle*.

`uint32_t starpu_csr_get_firstentry (starpu_data_handle_t handle)` [Function]

todo

`uintptr_t starpu_csr_get_local_nzval (starpu_data_handle_t handle)` [Function]

Return a local pointer to the non-zero values of the matrix designated by *handle*.

`uint32_t * starpu_csr_get_local_colind (starpu_data_handle_t handle)` [Function]

Return a local pointer to the column index of the matrix designated by *handle*.

`uint32_t * starpu_csr_get_local_rowptr (starpu_data_handle_t handle)` [Function]

Return a local pointer to the row pointer array of the matrix designated by *handle*.

`size_t starpu_csr_get_elemsize (starpu_data_handle_t handle)` [Function]

Return the size of the elements registered into the matrix designated by *handle*.

`STARPU_CSR_GET_NNZ (void *interface)` [Macro]

Return the number of non-zero values in the matrix designated by *interface*.

`STARPU_CSR_GET_NROW (void *interface)` [Macro]

Return the size of the row pointer array of the matrix designated by *interface*.

`STARPU_CSR_GET_NZVAL (void *interface)` [Macro]

Return a pointer to the non-zero values of the matrix designated by *interface*.

`STARPU_CSR_GET_COLIND (void *interface)` [Macro]

Return a pointer to the column index of the matrix designated by *interface*.

`STARPU_CSR_GET_ROWPTR (void *interface)` [Macro]

Return a pointer to the row pointer array of the matrix designated by *interface*.

`STARPU_CSR_GET_FIRSTENTRY (void *interface)` [Macro]

todo

`STARPU_CSR_GET_ELEMSIZE (void *interface)` [Macro]

Return the size of the elements registered into the matrix designated by *interface*.

13.5 Data Partition

13.5.1 Basic API

struct starpu_data_filter [Data Type]

The filter structure describes a data partitioning operation, to be given to the `starpu_data_partition` function, see [\[starpu_data_partition\]](#), page 79 for an example. The different fields are:

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

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

get_nchildren
This returns the number of children. This can be used instead of `nchildren` when the number of children depends on the actual data (e.g. the number of blocks in a sparse matrix).
`unsigned (*get_nchildren)(struct starpu_data_filter *, starpu_data_handle_t initial_handle);`

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

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

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

void starpu_data_partition (*starpu_data_handle_t initial_handle, struct starpu_data_filter *f*) [Function]

This requests partitioning one StarPU data *initial_handle* into several subdata according to the filter *f*, as shown in the following example:

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

`void starpu_data_unpartition (starpu_data_handle_t root_data, [Function]
uint32_t gathering_node)`

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

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

`int starpu_data_get_nb_children (starpu_data_handle_t handle) [Function]`

This function returns the number of children.

`starpu_data_handle_t starpu_data_get_child [Function]
(starpu_data_handle_t handle, unsigned i)
todo`

`starpu_data_handle_t starpu_data_get_sub_data [Function]
(starpu_data_handle_t root_data, unsigned depth, ...)`

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

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

`starpu_data_handle_t starpu_data_vget_sub_data [Function]
(starpu_data_handle_t root_data, unsigned depth, va_list pa)`

This function is similar to `starpu_data_get_sub_data` but uses a *va_list* for the parameter list.

`void starpu_data_map_filters (starpu_data_handle_t root_data, [Function]
unsigned nfilters, ...)
todo`

`void starpu_data_vmap_filters (starpu_data_handle_t root_data, [Function]
unsigned nfilters, va_list pa)
todo`

13.5.2 Predefined filter functions

This section gives a partial list of the predefined partitioning functions. Examples on how to use them are shown in [Section 5.4 \[Partitioning Data\], page 24](#). The complete list can be found in `starpu_data_filters.h`.

13.5.2.1 Partitioning BCSR Data

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

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

13.5.2.2 Partitioning BLAS interface

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

This partitions a dense Matrix into horizontal blocks.

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

This partitions a dense Matrix into vertical blocks.

13.5.2.3 Partitioning Vector Data

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

Return in **child_interface* the *id*th element of the vector represented by *father_interface* once partitioned in *nparts* chunks of equal size.

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

Return in **child_interface* the *id*th element of the vector represented by *father_interface* once partitioned into *nparts* chunks according to the *filter_arg_ptr* field of **f*.

The *filter_arg_ptr* field must point to an array of *nparts* *uint32_t* elements, each of which specifies the number of elements in each chunk of the partition.

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

Return in **child_interface* the *id*th element of the vector represented by *father_interface* once partitioned in two chunks of equal size, ignoring *nparts*. Thus, *id* must be 0 or 1.

13.5.2.4 Partitioning Block Data

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

This partitions a 3D matrix along the X axis.

13.6 Codelets and Tasks

This section describes the interface to manipulate codelets and tasks.

```
struct starpu_codelet [Data Type]
```

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

where (optional)

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. If the field is unset, its value will be automatically set based on the availability of the **XXX_funcs** fields defined below.

cpu_func (optional)

This field has been made deprecated. One should use instead the **cpu_funcs** field.

cpu_funcs (optional)

Is an array of function pointers to the CPU implementations of the codelet. It must be terminated by a NULL value. The functions 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. If the **where** field is set, then the **cpu_funcs** field is ignored if **STARPU_CPU** does not appear in the **where** field, it must be non-null otherwise.

cuda_func (optional)

This field has been made deprecated. One should use instead the **cuda_funcs** field.

cuda_funcs (optional)

Is an array of function pointers to the CUDA implementations of the codelet. It must be terminated by a NULL value. *The functions must be host-functions written in the CUDA runtime API.* Their prototype must be: `void cuda_func(void *buffers[], void *cl_arg);`. If the **where** field is set, then the **cuda_funcs** field is ignored if **STARPU_CUDA** does not appear in the **where** field, it must be non-null otherwise.

opencl_func (optional)

This field has been made deprecated. One should use instead the **opencl_funcs** field.

openccl_funcs (optional)

Is an array of function pointers to the OpenCL implementations of the codelet. It must be terminated by a NULL value. The functions prototype must be: `void openccl_func(void *buffers[], void *cl_arg);`. If the **where** field is set, then the **openccl_funcs** field is ignored if **STARPU_OPENCL** does not appear in the **where** field, it must be non-null otherwise.

gordon_func (optional)

This field has been made deprecated. One should use instead the **gordon_funcs** field.

gordon_funcs (optional)

Is an array of index of the Cell SPU implementations of the codelet within the Gordon library. It must be terminated by a NULL value. 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 **starpup_task** structure is not counted in this number. This value should not be above **STARPU_NMAXBUFS**.

modes Is an array of `enum starpu_access_mode`. It describes the required access modes to the data needed by the codelet (e.g. **STARPU_RW**). The number of entries in this array must be specified in the **nbuffers** field (defined above), and should not exceed **STARPU_NMAXBUFS**. If unsufficient, this value can be set with the `--enable-maxbuffers` option when configuring StarPU.

model (optional)

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

TODO

power_model (optional)

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

TODO

struct starpu_task

[Data Type]

The **starpup_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 **starpup_task_create** method, or declared statically. In the latter case, the programmer has to zero the **starpup_task** structure and to fill the different fields properly. The indicated default values correspond to the configuration of a task allocated with **starpup_task_create**.

- cl** Is a pointer to the corresponding `struct 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** This field has been made deprecated. One should use instead the **handles** field to specify the handles to the data accessed by the task. The access modes are now defined in the **mode** field of the `struct starpu_codelet` structure.
- handles** Is an array of `starpu_data_handle_t`. It specifies the handles to the different pieces of data accessed by the task. The number of entries in this array must be specified in the **nbuffers** field of the `struct starpu_codelet` structure, and should not exceed `STARPU_NMAXBUFS`. If insufficient, this value can be set with the `--enable-maxbuffers` option when configuring StarPU.
- cl_arg** (optional; default: `NULL`)
This pointer is passed to the codelet through the second argument of the codelet implementation (e.g. `cpu_func` or `cuda_func`). In the specific case of the Cell processor, see the **cl_arg_size** argument.
- cl_arg_size** (optional, Cell-specific)
In the case of the Cell processor, the **cl_arg** pointer is not directly given to the SPU function. A buffer of size **cl_arg_size** is allocated on the SPU. This buffer is then filled with the **cl_arg_size** bytes starting at address **cl_arg**. In this case, the argument given to the SPU codelet is therefore not the **cl_arg** pointer, but the address of the buffer in local store (LS) instead. This field is ignored for CPU, CUDA and OpenCL codelets, where the **cl_arg** pointer is given as such.
- callback_func** (optional) (default: `NULL`)
This is a function pointer of prototype `void (*f)(void *)` which specifies a possible callback. If this pointer is non-null, the callback function is executed *on the host* after the execution of the task. The callback is passed the value contained in the **callback_arg** field. No callback is executed if the field is set to `NULL`.
- callback_arg** (optional) (default: `NULL`)
This is the pointer passed to the callback function. This field is ignored if the **callback_func** is set to `NULL`.
- use_tag** (optional) (default: 0)
If set, this flag indicates that the task should be associated with the tag contained in the **tag_id** field. Tag allow the application to synchronize with the task and to express task dependencies easily.
- tag_id** This fields contains the tag associated to the task if the **use_tag** field was set, it is ignored otherwise.

synchronous

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

priority (optional) (default: `STARPU_DEFAULT_PRIO`)

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

execute_on_a_specific_worker (default: 0)

If this flag is set, StarPU will bypass the scheduler and directly affect this task to the worker specified by the `workerid` field.

workerid (optional)

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

detach (optional) (default: 1)

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

destroy (optional) (default: 1)

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

predicted (output field)

Predicted duration of the task. This field is only set if the scheduling strategy used performance models.

void `starpu_task_init` (*struct `starpu_task`* **task*) [Function]

Initialize *task* with default values. This function is implicitly called by `starpu_task_create`. By default, tasks initialized with `starpu_task_init` must be deinitialized explicitly with `starpu_task_deinit`. Tasks can also be initialized statically, using the constant `STARPU_TASK_INITIALIZER`.

struct starpu_task * starpu_task_create (void) [Function]

Allocate a task structure and initialize it with default values. Tasks allocated dynamically with **starpu_task_create** are automatically freed when the task is terminated. This means that the task pointer can not be used any more once the task is submitted, since it can be executed at any time (unless dependencies make it wait) and thus freed at any time. If the destroy flag is explicitly unset, the resources used by the task have to be freed by calling **starpu_task_destroy**.

void starpu_task_deinit (struct starpu_task *task) [Function]

Release all the structures automatically allocated to execute *task*. This is called automatically by **starpu_task_destroy**, but the task structure itself is not freed. This should be used for statically allocated tasks for instance.

void starpu_task_destroy (struct starpu_task *task) [Function]

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

int starpu_task_wait (struct starpu_task *task) [Function]

This function blocks until *task* has been executed. It is not possible to synchronize with a task more than once. It is not possible to wait for synchronous or detached tasks.

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

int starpu_task_submit (struct starpu_task *task) [Function]

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

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

int starpu_task_wait_for_all (void) [Function]

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

struct starpu_task * starpu_get_current_task (void) [Function]

This function returns the task currently executed by the worker, or NULL if it is called either from a thread that is not a task or simply because there is no task being executed at the moment.

void starpu_display_codelet_stats (struct starpu_codelet *cl) [Function]

Output on **stderr** some statistics on the codelet *cl*.

int starpu_task_wait_for_no_ready (*void*) [Function]
 This function waits until there is no more ready task.

13.7 Explicit Dependencies

void starpu_task_declare_deps_array (*struct starpu_task *task*, [Function]
unsigned ndeps, *struct starpu_task *task_array[]*)

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 be 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.

starpu_tag_t [Data Type]

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

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

void starpu_tag_declare_deps (*starpu_tag_t id*, *unsigned ndeps*, ...) [Function]

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**.

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.

```
/* 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);
```

void starpu_tag_declare_deps_array (*starpu_tag_t id*, *unsigned* [Function]
ndeps, *starpu_tag_t *array*)

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

```
/* 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);
```

void starpu_tag_wait (*starpu_tag_t id*) [Function]

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

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

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

void starpu_tag_remove (*starpu_tag_t id*) [Function]

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

void starpu_tag_notify_from_apps (*starpu_tag_t id*) [Function]

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

13.8 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.

void starpu_data_set_default_sequential_consistency_flag [Function]
(*unsigned flag*)

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

unsigned [Function]
starpu_data_get_default_sequential_consistency_flag (*void*)

Return the default sequential consistency flag

`unsigned` [Function]

`starpu_data_set_default_sequential_consistency_flag` (`void`)

This function returns the current default sequential consistency flag.

`void starpu_data_set_sequential_consistency_flag` [Function]

(`starpu_data_handle_t handle`, `unsigned flag`)

Sets 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`.

13.9 Performance Model API

`enum starpu_perf_archtype` [Data Type]

Enumerates the various types of architectures. CPU types range within `STARPU_CPU_DEFAULT` (1 CPU), `STARPU_CPU_DEFAULT+1` (2 CPUs), ... `STARPU_CPU_DEFAULT + STARPU_MAXCPUS - 1` (`STARPU_MAXCPUS` CPUs). CUDA types range within `STARPU_CUDA_DEFAULT` (GPU number 0), `STARPU_CUDA_DEFAULT + 1` (GPU number 1), ..., `STARPU_CUDA_DEFAULT + STARPU_MAXCUDADEVES - 1` (GPU number `STARPU_MAXCUDADEVES - 1`). OpenCL types range within `STARPU_OPENCL_DEFAULT` (GPU number 0), `STARPU_OPENCL_DEFAULT + 1` (GPU number 1), ..., `STARPU_OPENCL_DEFAULT + STARPU_MAXOPENCLDEVES - 1` (GPU number `STARPU_MAXOPENCLDEVES - 1`).

`STARPU_CPU_DEFAULT`

`STARPU_CUDA_DEFAULT`

`STARPU_OPENCL_DEFAULT`

`STARPU_GORDON_DEFAULT`

`struct starpu_perfmodel` [Data Type]

contains all information about a performance model. At least the `type` and `symbol` fields have to be filled when defining a performance model for a codelet. If not provided, other fields have to be zero.

`type` is the type of performance model. `STARPU_HISTORY_BASED`, `STARPU_REGRESSION_BASED`, `STARPU_NL_REGRESSION_BASED`: No other fields needs to be provided, this is purely history-based. `STARPU_PER_ARCH`: `per_arch` has to be filled with functions which return the cost in micro-seconds. `STARPU_COMMON`: `cost_function` has to be filled with a function that returns the cost in micro-seconds on a CPU, timing on other archs will be determined by multiplying by an arch-specific factor.

`symbol` is the symbol name for the performance model, which will be used as file name to store the model.

`cost_model`

This field is deprecated. Use instead the `cost_function` field.

`cost_function`

Used by `STARPU_COMMON`: takes a task and implementation number, and must return a task duration estimation in micro-seconds.

per_arch Used by STARPU_PER_ARCH: array of struct `starpu_per_arch_perfmodel` structures.

size_base Used by STARPU_HISTORY_BASED and STARPU_*REGRESSION_BASED. If not NULL, takes a task and implementation number, and returns the size to be used as index for history and regression.

struct starpu_per_arch_perfmodel [Data Type]
contains information about the performance model of a given arch.

cost_model
This field is deprecated. Use instead the `cost_function` field.

cost_function
Used by STARPU_PER_ARCH, must point to functions which take a task, the target arch and implementation number (as mere convenience, since the array is already indexed by these), and must return a task duration estimation in micro-seconds.

list Used by STARPU_HISTORY_BASED and STARPU_NL_REGRESSION_BASED, records all execution history measures.

regression
Used by STARPU_HISTORY_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED, contains the estimated factors of the regression.

size_base: Same as in struct `perfmodel`, but per-arch, in case it depends on the architecture-specific implementation.

int starpu_load_history_debug (*const char *symbol, struct starpu_perfmodel *model*) [Function]
loads a given performance model. The *model* structure has to be completely zero, and will be filled with the information saved in `~/.starpu`.

void starpu_perfmodel_debugfilepath (*struct starpu_perfmodel *model, enum starpu_perf_archtype arch, char *path, size_t maxlen*) [Function]
returns the path to the debuginf information for the performance model.

void starpu_perfmodel_get_arch_name (*enum starpu_perf_archtype arch, char *archname, size_t maxlen*) [Function]
returns the architecture name for *arch*.

void starpu_force_bus_sampling (*void*) [Function]
forces sampling the bus performance model again.

enum starpu_perf_archtype starpu_worker_get_perf_archtype (*int workerid*) [Function]
returns the architecture type of a given worker.

int starpu_list_models (*FILE *output*) [Function]
prints a list of all performance models on *output*.

void starpu_print_bus_bandwidth (*FILE *f*) [Function]
prints a matrix of bus bandwidths on *f*.

13.10 Profiling API

int starpu_profiling_status_set (*int status*) [Function]

This 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.

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

int starpu_profiling_status_get (*void*) [Function]

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

void starpu_set_profiling_id (*int new_id*) [Function]

This function sets the ID used for profiling trace filename

struct starpu_task_profiling_info [Data Type]

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. The different fields are:

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.

struct starpu_worker_profiling_info [Data Type]

This structure contains the profiling information associated to a worker. The different fields are:

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.

`int starpu_worker_get_profiling_info (int workerid, struct starpu_worker_profiling_info *worker_info)` [Function]

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*.

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

`struct starpu_bus_profiling_info` [Data Type]

TODO. The different fields are:

`start_time`
TODO

`total_time`
TODO

`transferred_bytes`
TODO

`transfer_count`
TODO

`int starpu_bus_get_profiling_info (int busid, struct starpu_bus_profiling_info *bus_info)` [Function]
todo

`int starpu_bus_get_count (void)` [Function]
TODO

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

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

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

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

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

`void starpu_bus_profiling_helper_display_summary (void)` [Function]
TODO

`void starpu_worker_profiling_helper_display_summary (void)` [Function]
TODO

13.11 CUDA extensions

`cudaStream_t * starpu_cuda_get_local_stream (void)` [Function]

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

`const struct cudaDeviceProp * starpu_cuda_get_device_properties (unsigned workerid)` [Function]

This function returns a pointer to device properties for worker *workerid* (assumed to be a CUDA worker).

`void starpu_helper_cublas_init (void)` [Function]

This function initializes CUBLAS on every CUDA device. 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.

`void starpu_helper_cublas_shutdown (void)` [Function]

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

13.12 OpenCL extensions

13.12.1 Writing OpenCL kernels

`void starpu_opengl_display_error (const char *func, const char *file, int line, const char *msg, cl_int status)` [Function]
todo

`size_t starpu_opengl_get_global_mem_size (int devid)` [Function]
todo

`void starpu_opengl_get_context (int devid, cl_context *context)` [Function]
todo

`void starpu_opengl_get_device (int devid, cl_device_id *device)` [Function]
todo

`void starpu_opengl_get_queue (int devid, cl_command_queue *queue);` [Function]
todo

`void starpu_opengl_get_current_context (cl_context *context)` [Function]
todo

```
void starpu_opengl_get_current_queue (cl_command_queue cl_command_queue
                                     *queue) [Function]
```

todo

```
int starpu_opengl_set_kernel_args (cl_int *err, cl_kernel cl_kernel
                                   *kernel, ...) [Function]
```

Sets the arguments of a given kernel. The list of arguments must be given as (size_t *size_of_the_argument*, cl_mem * *pointer_to_the_argument*). The last argument must be 0. Returns the number of arguments that were successfully set. In case of failure, *err* is set to the error returned by OpenCL.

13.12.2 Compiling OpenCL kernels

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

```
struct starpu_opengl_program [Data Type]
```

todo

```
int starpu_opengl_load_opengl_from_file (char char [Function]
                                         *source_file_name, struct starpu_opengl_program *opengl_programs,
                                         const char* build_options)
```

This function compiles an OpenCL source code stored in a file.

```
int starpu_opengl_load_opengl_from_string (char char [Function]
                                           *opengl_program_source, struct starpu_opengl_program
                                           *opengl_programs, const char* build_options)
```

This function compiles an OpenCL source code stored in a string.

```
int starpu_opengl_unload_opengl (struct starpu_opengl_program [Function]
                                 *opengl_programs)
```

This function unloads an OpenCL compiled code.

13.12.3 Loading OpenCL kernels

```
int starpu_opengl_load_kernel (cl_kernel *kernel, [Function]
                              cl_command_queue *queue, struct starpu_opengl_program
                              *opengl_programs, char *kernel_name, int devid)
```

TODO

```
int starpu_opengl_release_kernel (cl_kernel kernel) [Function]
```

TODO

13.12.4 OpenCL statistics

```
int starpu_opengl_collect_stats (cl_event event) [Function]
```

This function allows to collect statistics on a kernel execution. After termination of the kernels, the OpenCL codelet should call this function to pass it the even returned by

`clEnqueueNDRangeKernel`, to let StarPU collect statistics about the kernel execution (used cycles, consumed power).

13.13 Cell extensions

nothing yet.

13.14 Miscellaneous helpers

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

Copy the content of the *src_handle* into the *dst_handle* handle. The *asynchronous* parameter indicates whether the function should block or not. In the case of an asynchronous call, it is possible to synchronize with the termination of this operation either by the means of implicit dependencies (if enabled) or by calling `starpu_task_wait_for_all()`. If *callback_func* is not NULL, this callback function is executed after the handle has been copied, and it is given the *callback_arg* pointer as argument.

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

This function executes the given function on a subset of workers. 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 `struct 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.

14 StarPU Advanced API

14.1 Defining a new data interface

14.1.1 Data Interface API

struct starpu_data_interface_ops [Data Type]
 Defines the per-interface methods.

```
int {ram,cuda,opencl,spu}_to_{ram,cuda,opencl,spu}(void *src_interface,
unsigned src_node, void *dst_interface, unsigned dst_node);
```

These sixteen functions define how to copy data from the *src_interface* interface on the *src_node* node to the *dst_interface* interface on the *dst_node* node. They return 0 on success.

```
int (*ram_to_cuda_async)(void *src_interface, unsigned src_node, void
*dst_interface, unsigned dst_node, cudaStream_t stream);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (in RAM) to the *dst_interface* interface on the *dst_node* node (on a CUDA device), using the given *stream*. Return 0 on success.

```
int (*cuda_to_ram_async)(void *src_interface, unsigned src_node, void
*dst_interface, unsigned dst_node, cudaStream_t stream);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (on a CUDA device) to the *dst_interface* interface on the *dst_node* node (in RAM), using the given *stream*. Return 0 on success.

```
int (*cuda_to_cuda_async)(void *src_interface, unsigned src_node, void
*dst_interface, unsigned dst_node, cudaStream_t stream);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (on a CUDA device) to the *dst_interface* interface on the *dst_node* node (on another CUDA device), using the given *stream*. Return 0 on success.

```
int (*ram_to_opencl_async)(void *src_interface, unsigned src_node, void
*dst_interface, unsigned dst_node, /* cl_event */ void *event);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (in RAM) to the *dst_interface* interface on the *dst_node* node (on an OpenCL device), using *event*, a pointer to a *cl_event*. Return 0 on success.

```
int (*opencl_to_ram_async)(void *src_interface, unsigned src_node, void
*dst_interface, unsigned dst_node, /* cl_event */ void *event);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (on an OpenCL device) to the *dst_interface* interface on the *dst_node* node (in RAM), using the given *event*, a pointer to a *cl_event*. Return 0 on success.

```
int (*openc1_to_openc1_async)(void *src_interface, unsigned src_node,
void *dst_interface, unsigned dst_node, /* cl_event **/ void *event);
```

Define how to copy data from the *src_interface* interface on the *src_node* node (on an OpenCL device) to the *dst_interface* interface on the *dst_node* node (on another OpenCL device), using the given *event*, a pointer to a *cl_event*. Return 0 on success.

```
struct starpu_data_copy_methods
```

[Data Type]

Per-interface data transfer methods.

```
void (*register_data_handle)(starpu_data_handle_t handle, uint32_t
home_node, void *data_interface);
```

Register an existing interface into a data handle.

```
starpu_ssize_t (*allocate_data_on_node)(void *data_interface, uint32_t
node);
```

Allocate data for the interface on a given node.

```
void (*free_data_on_node)(void *data_interface, uint32_t node);
```

Free data of the interface on a given node.

```
const struct starpu_data_copy_methods *copy_methods;
```

ram/cuda/spu/openc1 synchronous and asynchronous transfer methods.

```
void * (*handle_to_pointer)(starpu_data_handle_t handle, uint32_t
node);
```

Return the current pointer (if any) for the handle on the given node.

```
size_t (*get_size)(starpu_data_handle_t handle);
```

Return an estimation of the size of data, for performance models.

```
uint32_t (*footprint)(starpu_data_handle_t handle);
```

Return a 32bit footprint which characterizes the data size.

```
int (*compare)(void *data_interface_a, void *data_interface_b);
```

Compare the data size of two interfaces.

```
void (*display)(starpu_data_handle_t handle, FILE *f);
```

Dump the sizes of a handle to a file.

```
int (*convert_to_gordon)(void *data_interface, uint64_t *ptr,
gordon_strideSize_t *ss);
```

Convert the data size to the spu size format. If no SPUs are used, this field can be set to NULL.

```
enum starpu_data_interface_id interfaceid;
```

An identifier that is unique to each interface.

```
size_t interface_size;
```

The size of the interface data descriptor.

14.1.2 An example of data interface

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

14.2 Multiformat Data Interface

struct starpu_multiformat_data_interface_ops [Data Type]

todo. The different fields are:

cpu_elemsize
the size of each element on CPUs,

opencl_elemsize
the size of each element on OpenCL devices,

cuda_elemsize
the size of each element on CUDA devices,

cpu_to_opencl_cl
pointer to a codelet which converts from CPU to OpenCL

opencl_to_cpu_cl
pointer to a codelet which converts from OpenCL to CPU

cpu_to_cuda_cl
pointer to a codelet which converts from CPU to CUDA

cuda_to_cpu_cl
pointer to a codelet which converts from CUDA to CPU

void starpu_multiformat_data_register (*starpu_data_handle_t* [Function]
**handle*, *uint32_t home_node*, *void *ptr*, *uint32_t nobjects*, *struct*
*starpu_multiformat_data_interface_ops *format_ops*);

Register a piece of data that can be represented in different ways, depending upon the processing unit that manipulates it. It allows the programmer, for instance, to use an array of structures when working on a CPU, and a structure of arrays when working on a GPU.

nobjects is the number of elements in the data. *format_ops* describes the format.

14.3 Task Bundles

starpu_task_bundle_t [Data Type]

Opaque structure describing a list of tasks that should be scheduled on the same worker whenever it's possible. It must be considered as a hint given to the scheduler as there is no guarantee that they will be executed on the same worker.

void starpu_task_bundle_create (*starpu_task_bundle_t *bundle*) [Function]

Factory function creating and initializing *bundle*, when the call returns, memory needed is allocated and *bundle* is ready to use.

int starpu_task_bundle_insert (*starpu_task_bundle_t bundle*, [Function]
*struct starpu_task *task*)

Insert *task* in *bundle*. Until *task* is removed from *bundle* its expected length and data transfer time will be considered along those of the other tasks of *bundle*. This function mustn't be called if *bundle* is already closed and/or *task* is already submitted.

int starpu_task_bundle_remove (*starpu_task_bundle_t bundle*, [Function]
*struct starpu_task *task*)

Remove *task* from *bundle*. Of course *task* must have been previously inserted *bundle*. This function mustn't be called if *bundle* is already closed and/or *task* is already submitted. Doing so would result in undefined behaviour.

void starpu_task_bundle_close (*starpu_task_bundle_t bundle*); [Function]

Inform the runtime that the user won't modify *bundle* anymore, it means no more inserting or removing task. Thus the runtime can destroy it when possible.

14.4 Task Lists

struct starpu_task_list [Data Type]
 Stores a double-chained list of tasks

void starpu_task_list_init (*struct starpu_task_list *list*) [Function]
 Initialize a list structure

void starpu_task_list_push_front (*struct starpu_task_list *list*, [Function]
*struct starpu_task *task*)
 Push a task at the front of a list

void starpu_task_list_push_back (*struct starpu_task_list *list*, [Function]
*struct starpu_task *task*)
 Push a task at the back of a list

struct starpu_task * starpu_task_list_front (*struct* [Function]
*starpu_task_list *list*)
 Get the front of the list (without removing it)

struct starpu_task * starpu_task_list_back (*struct* [Function]
*starpu_task_list *list*)
 Get the back of the list (without removing it)

int starpu_task_list_empty (*struct starpu_task_list *list*) [Function]
 Test if a list is empty

void starpu_task_list_erase (*struct starpu_task_list *list*, *struct* [Function]
*starpu_task *task*)
 Remove an element from the list

struct starpu_task * starpu_task_list_pop_front (*struct* [Function]
*starpu_task_list *list*)
 Remove the element at the front of the list

struct starpu_task * starpu_task_list_pop_back (*struct* [Function]
*starpu_task_list *list*)
 Remove the element at the back of the list

struct starpu_task * starpu_task_list_begin (*struct* [Function]
*starpu_task_list *list*)
 Get the first task of the list.

```
struct starpu_task * starpu_task_list_end (struct          [Function]
    starpu_task_list *list)
```

Get the end of the list.

```
struct starpu_task * starpu_task_list_next (struct starpu_task  [Function]
    *task)
```

Get the next task of the list. This is not erase-safe.

14.5 Defining a new scheduling policy

TODO

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

14.5.1 Scheduling Policy API

```
struct starpu_machine_topology          [Data Type]
    TODO
```

```
struct starpu_sched_policy             [Data Type]
```

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

```
init_sched
    Initialize the scheduling policy.
```

```
deinit_sched
    Cleanup the scheduling policy.
```

```
push_task
    Insert a task into the scheduler.
```

```
push_task_notify
    Notify the scheduler that a task was pushed on a given worker. This method is called when a task that was explicitly assigned to a worker becomes ready and is about to be executed by the worker. This method therefore permits to keep the state of of the scheduler coherent even when StarPU bypasses the scheduling strategy.
```

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

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

`post_exec_hook` (optional)

This method is called every time a task has been executed.

`policy_name`

Name of the policy (optional).

`policy_description`

Description of the policy (optional).

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

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

`void starpu_sched_set_min_priority (int min_prio) [Function]`

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

`void starpu_sched_set_max_priority (int max_prio) [Function]`

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

`int starpu_sched_get_min_priority (void) [Function]`

Returns the current minimum priority level supported by the scheduling policy

`int starpu_sched_get_max_priority (void) [Function]`

Returns the current maximum priority level supported by the scheduling policy

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

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

`int starpu_worker_can_execute_task (unsigned workerid, struct [Function]
starpu_task *task, unsigned nimpl)`

Check if the worker specified by `workerid` can execute the codelet. Schedulers need to call it before assigning a task to a worker, otherwise the task may fail to execute.

<code>double starpu_timing_now (void)</code>	[Function]
Return the current date in s	
<code>double starpu_task_expected_length (struct starpu_task *task, enum starpu_perf_archtype arch, unsigned nimpl)</code>	[Function]
Returns expected task duration in s	
<code>double starpu_worker_get_relative_speedup (enum starpu_perf_archtype perf_archtype)</code>	[Function]
Returns an estimated speedup factor relative to CPU speed	
<code>double starpu_task_expected_data_transfer_time (uint32_t memory_node, struct starpu_task *task)</code>	[Function]
Returns expected data transfer time in s	
<code>double starpu_data_expected_transfer_time (starpu_data_handle_t handle, unsigned memory_node, enum starpu_access_mode mode)</code>	[Function]
Predict the transfer time (in s) to move a handle to a memory node	
<code>double starpu_task_expected_power (struct starpu_task *task, enum starpu_perf_archtype arch, unsigned nimpl)</code>	[Function]
Returns expected power consumption in J	
<code>double starpu_task_expected_conversion_time (struct starpu_task *task, enum starpu_perf_archtype arch, unsigned nimpl)</code>	[Function]
Returns expected conversion time in ms (multiformat interface only)	

14.5.2 Source code

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

14.6 Expert mode

<code>void starpu_wake_all_blocked_workers (void)</code>	[Function]
todo	
<code>int starpu_progression_hook_register (unsigned (*func)(void *arg), void *arg)</code>	[Function]
todo	

```
void starpu_progression_hook_deregister (int hook_id) [Function]
    todo
```


15 Configuring StarPU

15.1 Compilation configuration

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

15.1.1 Common configuration

15.1.1.1 `--enable-debug`

Enable debugging messages.

15.1.1.2 `--enable-fast`

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

15.1.1.3 `--enable-verbose`

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

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

15.1.1.4 `--enable-coverage`

Enable flags for the `gcov` coverage tool.

15.1.2 Configuring workers

15.1.2.1 `--enable-maxcpus=<number>`

Define the maximum number of CPU cores that StarPU will support, then available as the `STARPU_MAXCPUS` macro.

15.1.2.2 `--disable-cpu`

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

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

Define the maximum number of CUDA devices that StarPU will support, then available as the `STARPU_MAXCUDADEV` macro.

15.1.2.4 `--disable-cuda`

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

15.1.2.5 `--with-cuda-dir=<path>`

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

15.1.2.6 `--with-cuda-include-dir=<path>`

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

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

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

15.1.2.8 --disable-cuda-memcpy-peer

Explicitely disable peer transfers when using CUDA 4.0

15.1.2.9 --enable-maxopencldev=<number>

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

15.1.2.10 --disable-opencl

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

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

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

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

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

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

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

15.1.2.14 --enable-gordon

Enable the use of the Gordon runtime for Cell SPUs.

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

Specify the location of the Gordon SDK.

15.1.2.16 --enable-maximplementations=<number>

Define the number of implementations that can be defined for a single kind of device. It is then available as the `STARPU_MAXIMPLEMENTATIONS` macro.

15.1.3 Advanced configuration**15.1.3.1 --enable-perf-debug**

Enable performance debugging through gprof.

15.1.3.2 --enable-model-debug

Enable performance model debugging.

15.1.3.3 --enable-stats

Enable statistics.

15.1.3.4 --enable-maxbuffers=<nbuffers>

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

15.1.3.5 --enable-allocation-cache

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

15.1.3.6 --enable-opengl-render

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

15.1.3.7 --enable-blas-lib=<name>

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

15.1.3.8 --disable-starpufft

Disable the build of libstarpufft, even if fftw or cuFFT is available.

15.1.3.9 --with-magma=<path>

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

15.1.3.10 --with-fxt=<path>

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

15.1.3.11 --with-perf-model-dir=<dir>

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

15.1.3.12 --with-mpicc=<path to mpicc>

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

15.1.3.13 --with-goto-dir=<dir>

Specify the location of GotoBLAS.

15.1.3.14 --with-atlas-dir=<dir>

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

15.1.3.15 --with-mkl-cflags=<cflags>

Specify the compilation flags for the MKL Library.

15.1.3.16 --with-mkl-ldflags=<ldflags>

Specify the linking flags for the MKL Library. Note that the <http://software.intel.com/en-us/articles/i> website provides a script to determine the linking flags.

15.1.3.17 --disable-gcc-extensions

Disable the GCC plug-in. It is by default enabled if the GCC compiler provides a plug-in support.

15.1.3.18 --disable-socl

Disable the SOCL extension. It is by default enabled if a valid OpenCL installation is found.

15.2 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.

15.2.1 Configuring workers**15.2.1.1 STARPU_NCPUS – Number of CPU workers**

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

15.2.1.2 STARPU_NCUDA – Number of CUDA workers

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

15.2.1.3 STARPU_NOPENCL – Number of OpenCL workers

OpenCL equivalent of the `STARPU_NCUDA` environment variable.

15.2.1.4 STARPU_NGORDON – Number of SPU workers (Cell)

Specify the number of SPUs that StarPU can use.

15.2.1.5 STARPU_WORKERS_CPUID – Bind workers to specific CPUs

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

Note that the first workers correspond to the CUDA workers, then come the OpenCL and the SPU, and finally the CPU workers. For example if we have `STARPU_NCUDA=1`, `STARPU_NOPENCL=1`, `STARPU_NCPUS=2` and `STARPU_WORKERS_CPUID = "0 2 1 3"`, the CUDA

device will be controlled by logical CPU #0, the OpenCL device will be controlled by logical CPU #2, and the logical CPUs #1 and #3 will be used by the CPU workers.

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

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

15.2.1.6 STARPU_WORKERS_CUDAID – Select specific CUDA devices

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.

15.2.1.7 STARPU_WORKERS_OPENCLID – Select specific OpenCL devices

OpenCL equivalent of the `STARPU_WORKERS_CUDAID` environment variable.

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

15.2.2 Configuring the Scheduling engine

15.2.2.1 STARPU_SCHED – Scheduling policy

Choose 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.

15.2.2.2 STARPU_CALIBRATE – Calibrate performance models

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

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

15.2.2.3 STARPU_PREFETCH – Use data prefetch

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

15.2.2.4 STARPU_SCHED_ALPHA – Computation factor

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.

15.2.2.5 STARPU_SCHED_BETA – Communication factor

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.

15.2.3 Miscellaneous and debug

15.2.3.1 STARPU_SILENT – Disable verbose mode

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

15.2.3.2 STARPU_LOGFILENAME – Select debug file name

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

15.2.3.3 STARPU_FXT_PREFIX – FxT trace location

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

15.2.3.4 STARPU_LIMIT_GPU_MEM – Restrict memory size on the GPUs

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

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

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

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->handles[0])
 * 3- how a kernel can manipulate the data (buffers[0].vector.ptr)
 */
#include <starpu.h>
#include <starpu_openccl.h>

#define    NX    2048

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

static struct starpu_codelet cl = {
    .where = STARPU_CPU | STARPU_CUDA | STARPU_OPENCL,
    /* CPU implementation of the codelet */
    .cpu_funcs = { scal_cpu_func, scal_sse_func, NULL },
#ifdef STARPU_USE_CUDA
    /* CUDA implementation of the codelet */
    .cuda_funcs = { scal_cuda_func, NULL },
#endif
#ifdef STARPU_USE_OPENCL
    /* OpenCL implementation of the codelet */
    .openccl_funcs = { scal_openccl_func, NULL },
#endif
    .nbuffers = 1,
    .modes = { STARPU_RW }
};

#ifdef STARPU_USE_OPENCL
struct starpu_openccl_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_openccl_load_openccl_from_file(

```

```

        "examples/basic_examples/vector_scal_opengl_kernel.cl", &programs, NULL);
#endif

/* Tell StaPU to associate the "vector" vector with the "vector_handle"
 * identifier. When a task needs to access a piece of data, it should
 * refer to the handle that is associated to it.
 * In the case of the "vector" data interface:
 * - the first argument of the registration method is a pointer to the
 *   handle that should describe the data
 * - the second argument is the memory node where the data (ie. "vector")
 *   resides initially: 0 stands for an address in main memory, as
 *   opposed to an adress on a GPU for instance.
 * - the third argument is the address 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_t 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->handles[0] = vector_handle;

/* 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_OPENGL
    starpu_opengl_unload_opengl(&programs);
#endif

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

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

return 0;
}

```


A.2 CPU Kernel

```

#include <starpu.h>
#include <xmmintrin.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->handles array: for instance
     * task->handles[0] 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 *). 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.
     */
    struct starpu_vector_interface *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;
}

void scal_sse_func(void *buffers[], void *cl_arg)
{
    float *vector = (float *) STARPU_VECTOR_GET_PTR(buffers[0]);
    unsigned int n = STARPU_VECTOR_GET_NX(buffers[0]);
    unsigned int n_iterations = n/4;

    __m128 *VECTOR = (__m128*) vector;
    __m128 FACTOR __attribute__((aligned(16)));
    float factor = *(float *) cl_arg;
    FACTOR = _mm_set1_ps(factor);

    unsigned int i;
    for (i = 0; i < n_iterations; i++)
        VECTOR[i] = _mm_mul_ps(FACTOR, VECTOR[i]);

    unsigned int remainder = n%4;
    if (remainder != 0)
    {
        unsigned int start = 4 * n_iterations;
        for (i = start; i < start+remainder; ++i)
        {
            vector[i] = factor * vector[i];
        }
    }
}

```

```
    }
}
```

A.3 CUDA Kernel

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

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

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

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

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

    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

A.4 OpenCL Kernel

A.4.1 Invoking the kernel

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

extern struct starpu_opengl_program programs;

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

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

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

    err = starpu_opengl_load_kernel(&kernel, &queue, &programs, "vector_mult_opengl",
```

```

                                devid);
if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);

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

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

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

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

clFinish(queue);
starpu_opencil_collect_stats(event);
clReleaseEvent(event);

starpu_opencil_release_kernel(kernel);
}

```

A.4.2 Source of the kernel

```

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

```


Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

- be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
 - C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
 - D. Preserve all the copyright notices of the Document.
 - E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
 - F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
 - G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
 - H. Include an unaltered copy of this License.
 - I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
 - J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
 - K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
 - L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
 - M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
 - N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
 - O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Function Index

starpu_bcsr_data_register	73	starpu_csr_get_local_rowptr	78
starpu_bcsr_get_c	77	starpu_csr_get_nnz	78
starpu_bcsr_get_elemsize	77	STARPU_CSR_GET_NNZ	78
starpu_bcsr_get_firstentry	77	starpu_csr_get_nrow	78
starpu_bcsr_get_local_colind	77	STARPU_CSR_GET_NROW	78
starpu_bcsr_get_local_nzval	77	STARPU_CSR_GET_NZVAL	78
starpu_bcsr_get_local_rowptr	77	STARPU_CSR_GET_ROWPTR	78
starpu_bcsr_get_nnz	77	starpu_cuda_get_device_properties	93
starpu_bcsr_get_nrow	77	starpu_cuda_get_local_stream	93
starpu_bcsr_get_r	77	starpu_cuda_worker_get_count	67
starpu_block_data_register	73	starpu_data_acquire	71
starpu_block_filter_func	81	starpu_data_acquire_cb	71
starpu_block_filter_func_block	82	STARPU_DATA_ACQUIRE_CB	72
starpu_block_filter_func_vector	81	starpu_data_advise_as_important	71
STARPU_BLOCK_GET_DEV_HANDLE	76	starpu_data_cpy	95
starpu_block_get_elemsize	76	starpu_data_expected_transfer_time	103
STARPU_BLOCK_GET_ELEMSIZE	77	starpu_data_get_child	80
STARPU_BLOCK_GET_LDY	77	starpu_data_get_default_sequential_	
STARPU_BLOCK_GET_LDZ	77	consistency_flag	88
starpu_block_get_local_ldy	76	starpu_data_get_interface_on_node	73
starpu_block_get_local_ldz	76	starpu_data_get_nb_children	80
starpu_block_get_local_ptr	76	starpu_data_get_rank	51
starpu_block_get_nx	76	starpu_data_get_sub_data	80
STARPU_BLOCK_GET_NX	77	starpu_data_get_tag	51
starpu_block_get_ny	76	starpu_data_invalidate	70
STARPU_BLOCK_GET_NY	77	starpu_data_lookup	70
starpu_block_get_nz	76	starpu_data_map_filters	80
STARPU_BLOCK_GET_NZ	77	starpu_data_partition	79
STARPU_BLOCK_GET_OFFSET	76	starpu_data_prefetch_on_node	70
STARPU_BLOCK_GET_PTR	76	starpu_data_query_status	71
starpu_bound_compute	44	starpu_data_register	70
starpu_bound_print	44	starpu_data_release	72
starpu_bound_print_dot	44	starpu_data_request_allocation	71
starpu_bound_print_lp	44	starpu_data_set_default_sequential_	
starpu_bound_print_mps	44	consistency_flag	88, 89
starpu_bound_start	44	starpu_data_set_rank	51
starpu_bound_stop	44	starpu_data_set_reduction_methods	71
starpu_bus_get_count	92	starpu_data_set_sequential_consistency_flag	
starpu_bus_get_dst	92	89
starpu_bus_get_id	92	starpu_data_set_tag	51
starpu_bus_get_profiling_info	92	starpu_data_set_wt_mask	70
starpu_bus_get_src	92	starpu_data_unpartition	80
starpu_bus_profiling_helper_display_summary		starpu_data_unregister	70
.....	92	starpu_data_unregister_no_coherency	70
starpu_canonical_block_filter_bcsr	81	starpu_data_vget_sub_data	80
starpu_conf_init	66	starpu_data_vmap_filters	80
starpu_cpu_worker_get_count	67	starpu_display_codelet_stats	86
starpu_csr_data_register	73	starpu_execute_on_each_worker	95
STARPU_CSR_GET_COLIND	78	starpu_force_bus_sampling	90
starpu_csr_get_elemsize	78	starpu_free	69
STARPU_CSR_GET_ELEMSIZE	78	STARPU_GCC_PLUGIN	60
starpu_csr_get_firstentry	78	starpu_get_current_task	86
STARPU_CSR_GET_FIRSTENTRY	78	starpu_get_handle_interface_id	74
starpu_csr_get_local_colind	78	starpu_handle_get_local_ptr	74
starpu_csr_get_local_nzval	78	starpu_handle_to_pointer	74

starpu_helper_cublas_init	93	starpu_perfmodel_debugfilepath.....	90
starpu_helper_cublas_shutdown.....	93	starpu_perfmodel_get_arch_name.....	90
starpu_init	65	starpu_print_bus_bandwidth.....	90
starpu_insert_task	28	starpu_profiling_status_get	91
starpu_list_models	90	starpu_profiling_status_set	91
starpu_load_history_debug	90	starpu_progression_hook_deregister.....	104
starpu_malloc	69	starpu_progression_hook_register	103
starpu_matrix_data_register.....	73	starpu_push_local_task.....	102
STARPU_MATRIX_GET_DEV_HANDLE.....	75	starpu_sched_get_max_priority.....	102
starpu_matrix_get_elemsize	75	starpu_sched_get_min_priority.....	102
STARPU_MATRIX_GET_ELEMSIZE	76	starpu_sched_set_max_priority.....	102
STARPU_MATRIX_GET_LD	76	starpu_sched_set_min_priority.....	102
starpu_matrix_get_local_ld.....	75	starpu_set_profiling_id.....	91
starpu_matrix_get_local_ptr.....	75	starpu_shutdown.....	67
starpu_matrix_get_nx	75	starpu_spu_worker_get_count	67
STARPU_MATRIX_GET_NX	76	starpu_tag_declare_deps.....	87
starpu_matrix_get_ny	75	starpu_tag_declare_deps_array.....	87
STARPU_MATRIX_GET_NY	76	starpu_tag_notify_from_apps	88
STARPU_MATRIX_GET_OFFSET	76	starpu_tag_remove.....	88
STARPU_MATRIX_GET_PTR.....	75	starpu_tag_wait.....	88
starpu_mpi_barrier	48	starpu_tag_wait_array.....	88
starpu_mpi_gather_detached.....	53	starpu_task_bundle_close	100
starpu_mpi_get_data_on_node.....	52	starpu_task_bundle_create	99
starpu_mpi_initialize.....	47	starpu_task_bundle_insert	99
starpu_mpi_initialize_extended.....	47	starpu_task_bundle_remove	100
starpu_mpi_insert_task.....	51	starpu_task_create	86
starpu_mpi_irecv.....	48	starpu_task_declare_deps_array.....	87
starpu_mpi_irecv_array_detached_unlock_tag.....	48	starpu_task_deinit	86
starpu_mpi_irecv_detached	48	starpu_task_destroy	86
starpu_mpi_irecv_detached_unlock_tag.....	48	starpu_task_expected_conversion_time.....	103
starpu_mpi_isend.....	48	starpu_task_expected_data_transfer_time.....	103
starpu_mpi_isend_array_detached_unlock_tag.....	48	starpu_task_expected_length	103
starpu_mpi_isend_detached	48	starpu_task_expected_power	103
starpu_mpi_isend_detached_unlock_tag.....	48	starpu_task_init.....	85
starpu_mpi_recv	47	starpu_task_list_back.....	100
starpu_mpi_scatter_detached.....	53	starpu_task_list_begin.....	100
starpu_mpi_send.....	47	starpu_task_list_empty.....	100
starpu_mpi_shutdown.....	47	starpu_task_list_end.....	101
starpu_mpi_test	48	starpu_task_list_erase.....	100
starpu_mpi_wait	48	starpu_task_list_front.....	100
starpu_multiformat_data_register.....	99	starpu_task_list_init.....	100
starpu_opengl_collect_stats.....	94	starpu_task_list_next.....	101
starpu_opengl_display_error	93	starpu_task_list_pop_back	100
starpu_opengl_get_context	93	starpu_task_list_pop_front.....	100
starpu_opengl_get_current_context.....	93	starpu_task_list_push_back.....	100
starpu_opengl_get_current_queue	94	starpu_task_list_push_front	100
starpu_opengl_get_device	93	starpu_task_submit	86
starpu_opengl_get_global_mem_size.....	93	starpu_task_wait.....	86
starpu_opengl_get_queue.....	93	starpu_task_wait_for_all	86
starpu_opengl_load_kernel	94	starpu_task_wait_for_no_ready.....	87
starpu_opengl_load_opengl_from_file.....	94	starpu_timing_now	103
starpu_opengl_load_opengl_from_string.....	94	starpu_timing_timespec_delay_us	92
starpu_opengl_release_kernel.....	94	starpu_timing_timespec_to_us.....	92
starpu_opengl_set_kernel_args.....	94	starpu_variable_data_register.....	72
starpu_opengl_unload_opengl.....	94	starpu_variable_get_elemsize.....	74
starpu_opengl_worker_get_count.....	67	STARPU_VARIABLE_GET_ELEMSIZE.....	74
		starpu_variable_get_local_ptr.....	74

STARPU_VARIABLE_GET_PTR.....	74	starpu_worker_get_ids_by_type.....	68
starpu_vector_data_register.....	72	starpu_worker_get_memory_node.....	68
starpu_vector_divide_in_2_filter_func.....	81	starpu_worker_get_name.....	68
STARPU_VECTOR_GET_DEV_HANDLE.....	75	starpu_worker_get_perf_archtype.....	90
starpu_vector_get_elemsize.....	74	starpu_worker_get_profiling_info.....	92
STARPU_VECTOR_GET_ELEMSIZE.....	75	starpu_worker_get_relative_speedup.....	103
starpu_vector_get_local_ptr.....	75	starpu_worker_get_type.....	68
starpu_vector_get_nx.....	74	starpu_worker_profiling_helper_display_	
STARPU_VECTOR_GET_NX.....	75	summary.....	92
STARPU_VECTOR_GET_OFFSET.....	75	starpu_worker_set_sched_condition.....	102
STARPU_VECTOR_GET_PTR.....	75	starpufft_cleanup.....	56
starpu_vector_list_filter_func.....	81	starpufft_destroy_plan.....	56
starpu_vertical_block_filter_func.....	81	starpufft_execute.....	56
starpu_vertical_block_filter_func_csr.....	81	starpufft_execute_handle.....	56
starpu_void_data_register.....	72	starpufft_free.....	55
starpu_wake_all_blocked_workers.....	103	starpufft_malloc.....	55
starpu_worker_can_execute_task.....	102	starpufft_plan_dft_1d.....	55
starpu_worker_get_count.....	67	starpufft_plan_dft_2d.....	56
starpu_worker_get_count_by_type.....	67	starpufft_start.....	56
starpu_worker_get_devid.....	68	starpufft_start_handle.....	56
starpu_worker_get_id.....	67		

Datatype Index

E

enum starpu_access_mode	69
enum starpu_archtype	67
enum starpu_data_interface_id	74
enum starpu_perf_archtype	89

S

starpu_data_handle_t	69
starpu_tag_t	87
starpu_task_bundle_t	99
struct starpu_bus_profiling_info	92
struct starpu_codelet	82
struct starpu_conf	65

struct starpu_data_copy_methods	98
struct starpu_data_filter	79
struct starpu_data_interface_ops	97
struct starpu_machine_topology	101
struct starpu_multiformat_data_interface_ops	99
struct starpu_opencl_program	94
struct starpu_per_arch_perfmodel	90
struct starpu_perfmodel	89
struct starpu_sched_policy	101
struct starpu_task	83
struct starpu_task_list	100
struct starpu_task_profiling_info	91
struct starpu_worker_profiling_info	91

