

StarPU Handbook - StarPU Basics

for StarPU 1.4.8

1 Organization	3
2 StarPU Applications	5
2.1 Setting Flags for Compiling, Linking and Running Applications	5
2.2 Integrating StarPU in a Build System	6
2.2.1 Integrating StarPU in a Make Build System	6
2.2.2 Integrating StarPU in a CMake Build System	6
2.3 Running a Basic StarPU Application	7
2.4 Running a Basic StarPU Application on Microsoft Visual C	7
2.5 Kernel Threads Started by StarPU	8
2.6 Enabling OpenCL	8
2.7 Storing Performance Model Files	8
3 Basic Examples	11
3.1 Hello World	11
3.1.1 Required Headers	11
3.1.2 Defining A Codelet	11
3.1.3 Submitting A Task	11
3.1.4 Execution Of Hello World	12
3.1.5 Passing Arguments To The Codelet	12
3.1.6 Defining A Callback	13
3.1.7 Where To Execute A Codelet	13
3.2 Vector Scaling	13
3.2.1 Source Code of Vector Scaling	13
3.2.2 Execution of Vector Scaling	14
3.3 Vector Scaling on an Hybrid CPU/GPU Machine	14
3.3.1 Definition of the CUDA Kernel	14
3.3.2 Definition of the OpenCL Kernel	15
3.3.3 Definition of the Main Code	15
3.3.4 Execution of Hybrid Vector Scaling	17
4 Full Source Code for the 'Scaling a Vector' Example	19
4.1 Main Application	19
4.2 CPU Kernel	20
4.3 CUDA Kernel	21
4.4 OpenCL Kernel	21
4.4.1 Invoking the Kernel	21
4.4.2 Source of the Kernel	21
5 Tasks In StarPU	23
5.1 Task Granularity	23
5.2 Task Submission	24
5.3 Task Priorities	24
5.4 Setting Many Data Handles For a Task	25

5.5 Setting a Variable Number Of Data Handles For a Task	25
5.6 Insert Task Utility	25
5.7 Other Task Utility Functions	27
6 Data Management	29
6.1 Data Interface	29
6.1.1 Variable Data Interface	29
6.1.2 Vector Data Interface	29
6.1.3 Matrix Data Interface	30
6.1.4 Block Data Interface	30
6.1.5 Tensor Data Interface	30
6.1.6 Ndim Data Interface	31
6.1.7 BCSR Data Interface	32
6.1.8 CSR Data Interface	32
6.1.9 COO Data Interface	32
6.2 Partitioning Data	33
6.3 Asynchronous Partitioning	33
6.4 Commute Data Access	34
6.5 Data Reduction	34
6.6 Concurrent Data Accesses	36
6.7 Temporary Buffers	37
6.7.1 Temporary Data	37
6.7.2 Scratch Data	37
7 Scheduling	39
7.1 Task Scheduling Policies	39
7.1.1 Non Performance Modelling Policies	39
7.1.2 Performance Model-Based Task Scheduling Policies	39
7.1.3 Modularized Schedulers	40
7.2 Task Distribution Vs Data Transfer	41
8 Examples in StarPU Sources	43
I Appendix	45
9 The GNU Free Documentation License	47
9.1 ADDENDUM: How to use this License for your documents	51

This manual documents the usage of StarPU version 1.4.8. Its contents was last updated on 2025-06-16.

Copyright © 2009-2025 University of Bordeaux, CNRS (LaBRI UMR 5800), Inria

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".

Chapter 1

Organization

This part presents the basic knowledge of StarPU. It should be read to understand how StarPU works and how to execute a basic StarPU application.

- Chapter [StarPU Applications, setting up Your Own Code](#) shows how to create and run your own StarPU applications.
- Chapter [Basic Examples](#) shows how to implement simple programs that submit tasks to StarPU.
- Chapter [Full source code for the 'Scaling a Vector' example](#) gives the full source code for a vector scaling application.

The next chapters cover the most important and core concepts in StarPU:

- Chapter [Tasks In StarPU](#) explains the basic information on tasks management.
- Chapter [Data Management](#) shows how to manage the data layout of your application data by using the different data interfaces provided by StarPU.
- Chapter [Scheduling](#) explains the scheduling policies provided by StarPU.

Some examples applications are provided from the StarPU sources for you to try. Chapter [Examples in StarPU Sources](#) lists these applications.

Chapter 2

StarPU Applications

2.1 Setting Flags for Compiling, Linking and Running Applications

StarPU provides a `pkg-config` executable to facilitate the retrieval of necessary compiler and linker flags. This is useful when compiling and linking an application with StarPU, as certain flags or libraries (such as CUDA or `libspe2`) may be required.

If StarPU is not installed in a standard location, the path of StarPU's library must be specified in the environment variable `PKG_CONFIG_PATH` to allow `pkg-config` to find it. For example, if StarPU is installed in `$STARPU_PATH`, you can set the variable `PKG_CONFIG_PATH` like this:

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$STARPU_PATH/lib/pkgconfig
```

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

```
$ pkg-config --cflags starpu-1.4 # options for the compiler
$ pkg-config --libs starpu-1.4  # options for the linker
```

Please note that it is still possible to use the API provided in StarPU version 1.0 by calling `pkg-config` with the `starpu-1.0` package. Similar packages are provided for `starpumpi-1.0` and `starpufft-1.0`. For the API provided in StarPU version 0.9, you can use `pkg-config` with the `libstarpu` package. Similar packages are provided for `libstarpumpi` and `libstarpufft`.

Make sure that `pkg-config --libs starpu-1.4` produces valid output before going further. To achieve this, make sure that your `PKG_CONFIG_PATH` is correctly set to the location where `starpu-1.4.pc` was installed during the `make install` process.

Furthermore, if you intend to link your application statically, remember to include the `-static` option during the linking process.

Additionally, for runtime execution, it is necessary to set the `LD_LIBRARY_PATH` environment variable. This ensures that dynamic libraries are located and loaded correctly during runtime.

```
$ export LD_LIBRARY_PATH=$STARPU_PATH/lib:$LD_LIBRARY_PATH
```

And finally you should set the `PATH` variable to get access to various StarPU tools:

```
$ export PATH=$PATH:$STARPU_PATH/bin
```

Run the following command to ensure that StarPU is executing properly and successfully detecting your hardware. If any issues arise, examine the output of `lstopo` from the `hwloc` project and report any problems either to the `hwloc` project or to us.

```
$ starpu_machine_display
```

A tool is provided to help set all the environment variables needed by StarPU. Once StarPU is installed in a specific directory, calling the script `bin/starpu_env` will set in your current environment the variables `STARPU_PATH`, `LD_LIBRARY_PATH`, `PKG_CONFIG_PATH`, `PATH` and `MANPATH`.

```
$ source $STARPU_PATH/bin/starpu_env
```

2.2 Integrating StarPU in a Build System

2.2.1 Integrating StarPU in a Make Build System

When using a Makefile, the following lines can be added to set the options for the compiler and the linker:

```
CFLAGS      +=      $$$(pkg-config --cflags starpu-1.4)
LDLIBS     +=      $$$(pkg-config --libs starpu-1.4)
```

If you have a `test-starpu.c` file containing for instance:

```
#include <starpu.h>
#include <stdio.h>
int main(void)
{
    int ret;
    ret = starpu_init(NULL);
    if (ret != 0)
    {
        return 1;
    }
    printf("%d CPU cores\n", starpu_worker_get_count_by_type(STARPU_CPU_WORKER));
    printf("%d CUDA GPUs\n", starpu_worker_get_count_by_type(STARPU_CUDA_WORKER));
    printf("%d OpenCL GPUs\n", starpu_worker_get_count_by_type(STARPU_OPENCL_WORKER));
    starpu_shutdown();
    return 0;
}
```

You can build it with `make test-starpu` and run it with `./test-starpu`

2.2.2 Integrating StarPU in a CMake Build System

This section shows a minimal example integrating StarPU in an existing application's CMake build system.

Let's assume we want to build an executable from the following source code using CMake:

```
#include <starpu.h>
#include <stdio.h>
int main(void)
{
    int ret;
    ret = starpu_init(NULL);
    if (ret != 0)
    {
        return 1;
    }
    printf("%d CPU cores\n", starpu_worker_get_count_by_type(STARPU_CPU_WORKER));
    printf("%d CUDA GPUs\n", starpu_worker_get_count_by_type(STARPU_CUDA_WORKER));
    printf("%d OpenCL GPUs\n", starpu_worker_get_count_by_type(STARPU_OPENCL_WORKER));
    starpu_shutdown();
    return 0;
}
```

The `CMakeLists.txt` file below uses the Pkg-Config support from CMake to autodetect the StarPU installation and library dependences (such as `libhwloc`) provided that the `PKG_CONFIG_PATH` variable is set, and is sufficient to build a statically-linked executable.

The CMake code uses the `IMPORTED_TARGET` option of `pkg_check_modules` to define a CMake target that can be used to compile and link StarPU codes:

```
{File CMakeLists.txt}
cmake_minimum_required (VERSION 3.2)
project (hello_starpu)
find_package(PkgConfig)
pkg_check_modules(STARPU REQUIRED IMPORTED_TARGET starpu-1.4)
add_executable(hello_starpu hello_starpu.c PkgConfig::STARPU)
```

One can also use the following alternative.

```
{File CMakeLists.txt}
cmake_minimum_required (VERSION 3.2)
project (hello_starpu)
find_package(PkgConfig)
pkg_check_modules(STARPU REQUIRED starpu-1.4)
if (STARPU_FOUND)
    include_directories (${STARPU_INCLUDE_DIRS})
    link_directories    (${STARPU_STATIC_LIBRARY_DIRS})
    link_libraries      (${STARPU_STATIC_LIBRARIES})
else (STARPU_FOUND)
    message(FATAL_ERROR "StarPU not found")
endif()
add_executable(hello_starpu hello_starpu.c)
```

The following `CMakeLists.txt` implements a more complex strategy, still relying on Pkg-Config, but also taking into account additional flags. While more complete, this approach makes CMake's build types (Debug, Release, ...) unavailable because of the direct affectation to variable `CMAKE_C_FLAGS`. If both the full flags support and the

build types support are needed, the `CMakeLists.txt` below may be altered to work with `CMAKE_C_FLAGS`, `_RELEASE`, `CMAKE_C_FLAGS_DEBUG`, and others as needed. This example has been successfully tested with CMake 3.2, though it may work with earlier CMake 3.x versions.

```
{File CMakeLists.txt}
cmake_minimum_required (VERSION 3.2)
project (hello_starpu)
find_package(PkgConfig)
pkg_check_modules(STARPU REQUIRED starpu-1.4)
# This section must appear before 'add_executable'
if (STARPU_FOUND)
# CFLAGS other than -I
  foreach(CFLAG ${STARPU_CFLAGS_OTHER})
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${CFLAG}")
  endforeach()
# Static LDFLAGS other than -L
  foreach(LDFLAG ${STARPU_STATIC_LDFLAGS_OTHER})
    set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${LDFLAG}")
  endforeach()
# -L directories
  link_directories(${STARPU_STATIC_LIBRARY_DIRS})
else (STARPU_FOUND)
  message(FATAL_ERROR "StarPU not found")
endif()
add_executable(hello_starpu hello_starpu.c)
# This section must appear after 'add_executable'
if (STARPU_FOUND)
# -I directories
  target_include_directories(hello_starpu PRIVATE ${STARPU_INCLUDE_DIRS})
# Static -l libs
  target_link_libraries(hello_starpu PRIVATE ${STARPU_STATIC_LIBRARIES})
endif()
```

2.3 Running a Basic StarPU Application

Basic examples using StarPU are built in the directory `examples/basic_examples/` (and installed in `$STARPU_PATH/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 this directory (`STARPU_HOME`).

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

2.4 Running a Basic StarPU Application on Microsoft Visual C

Batch files are provided to run StarPU applications under Microsoft Visual C. They are installed in `$STARPU_PATH/bin/msvc`.

To execute a StarPU application, you first need to set the environment variable `STARPU_PATH`.

```
c:\...> cd c:\cygwin\home\ci\starpu\
c:\...> set STARPU_PATH=c:\cygwin\home\ci\starpu\
c:\...> cd bin\msvc
c:\...> starpu_open.bat starpu_simple.c
```

The batch script will run Microsoft Visual C with a basic project file to run the given application.

The batch script `starpu_clean.bat` can be used to delete all compilation generated files.

The batch script `starpu_exec.bat` can be used to compile and execute a StarPU application from the command prompt.

```
c:\...> cd c:\cygwin\home\ci\starpu\
c:\...> set STARPU_PATH=c:\cygwin\home\ci\starpu\
c:\...> cd bin\msvc
c:\...> starpu_exec.bat ..\..\..\examples\basic_examples\hello_world.c
```

```
MSVC StarPU Execution
...
/out:hello_world.exe
...
```

```
Hello world (params = {1, 2.00000})
Callback function got argument 0000042
c:\....>
```

2.5 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.

If the application needs to reserve some cores for its own computations, it can do so with the field `starpu_conf::reserve_ncpus`, get the core IDs with `starpu_get_next_bindid()`, and bind to them with `starpu_bind_thread_on()`. Another option is for the application to pause StarPU by calling `starpu_pause()`, then to perform its own computations, and then to resume StarPU by calling `starpu_resume()` so that StarPU can execute tasks.

If a computation library used by the application actually creates its own thread, it may be useful to call `starpu_bind_thread_on_worker()` before e.g. initializing the library, so that the library records which binding it is supposed to use. And then call `starpu_bind_thread_on_main()` again, or `starpu_bind_thread_on_cpu()` if a core was reserved with `starpu_get_next_bindid()`.

In case that computation library wants to bind threads itself, and uses physical numbering instead of logical numbering (as defined by `hwloc`), `starpu_cpu_os_index()` can be used to convert from StarPU `cpuid` to OS `cpu` index.

2.6 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 other 2 devices by calling:

```
$ STARPU_NCUDA=2 ./application
```

2.7 Storing Performance Model Files

StarPU stores performance model files for bus benchmarking and codelet profiles in different directories.

By default, all files are stored in `$STARPU_HOME/.starpu/sampling`.

If the environment variable `STARPU_HOME` is not defined, its default value is `$HOME` on Unix environments, and `$USERPROFILE` on Windows environments.

Environment variables `STARPU_PERF_MODEL_DIR` and `STARPU_PERF_MODEL_PATH` can also be used to specify other directories in which to store performance files (SimulatedBenchmarks).

The configure option `--with-perf-model-dir` can also be used to define a performance model directory.

When looking for performance files either for bus benchmarking or for codelet performances, StarPU

- first looks in the directory specified by the environment variable `STARPU_PERF_MODEL_DIR`
- then looks in the directory specified by the configure option `--with-perf-model-dir` or in `$STARPU_HOME/.starpu/sampling` if the option is not set
- then looks in the directories specified by the environment variable `STARPU_PERF_MODEL_PATH`
- and finally looks in `$prefix/share/starpu/perfmodels/sampling`

If the files are not present and must be created, they will be created in the first defined directory from the list above.

```
rm -rf $PWD/xxx && STARPU_PERF_MODEL_DIR=$PWD/xxx ./application
```

will use performance model files from the directory `$STARPU_HOME/.starpu/sampling` if they are available, otherwise will create these files in `$STARPU_PERF_MODEL_DIR`.

To know the list of directories StarPU will search for performances files, one can use the tool `starpu_perfmodel_display`

```
$ starpu_perfmodel_display -d
directory: </home/user1/.starpu/sampling/codelets/45/>
directory: </usr/local/install/share/starpu/perfmodels/sampling/codelets/45/>
```

```
$ STARPU_PERF_MODEL_DIR=/tmp/xxx starpu_perfmodel_display -d
directory: </tmp/xxx/codelets/45/>
directory: </home/user1/.starpu/sampling/codelets/45/>
directory: </usr/local/install/share/starpu/perfmodels/sampling/codelets/45/>
```

When using the variable `STARPU_PERF_MODEL_DIR`, the directory will be created if it does not exist when dumping new performance model files.

When using the variable `STARPU_PERF_MODEL_PATH`, only existing directories will be taken into account.

```
$ mkdir /tmp/yyy && STARPU_PERF_MODEL_DIR=/tmp/xxx STARPU_PERF_MODEL_PATH=/tmp/zzz:/tmp/yyy starpu_perfmodel_c
[starpu][adrets][_perf_model_add_dir] Warning: directory </tmp/zzz> as set by variable STARPU_PERF_MODEL_PATH
directory: </tmp/xxx/codelets/45/>
directory: </home/user1/.starpu/sampling/codelets/45/>
directory: </tmp/yyy/codelets/45/>
directory: </usr/local/install/share/starpu/perfmodels/sampling/codelets/45/>
```

Once your application has created the performance files in a given directory, it is thus possible to move these files in another location and keep using them.

```
./application
# files are created in $HOME/.starpu/sampling
mv $HOME/.starpu/sampling /usr/local/starpu/sampling
STARPU_PERF_MODEL_DIR=/usr/local/starpu/sampling ./application
```


Chapter 3

Basic Examples

3.1 Hello World

This section shows how to implement a simple program that submits a task to StarPU. The full source code for this example is available in the file `examples/basic_examples/hello_world.c`

3.1.1 Required Headers

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

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

3.1.2 Defining A Codelet

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, x86, ...). For compatibility, make sure that the whole structure is properly initialized to zero, either by using the function `starpu_codelet_init()`, or by letting the compiler implicitly do it as exemplified below.

The field `starpu_codelet::nbuffers` 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.

We create a codelet which may only be executed on CPUs. When a CPU core will execute a codelet, it will call the function `cpu_func`, 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. We also ignore the second argument, which is a pointer to optional arguments for the codelet.

```
void cpu_func(void *buffers[], void *cl_arg)
{
    printf("Hello world\n");
}
struct starpu_codelet cl =
{
    .cpu_funcs = { cpu_func },
    .nbuffers = 0
};
```

3.1.3 Submitting A Task

Before submitting any tasks to StarPU, `starpu_init()` must be called, or `starpu_initialize()` must be called by giving application arguments. The `NULL` argument specifies that we use the default configuration. Tasks can then be submitted until the termination of StarPU – done by a call to `starpu_shutdown()`.

In the example below, a task structure is allocated by a call to `starpu_task_create()`. This function allocates and fills the task structure with its default settings, it does not submit the task to StarPU.

The field `starpu_task::cl` 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.

If the field `starpu_task::synchronous` is non-zero, task submission will be synchronous: the function `starpu_task_submit()` will not return until the task has been executed. Note that the function `starpu_shutdown()` does not guarantee that asynchronous tasks have been executed before it returns, `starpu_task_wait_for_all()` can be

used to this effect, or data can be unregistered (`starpu_data_unregister()`), 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()`.

```
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 */
    /* starpu_task_submit will be a blocking call. If unset,
    starpu_task_wait() needs to be called after submitting the task. */
    task->synchronous = 1;
    /* submit the task to StarPU */
    starpu_task_submit(task);
    /* terminate StarPU */
    starpu_shutdown();
    return 0;
}
```

3.1.4 Execution Of Hello World

```
$ make hello_world
cc $(pkg-config --cflags starpu-1.4) hello_world.c -o hello_world $(pkg-config --libs starpu-1.4)
$ ./hello_world
Hello world
```

3.1.5 Passing Arguments To The Codelet

The optional field `starpu_task::cl_arg` field is a pointer to a buffer (of size `starpu_task::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 to 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.

```
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);
}
```

As said before, the field `starpu_codelet::nbuffers` specifies the number of data buffers which are manipulated by the codelet. It does not count the argument — the parameter `cl_arg` of the function `cpu_func` — since it is not managed by our data management library, but just contains trivial parameters.

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 [Vector Scaling](#).

```
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);
    /* 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;
}
```

```
$ make hello_world
cc $(pkg-config --cflags starpu-1.4) hello_world.c -o hello_world $(pkg-config --libs starpu-1.4)
$ ./hello_world
Hello world (params = {1, 2.000000} )
```

3.1.6 Defining A Callback

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

```
void callback_function(void *);
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 */
    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;
}

$ make hello_world
cc $(pkg-config --cflags starpu-1.4) hello_world.c -o hello_world $(pkg-config --libs starpu-1.4)
$ ./hello_world
Hello world
Callback function (arg 42)
```

3.1.7 Where To Execute A Codelet

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

We create a codelet which may only be executed on the CPUs. The optional field `starpu_codelet::where` is a bitmask which defines where the codelet may be executed. Here, the value `STARPU_CPU` means that only CPUs can execute this codelet. When the optional field `starpu_codelet::where` is unset, its value is automatically set based on the availability of the different fields `XXX_funcs`.

3.2 Vector Scaling

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 [Full source code for the 'Scaling a Vector' example](#).

3.2.1 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, programmers first need to declare the different pieces of data to StarPU using the functions `starpu_*_data_register`. 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, STARPU_MAIN_RAM, (uintptr_t)vector, NX, sizeof(vector[0]));
```

The first argument, called the **data handle**, is an opaque pointer which designates the array within 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 `STARPU_MAIN_RAM` since the array `vector` 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 pointer `starpu_task::cl_arg` like in the previous example. The vector parameter is described by its handle. `starpu_task::handles` should be set with the handles of the data, the access modes for the data are defined in the field `starpu_codelet::modes` (`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 =
{
    .cpu_funcs = { scal_cpu_func },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

The first argument is an array that gives a description of all the buffers passed in the array `starpu_task::handles`. The size of this array is given by the field `starpu_codelet::nbuffers`. 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 `starpu_vector_interface::ptr` (resp. `starpu_vector_interface::nx`) of this interface. 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 function `scal_cpu_func` contains a pointer to the parameters of the codelet (given in `starpu_task::cl_arg`), so that we read the constant factor from this pointer.

3.2.2 Execution of Vector Scaling

```
$ make vector_scal
cc $(pkg-config --cflags starpu-1.4) vector_scal.c -o vector_scal $(pkg-config --libs starpu-1.4)
$ ./vector_scal
0.000000 3.000000 6.000000 9.000000 12.000000
```

3.3 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.

3.3.1 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 kernel call `vector_mult_cuda`.

```
#include <starpu.h>
static __global__ void vector_mult_cuda(unsigned n, float *val, 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()>>(n, val, *factor);
    cudaError_t status = cudaGetLastError();
    if (status != cudaSuccess) STARPU_CUDA_REPORT_ERROR(status);
    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}

```

3.3.2 Definition of the OpenCL Kernel

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

```
__kernel void vector_mult_opencl(int nx, __global float* val, 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 [Partitioning Data](#).

```
#include <starpu.h>
extern struct starpu_opencl_program programs;
void scal_opencl_func(void *buffers[], void *_args)
{
    float *factor = _args;
    int id, devid, err; /* OpenCL specific code */
    cl_kernel kernel; /* OpenCL specific code */
    cl_command_queue queue; /* OpenCL specific code */
    cl_event event; /* OpenCL specific code */
    /* 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]);
    { /* OpenCL specific code */
        id = starpu_worker_get_id();
        devid = starpu_worker_get_devid(id);
        err = starpu_opencl_load_kernel(&kernel, &queue, &programs,
            "vector_mult_opencl", /* Name of the codelet */
            devid);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
        err = clSetKernelArg(kernel, 0, sizeof(n), &n);
        err |= clSetKernelArg(kernel, 1, sizeof(val), &val);
        err |= clSetKernelArg(kernel, 2, sizeof(*factor), factor);
        if (err) STARPU_OPENCL_REPORT_ERROR(err);
    }
    { /* OpenCL specific code */
        size_t global=n;
        size_t local;
        size_t s;
        cl_device_id device;
        starpu_opencl_get_device(devid, &device);
        err = clGetKernelWorkGroupInfo (kernel, device, CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local,
            &s);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
        if (local > global) local=global;
        else global = (global + local-1) / local * local;
        err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
        if (err != CL_SUCCESS) STARPU_OPENCL_REPORT_ERROR(err);
    }
    { /* OpenCL specific code */
        clFinish(queue);
        starpu_opencl_collect_stats(event);
        clReleaseEvent(event);
        starpu_opencl_release_kernel(kernel);
    }
}

```

3.3.3 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 that the fields `starpu_codelet::cuda_funcs` and `starpu_codelet::openccl_funcs` are set to define the pointers to the CUDA and OpenCL implementations of the task.

```

/*
 * 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>
#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 },
    .cpu_funcs_name = { "scal_cpu_func", "scal_sse_func" },
#ifdef STARPU_USE_CUDA
    /* CUDA implementation of the codelet */
    .cuda_funcs = { scal_cuda_func },
#endif
#ifdef STARPU_USE_OPENCL
    /* OpenCL implementation of the codelet */
    .openccl_funcs = { scal_openccl_func },
#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_openccl_kernel.cl", &programs,
    NULL);
#endif
    /* Tell StarPU 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: STARPU_MAIN_RAM stands for an address in main memory, as
 * opposed to an address 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, STARPU_MAIN_RAM, (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 resource */
    starpu_task_submit(task);
    /* StarPU does not need to manipulate the array anymore so we can stop
 * monitoring it */
    starpu_data_unregister(vector_handle);
#ifdef STARPU_USE_OPENCL
    starpu_openccl_unload_openccl(&programs);
#endif
    /* terminate StarPU, no task can be submitted after */
}

```

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

3.3.4 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 runtime when calling the function `starpu_opencil_load_opencil_from_file()`.

```
CFLAGS += $(shell pkg-config --cflags starpu-1.4)
LDLIBS += $(shell pkg-config --libs starpu-1.4)
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_NCPU=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 [Enabling OpenCL](#)) :

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


Chapter 4

Full Source Code for the 'Scaling a Vector' Example

4.1 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>
#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_openc1_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 },
    .cpu_funcs_name = { "scal_cpu_func", "scal_sse_func" },
#ifdef STARPU_USE_CUDA
    /* CUDA implementation of the codelet */
    .cuda_funcs = { scal_cuda_func },
#endif
#ifdef STARPU_USE_OPENCL
    /* OpenCL implementation of the codelet */
    .openc1_funcs = { scal_openc1_func },
#endif
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
#ifdef STARPU_USE_OPENCL
struct starpu_openc1_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_openc1_load_openc1_from_file("examples/basic_examples/vector_scal_openc1_kernel.cl", &programs,
    NULL);
#endif
    /* Tell StarPU 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: STARPU_MAIN_RAM stands for an address in main memory, as
 * opposed to an address 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, STARPU_MAIN_RAM, (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 resource */
    starpu_task_submit(task);
    /* StarPU does not need to manipulate the array anymore so we can stop
 * monitoring it */
    starpu_data_unregister(vector_handle);
#ifdef STARPU_USE_OPENCL
    starpu_opencil_unload_opencil(&programs);
#endif
    /* terminate StarPU, no task can be submitted after */
    starpu_shutdown();
    fprintf(stderr, "AFTER First element is %f\n", vector[0]);
    return 0;
}

```

4.2 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 = STARPU_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];
        }
    }
}

```

4.3 CUDA Kernel

```
#include <starp_u.h>
static __global__ void vector_mult_cuda(unsigned n, float *val, 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()>>(n, val, *factor);
    cudaError_t status = cudaGetLastError();
    if (status != cudaSuccess) STARPU_CUDA_REPORT_ERROR(status);
    cudaStreamSynchronize(starp_u_cuda_get_local_stream());
}
```

4.4 OpenCL Kernel

4.4.1 Invoking the Kernel

```
#include <starp_u.h>
extern struct starpu_opencil_program programs;
void scal_opencil_func(void *buffers[], void *_args)
{
    float *factor = _args;
    int id, devid, err; /* OpenCL specific code */
    cl_kernel kernel; /* OpenCL specific code */
    cl_command_queue queue; /* OpenCL specific code */
    cl_event event; /* OpenCL specific code */
    /* 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]);
    { /* OpenCL specific code */
        id = starpu_worker_get_id();
        devid = starpu_worker_get_devid(id);
        err = starpu_opencil_load_kernel(&kernel, &queue, &programs,
            "vector_mult_opencil", /* Name of the codelet */
            devid);
        if (err != CL_SUCCESS) STARPU_OPENCIL_REPORT_ERROR(err);
        err = clSetKernelArg(kernel, 0, sizeof(n), &n);
        err |= clSetKernelArg(kernel, 1, sizeof(val), &val);
        err |= clSetKernelArg(kernel, 2, sizeof(*factor), factor);
        if (err) STARPU_OPENCIL_REPORT_ERROR(err);
    }
    { /* OpenCL specific code */
        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_OPENCIL_REPORT_ERROR(err);
        if (local > global) local=global;
        else global = (global + local - 1) / local * local;
        err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
        if (err != CL_SUCCESS) STARPU_OPENCIL_REPORT_ERROR(err);
    }
    { /* OpenCL specific code */
        clFinish(queue);
        starpu_opencil_collect_stats(event);
        clReleaseEvent(event);
        starpu_opencil_release_kernel(kernel);
    }
}
```

4.4.2 Source of the Kernel

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

```
        val[i] *= factor;  
    }  
}
```

Chapter 5

Tasks In StarPU

5.1 Task Granularity

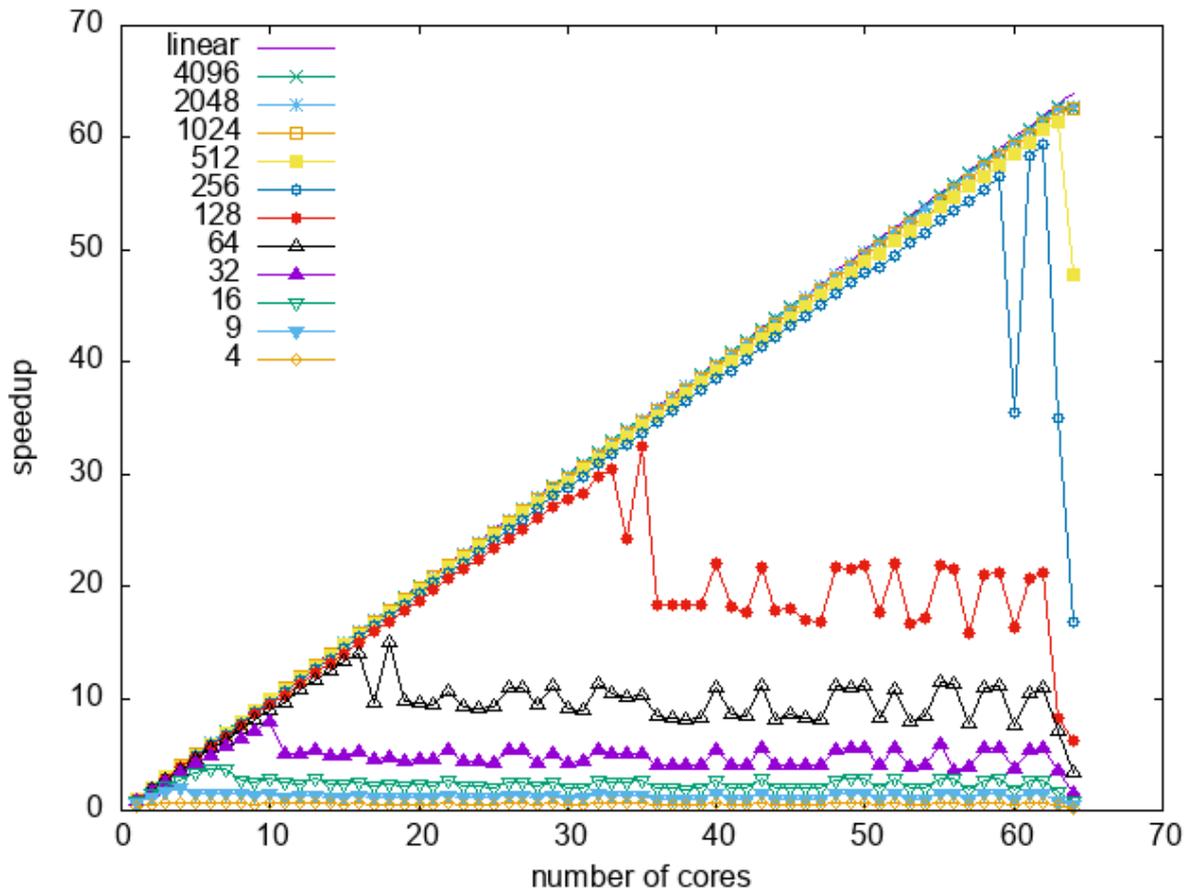
Similar to other runtimes, StarPU introduces some overhead in managing tasks. This overhead, while not always negligible, is mitigated by its intelligent scheduling and data management capabilities. The typical order of magnitude for this overhead is a few microseconds, which is notably smaller than the inherent CUDA overhead. To ensure that this overhead remains insignificant, the work assigned to a task should be substantial enough.

The length of tasks should ideally be relatively larger to effectively counterbalance this overhead. It is advised to consider the offline performance feedback, which provides insights into task lengths. Monitoring task lengths becomes crucial if you're encountering suboptimal performance.

To gauge the scalability potential based task size, you can run the `tests/microbenchs/tasks_size_↵overhead.sh` script. It provides a visual representation of the speedup achievable with independent tasks of very small sizes.

This benchmark is installed in `$STARPU_PATH/lib/starpu/examples/`. It gives a glimpse into how long a task should be (in μs) for StarPU overhead to be low enough to keep efficiency. The script generates a plot illustrating the speedup trends for tasks of different sizes, correlated with the number of CPUs in use.

For example, in the figure below, for 128 μs tasks (the red line), StarPU overhead is low enough to guarantee a good speedup if the number of CPUs is not more than 36. But with the same number of CPUs, 64 μs tasks (the black line) cannot have a correct speedup. The number of CPUs must be decreased to about 17 in order to keep efficiency.



To determine the task size your application is using, it is possible to use `starpu_fxt_data_trace` as explained in DataTrace.

The selection of a scheduler in StarPU also plays a significant role. Different schedulers have varying impacts on the overall execution. For example, the `dmda` scheduler may require additional time to make decisions, while the `eager` scheduler tends to be more immediate in its decisions.

To assess the impact of scheduler choice on your target machine, you can once again utilize the `tasks_size←_overhead.sh` script. This script provides valuable insights into how different schedulers affect performance in conjunction with task sizes.

5.2 Task Submission

To enable StarPU to perform online optimizations effectively, it is recommended to submit tasks asynchronously whenever possible. The goal is to maximize the level of asynchronous submission, allowing StarPU to have more flexibility in optimizing the scheduling process. Ideally, all tasks should be submitted asynchronously, and the use of functions like `starpu_task_wait_for_all()` or `starpu_data_unregister()` should be limited to waiting for task completion. StarPU will then be able to rework the whole schedule, overlap computation with communication, manage accelerator local memory usage, etc. A simple example is in the file `examples/basic_examples/variable.c`

5.3 Task Priorities

StarPU's default behavior considers tasks in the order they are submitted by the application. However, in scenarios where the application programmer possesses knowledge about certain tasks that should take priority due to their impact on performance (such as tasks whose output is crucial for subsequent tasks), the `starpu_task::priority` field can be utilized to convey this information to StarPU's scheduling process.

An example is provided in the application `examples/heat/dw_factolu_tag.c`.

5.4 Setting Many Data Handles For a Task

The maximum number of data that a task can manage is fixed by the macro `STARPU_NMAXBUFS`. This macro has a default value which can be customized through the `configure` option `--enable-maxbuffers`.

However, if you have specific cases where you need tasks to manage more data than the maximum allowed, you can use the field `starpu_task::dyn_handles` when defining a task, along with the field `starpu_codelet::dyn_modes` when defining the corresponding codelet.

This dynamic handle mechanism enables tasks to handle additional data beyond the usual limit imposed by `STARPU_NMAXBUFS`.

```
enum starpu_data_access_mode modes[STARPU_NMAXBUFS+1] =
{
    STARPU_R, STARPU_R, ...
};
struct starpu_codelet dummy_big_cl =
{
    .cuda_funcs = { dummy_big_kernel },
    .opencl_funcs = { dummy_big_kernel },
    .cpu_funcs = { dummy_big_kernel },
    .cpu_funcs_name = { "dummy_big_kernel" },
    .nbuffers = STARPU_NMAXBUFS+1,
    .dyn_modes = modes
};
task = starpu_task_create();
task->cl = &dummy_big_cl;
task->dyn_handles = malloc(task->cl->nbuffers * sizeof(starpu_data_handle_t));
for(i=0 ; i<task->cl->nbuffers ; i++)
{
    task->dyn_handles[i] = handle;
}
starpu_task_submit(task);
starpu_data_handle_t *handles = malloc(dummy_big_cl.nbuffers * sizeof(starpu_data_handle_t));
for(i=0 ; i<dummy_big_cl.nbuffers ; i++)
{
    handles[i] = handle;
}
starpu_task_insert(&dummy_big_cl,
    STARPU_VALUE, &dummy_big_cl.nbuffers, sizeof(dummy_big_cl.nbuffers),
    STARPU_DATA_ARRAY, handles, dummy_big_cl.nbuffers,
    0);
```

The whole code for this complex data interface is available in the file `examples/basic_examples/dynamic_handles.c`.

5.5 Setting a Variable Number Of Data Handles For a Task

Normally, the number of data handles given to a task is set with `starpu_codelet::nbuffers`. This field can however be set to `STARPU_VARIABLE_NBUFFERS`, in which case `starpu_task::nbuffers` must be set, and `starpu_task::modes` (or `starpu_task::dyn_modes`, see [Setting Many Data Handles For a Task](#)) should be used to specify the modes for the handles. Examples in `examples/basic_examples/dynamic_handles.c` show how to implement it.

5.6 Insert Task Utility

StarPU provides the wrapper function `starpu_task_insert()` to ease the creation and submission of tasks.

Here is the implementation of a 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_codelet_unpack_args(_args, &ifactor, &ffactor);
    *x0 = *x0 * ifactor;
    *x1 = *x1 * ffactor;
}
struct starpu_codelet mycodelet =
{
    .cpu_funcs = { func_cpu },
    .cpu_funcs_name = { "func_cpu" },
    .nbuffers = 2,
    .modes = { STARPU_RW, STARPU_RW }
};
```

And the call to `starpu_task_insert()`:

```
starpu_task_insert(&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_task_insert()` 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_codelet_pack_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);

```

In the example file `tests/main/insert_task_value.c`, we use these two ways to create and submit tasks.

Instead of calling `starpu_codelet_pack_args()`, one can also call `starpu_codelet_pack_arg_init()`, then `starpu_codelet_pack_arg()` for each data, then `starpu_codelet_pack_arg_fini()` as follow:

```

struct starpu_task *task = starpu_task_create();
task->cl = &mycodelet;
task->handles[0] = data_handles[0];
task->handles[1] = data_handles[1];
struct starpu_codelet_pack_arg_data state;
starpu_codelet_pack_arg_init(&state);
starpu_codelet_pack_arg(&state, &ifactor, sizeof(ifactor));
starpu_codelet_pack_arg(&state, &ffactor, sizeof(ffactor));
starpu_codelet_pack_arg_fini(&state, &task->cl_arg, &task->cl_arg_size);
int ret = starpu_task_submit(task);

```

A full code example is in file `tests/main/pack.c`.

Here a similar call using `STARPU_DATA_ARRAY`.

```

starpu_task_insert(&mycodelet,
                  STARPU_DATA_ARRAY, data_handles, 2,
                  STARPU_VALUE, &ifactor, sizeof(ifactor),
                  STARPU_VALUE, &ffactor, sizeof(ffactor),
                  0);

```

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

```

/* Compute which portion we will work on, e.g. pivot */
starpu_task_insert(&which_index, STARPU_W, i_handle, 0);
/* And submit the corresponding task */
STARPU_DATA_ACQUIRE_CB(i_handle, STARPU_R,
                        starpu_task_insert(&work, STARPU_RW, A_handle[i], 0));

```

The macro `STARPU_DATA_ACQUIRE_CB` submits an asynchronous request for acquiring data `i` for the main application, and will execute the code given as the third parameter when it is acquired. In other words, as soon as the value of `i` computed by the codelet `which_index` can be read, the portion of code passed as the 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`. In the example file `tests/datawizard/acquire_cb_insert.c`, this macro is used.

StarPU also provides a utility function `starpu_codelet_unpack_args()` to retrieve the `STARPU_VALUE` arguments passed to the task. There is several ways of calling `starpu_codelet_unpack_args()`. The full code examples are available in the file `tests/main/insert_task_value.c`.

```

void func_cpu(void *descr[], void *_args)
{
    int ifactor;
    float ffactor;
    starpu_codelet_unpack_args(_args, &ifactor, &ffactor);
}
void func_cpu(void *descr[], void *_args)
{
    int ifactor;
    float ffactor;
    starpu_codelet_unpack_args(_args, &ifactor, 0);
    starpu_codelet_unpack_args(_args, &ifactor, &ffactor);
}
void func_cpu(void *descr[], void *_args)
{
    int ifactor;
    float ffactor;
    char buffer[100];
    starpu_codelet_unpack_args_and_copyleft(_args, buffer, 100, &ifactor, 0);
    starpu_codelet_unpack_args(buffer, &ffactor);
}

```

Instead of calling `starpu_codelet_unpack_args()`, one can also call `starpu_codelet_unpack_arg_init()`, then `starpu_codelet_pack_arg()` or `starpu_codelet_dup_arg()` or `starpu_codelet_pick_arg()` for each data, then

`starpu_codelet_unpack_arg_fini()` as follow:

```
void func_cpu(void *descr[], void *_args)
{
    int ifactor;
    float ffactor;
    size_t size = sizeof(int) + 2*sizeof(size_t) + sizeof(int) + sizeof(float);
    struct starpu_codelet_pack_arg_data state;
    starpu_codelet_unpack_arg_init(&state, _args, size);
    starpu_codelet_unpack_arg(&state, (void*)&ifactor, sizeof(ifactor));
    starpu_codelet_unpack_arg(&state, (void*)&ffactor, sizeof(ffactor));
    starpu_codelet_unpack_arg_fini(&state);
}

void func_cpu(void *descr[], void *_args)
{
    int *ifactor;
    float *ffactor;
    size_t size;
    size_t psize = sizeof(int) + 2*sizeof(size_t) + sizeof(int) + sizeof(float);
    struct starpu_codelet_pack_arg_data state;
    starpu_codelet_unpack_arg_init(&state, _args, psize);
    starpu_codelet_dup_arg(&state, (void*)&ifactor, &size);
    assert(size == sizeof(*ifactor));
    starpu_codelet_dup_arg(&state, (void*)&ffactor, &size);
    assert(size == sizeof(*ffactor));
    starpu_codelet_unpack_arg_fini(&state);
}

void func_cpu(void *descr[], void *_args)
{
    int *ifactor;
    float *ffactor;
    size_t size;
    size_t psize = sizeof(int) + 2*sizeof(size_t) + sizeof(int) + sizeof(float);
    struct starpu_codelet_pack_arg_data state;
    starpu_codelet_unpack_arg_init(&state, _args, psize);
    starpu_codelet_pick_arg(&state, (void*)&ifactor, &size);
    assert(size == sizeof(*ifactor));
    starpu_codelet_pick_arg(&state, (void*)&ffactor, &size);
    assert(size == sizeof(*ffactor));
    starpu_codelet_unpack_arg_fini(&state);
}
```

During unpacking one can also call `starpu_codelet_unpack_discard_arg()` to skip saving the argument in pointer.

A full code example is in file `tests/main/pack.c`.

5.7 Other Task Utility Functions

Here a list of other functions to help with task management.

- The function `starpu_task_dup()` creates a duplicate of an existing task. The new task is identical to the original task in terms of its parameters, dependencies, and execution characteristics.
- The function `starpu_task_set()` is used to set the parameters of a task before it is executed, while `starpu_task_build()` is used to create a task with the specified parameters.

StarPU provides several functions to help insert data into a task. The function `starpu_task_insert_data_make_room()` is used to allocate memory space for a data structure that is required for inserting data into a task. This function is called before inserting any data handles into a task, and ensures that enough memory is available for the data to be stored. Once memory is allocated, the data handle can be inserted into the task using the following functions

- `starpu_task_insert_data_process_arg()` processes a scalar argument of a task and inserts it into the task's data structure. This function also performs any necessary data allocation and transfer operations.
- `starpu_task_insert_data_process_array_arg()` processes an array argument of a task and inserts it into the task's data structure. This function handles the allocation and transfer of the array data, as well as setting up the appropriate metadata to describe the array.
- `starpu_task_insert_data_process_mode_array_arg()` processes a mode array argument of a task and inserts it into the task's data structure. This function handles the allocation and transfer of the mode array data, as well as setting up the appropriate metadata to describe the mode array. Additionally, this function also computes the necessary sizes and strides for the data associated with the mode array argument.

Chapter 6

Data Management

TODO: intro which mentions consistency among other things

6.1 Data Interface

StarPU provides several data interfaces for programmers to describe the data layout of their application. There are predefined interfaces already available in StarPU. Users can define new data interfaces as explained in [Defining a New Data Interface](#). All functions provided by StarPU are documented in [Data Interfaces](#). You will find a short list below.

6.1.1 Variable Data Interface

A variable is a given-size byte element, typically a scalar or a pointer to an application-specific structure. Here is an example of how to register a variable data to StarPU by using [starpu_variable_data_register\(\)](#). A full code example for the variable data interface is available in the file `examples/basic_examples/variable.c`.

```
float var = 42.0;
starpu_data_handle_t var_handle;
starpu_variable_data_register(&var_handle, STARPU_MAIN_RAM, (uintptr_t)&var, sizeof(var));
```

Here is an example of how to register an application-specific data to StarPU, the idea is to register the variable that contains the pointer to the application-specific data. This will not provide support for GPUs and MPI, but can be an easy start before defining your own data interface to describe the application-specific structure (see [Defining a New Data Interface](#)).

```
struct mystructure *A = ...;
starpu_data_handle_t var_handle;
starpu_variable_data_register(&var_handle, STARPU_MAIN_RAM, (uintptr_t)&A, sizeof(A));
```

6.1.2 Vector Data Interface

A vector is a fixed number of elements of a given size. Here is an example of how to register a vector data to StarPU by using [starpu_vector_data_register\(\)](#). A full code example for the vector data interface is available in the file `examples/filters/fvector.c`.

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

Vectors can be partitioned into pieces by using [starpu_vector_filter_block\(\)](#). They can also be partitioned with some overlapping by using [starpu_vector_filter_block_shadow\(\)](#). An example is in the file `examples/filters/shadow.c`.

By default, StarPU uses the same size for each piece. If different sizes are desired, [starpu_vector_filter_list\(\)](#) or [starpu_vector_filter_list_long\(\)](#) can be used instead.

To just divide in two pieces, [starpu_vector_filter_divide_in_2\(\)](#) can be used.

In addition, contiguous variables can be picked from a vector by using [starpu_vector_filter_pick_variable\(\)](#) with [starpu_data_filter::get_child_ops](#) set to [starpu_vector_filter_pick_variable_child_ops\(\)](#). An example is in the file `examples/filters/fvector_pick_variable.c`.

6.1.3 Matrix Data Interface

To register 2-D matrices with a potential padding, one can use the matrix data interface. Here is an example of how to register a matrix data to StarPU by using `starpu_matrix_data_register()`.

A full code example for the matrix data interface is available in the file `examples/filters/fmatrix.c`.

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

2D matrices can be partitioned into 2D matrices along the x dimension by using `starpu_matrix_filter_block()`, and along the y dimension by using `starpu_matrix_filter_vertical_block()`.

They can also be partitioned with some overlapping by using `starpu_matrix_filter_block_shadow()` and `starpu_matrix_filter_vertical_block_shadow()`. An example is in the file `examples/filters/shadow2d.c`.

In addition, contiguous vectors can be picked from a matrix along the Y dimension by using `starpu_matrix_filter_pick_vector_y()` with `starpu_data_filter::get_child_ops` set to `starpu_matrix_filter_pick_vector_child_ops()`. An example is in the file `examples/filters/fmatrix_pick_vector.c`.

Variable can be also picked from a matrix by using `starpu_matrix_filter_pick_variable()` with `starpu_data_filter::get_child_ops` needs set to `starpu_matrix_filter_pick_variable_child_ops()`. An example is in the file `examples/filters/fmatrix_pick_variable.c`.

6.1.4 Block Data Interface

To register 3-D matrices with potential paddings on Y and Z dimensions, one can use the block data interface. Here is an example of how to register a block data to StarPU by using `starpu_block_data_register()`. A full code example for the block data interface is available in the file `examples/filters/fblock.c`.

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

3D matrices can be partitioned along the x dimension by using `starpu_block_filter_block()`, or along the y dimension by using `starpu_block_filter_vertical_block()`, or along the z dimension by using `starpu_block_filter_depth_block()`.

They can also be partitioned with some overlapping by using `starpu_block_filter_block_shadow()`, `starpu_block_filter_vertical_block_shadow()` or `starpu_block_filter_depth_block_shadow()`. An example is in the file `examples/filters/shadow3d.c`.

In addition, contiguous matrices can be picked from a block along the Z dimension or the Y dimension by using `starpu_block_filter_pick_matrix_z()` or `starpu_block_filter_pick_matrix_y()` with `starpu_data_filter::get_child_ops` set to `starpu_block_filter_pick_matrix_child_ops()`. An example is in the file `examples/filters/fblock_pick_matrix.c`.

Variable can be also picked from a block by using `starpu_block_filter_pick_variable()` with `starpu_data_filter::get_child_ops` set to `starpu_block_filter_pick_variable_child_ops()`. An example is in the file `examples/filters/fblock_pick_variable.c`.

6.1.5 Tensor Data Interface

To register 4-D matrices with potential paddings on Y, Z, and T dimensions, one can use the tensor data interface. Here is an example of how to register a tensor data to StarPU by using `starpu_tensor_data_register()`. A full code example for the tensor data interface is available in the file `examples/filters/ftensor.c`.

```
float *block;
starpu_data_handle_t block_handle;
block = (float*)malloc(nx*ny*nz*nt*sizeof(float));
starpu_tensor_data_register(&block_handle, STARPU_MAIN_RAM, (uintptr_t)block, nx, nx*ny, nx*ny*nz, nx, ny,
    nz, nt, sizeof(float));
```

4D matrices can be partitioned along the x dimension by using `starpu_tensor_filter_block()`, or along the y dimension by using `starpu_tensor_filter_vertical_block()`, or along the z dimension by using `starpu_tensor_filter_depth_block()`, or along the t dimension by using `starpu_tensor_filter_time_block()`.

They can also be partitioned with some overlapping by using `starpu_tensor_filter_block_shadow()`, `starpu_tensor_filter_vertical_block_shadow()`, `starpu_tensor_filter_depth_block_shadow()`, or `starpu_tensor_filter_time_block_shadow()`. An example is in the file `examples/filters/shadow4d.c`.

In addition, contiguous blocks can be picked from a block along the T dimension, Z dimension or the Y dimension by using `starpu_tensor_filter_pick_block_t()`, `starpu_tensor_filter_pick_block_z()`, or `starpu_tensor_filter_pick_block_y()`, and `starpu_data_filter::get_child_ops` set to `starpu_tensor_filter_pick_block_child_ops()`. An example is in the file `examples/filters/ftensor_pick_block.c`.

Variable can be also picked from a tensor by using `starpu_tensor_filter_pick_variable()` with `starpu_data_filter::get_child_ops` set to `starpu_tensor_filter_pick_variable_child_ops()`. An example is in the file `examples/filters/ftensor←_pick_variable.c`.

6.1.6 Ndim Data Interface

To register N-dim matrices, one can use the Ndim data interface. Here is an example of how to register a 5-dim data to StarPU by using `starpu_ndim_data_register()`. A full code example for the ndim data interface is available in the file `examples/filters/fndim.c`.

```
float *arr5d;
starpu_data_handle_t arr5d_handle;
starpu_malloc((void **)&arr5d, NX*NY*NZ*NT*NG*sizeof(float));
unsigned nn[5] = {NX, NY, NZ, NT, NG};
unsigned ldn[5] = {1, NX, NX*NY, NX*NY*NZ, NX*NY*NZ*NT};
starpu_ndim_data_register(&arr5d_handle, STARPU_MAIN_RAM, (uintptr_t)arr5d, ldn, nn, 5, sizeof(float));
```

N-dim matrices can be partitioned along the given dimension by using `starpu_ndim_filter_block()`. They can also be partitioned with some overlapping by using `starpu_ndim_filter_block_shadow()`. An example is in the file `examples/filters/shadownd.c`.

Taking into account existing data interfaces, there are several specialized functions which can partition a 0-dim array, 1-dim array, 2-dim array, 3-dim array or 4-dim array into

- variables by using `starpu_ndim_filter_to_variable()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_to_variable_c` (see file `examples/filters/fndim_to_variable.c`),
- vectors by using `starpu_ndim_filter_to_vector()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_to_vector_child` (see file `examples/filters/fndim_to_vector.c`),
- matrices by using `starpu_ndim_filter_to_matrix()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_to_matrix_child` (see file `examples/filters/fndim_to_matrix.c`),
- blocks by using `starpu_ndim_filter_to_block()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_to_block_child_ops` (see file `examples/filters/fndim_to_block.c`),
- or tensors by using `starpu_ndim_filter_to_tensor()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_to_tensor_chi` (see file `examples/filters/fndim_to_tensor.c`).

In addition, contiguous (n-1)dim arrays can be picked from a ndim array along the given dimension by using `starpu_ndim_filter_pick_ndim()`. An example is in the file `examples/filters/fndim_pick_ndim.c`.

In specific cases which consider existing data interfaces, contiguous variables, vectors, matrices, blocks, or tensors can be along the given dimension picked from a

- 1-dim array by using `starpu_ndim_filter_1d_pick_variable()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_variable_child_ops()` (see file `examples/filters/fndim_1d_pick←_variable.c`),
- 2-dim array by using `starpu_ndim_filter_2d_pick_vector()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_vector_child_ops()` (see file `examples/filters/fndim_2d_pick←_vector.c`),
- 3-dim array by using `starpu_ndim_filter_3d_pick_matrix()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_matrix_child_ops()` (see file `examples/filters/fndim_3d_pick←_matrix.c`),
- 4-dim array by using `starpu_ndim_filter_4d_pick_block()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_block_child_ops()` (see file `examples/filters/fndim_4d_pick←_block.c`),
- or 5-dim array by using `starpu_ndim_filter_5d_pick_tensor()` and `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_tensor_child_ops()` (see file `examples/filters/fndim_5d_pick←_tensor.c`).

Variable can be also picked from a ndim array by using `starpu_ndim_filter_pick_variable()` with `starpu_data_filter::get_child_ops` set to `starpu_ndim_filter_pick_variable_child_ops()`. An example is in the file `examples/filters/fndim←_pick_variable.c`.

6.1.7 BCSR Data Interface

BCSR (Blocked Compressed Sparse Row Representation) sparse matrix data can be registered to StarPU using the bcsr data interface. Here is an example on how to do so by using [starpu_bcsr_data_register\(\)](#).

```

/*
 * We use the following matrix:
 *
 * +-----+
 * | 0  1  0  0 |
 * | 2  3  0  0 |
 * | 4  5  8  9 |
 * | 6  7 10 11 |
 * +-----+
 *
 * nzval = [0, 1, 2, 3] ++ [4, 5, 6, 7] ++ [8, 9, 10, 11]
 * colind = [0, 0, 1]
 * rowptr = [0, 1, 3]
 * r = c = 2
 */
/* Size of the blocks */
int R = 2;
int C = 2;
int NROWS = 2;
int NNZ_BLOCKS = 3; /* out of 4 */
int NZVAL_SIZE = (R*C*NNZ_BLOCKS);
int nzval[NZVAL_SIZE] =
{
    0, 1, 2, 3, /* First block */
    4, 5, 6, 7, /* Second block */
    8, 9, 10, 11 /* Third block */
};
uint32_t colind[NNZ_BLOCKS] =
{
    0, /* block-column index for first block in nzval */
    0, /* block-column index for second block in nzval */
    1 /* block-column index for third block in nzval */
};
uint32_t rowptr[NROWS+1] =
{
    0, /* block-index in nzval of the first block of the first row. */
    1, /* block-index in nzval of the first block of the second row. */
    NNZ_BLOCKS /* number of blocks, to allow an easier element's access for the kernels */
};
starpu_data_handle_t bcsr_handle;
starpu_bcsr_data_register(&bcsr_handle,
    STARPU_MAIN_RAM,
    NNZ_BLOCKS,
    NROWS,
    (uintptr_t) nzval,
    colind,
    rowptr,
    0, /* firstentry */
    R,
    C,
    sizeof(nzval[0]));

```

An example on how to deal with such matrices is in the file `examples/spmv/dw_block_spmv.c`.

BCSR data handles can be partitioned into its dense matrix blocks by using [starpu_bcsr_filter_canonical_block\(\)](#), or split into other BCSR data handles by using [starpu_bcsr_filter_vertical_block\(\)](#) (but only split along the leading dimension is supported, i.e. along adjacent nnz blocks). [starpu_data_filter::get_child_ops](#) needs to be set to [starpu_bcsr_filter_canonical_block_child_ops\(\)](#) and [starpu_data_filter::get_nchildren](#) set to [starpu_bcsr_filter_canonical_block_get_nchildren\(\)](#). An example is available in `tests/datawizard/bcsr.c`.

6.1.8 CSR Data Interface

TODO

To register a Compressed Sparse Row Representation (CSR) sparse matrix, one can use the CSR data interface. A full code example for the CSR data interface is available in the file `mpi/tests/datatypes.c` to show how to register a COO matrix data to StarPU by using [starpu_csr_data_register\(\)](#).

CSR data handles can be partitioned into vertical CSR matrices by using [starpu_csr_filter_vertical_block\(\)](#). An example is available in the file `examples/spmv/spmv.c`.

6.1.9 COO Data Interface

To register 2-D matrices given in the coordinate format (COO), one can use the COO data interface. A full code example for the COO data interface is available in the file `tests/datawizard/interfaces/coo/coo↔interface.c` to show how to register a COO matrix data to StarPU by using [starpu_coo_data_register\(\)](#).

6.2 Partitioning Data

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

```
#define NX 1048576
#define PARTS 16
int vector[NX];
starpu_data_handle_t handle;
/* Declare data to StarPU */
starpu_vector_data_register(&handle, STARPU_MAIN_RAM, (uintptr_t)vector, NX, sizeof(vector[0]));
/* Partition the vector in PARTS sub-vectors */
struct starpu_data_filter f =
{
    .filter_func = starpu_vector_filter_block,
    .nchildren = PARTS
};
starpu_data_partition(handle, &f);
```

The handle of a sub-data block of a composite data block can be retrieved by calling `starpu_data_get_child()`. Or the task submission first retrieves the number of sub-data blocks in a composite data block by calling `starpu_data_get_nb_children()` and then uses the function `starpu_data_get_sub_data()` or `starpu_data_vget_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 by using `starpu_data_map_filters()` or `starpu_data_vmap_filters()` or `starpu_data_map_filters_parray()` or `starpu_data_map_filters_array()`, 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, where the kernel parameters are not pointers, but `cl_mem` handles. The kernel thus needs to be also passed the offset within the OpenCL buffer:

```
void opencil_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 opencil_kernel(__global int *vector, unsigned offset)
{
    block = (__global void *)block + offset;
    ...
}
```

When the sub-data is not of the same type as the original data, the field `starpu_data_filter::get_child_ops` needs to be set appropriately for StarPU to know which type should be used.

`starpu_data_unpartition()` should be called in the end to collect back the sub-pieces of data into the original piece of data.

StarPU provides various interfaces and filters for matrices, vectors, etc., but applications can also write their own data interfaces and filters, see `examples/interface` and `examples/filters/custom_mf` for an example, and see `DefiningANewDataInterface` and `DefiningANewDataFilter` for documentation.

6.3 Asynchronous Partitioning

The partitioning functions described in the previous section are synchronous: `starpu_data_partition()` and `starpu_data_unpartition()` both wait for all the tasks currently working on the data. This can be a bottleneck for the application.

An asynchronous API also exists, it works only on handles with sequential consistency. The principle is to first plan the partitioning, which returns data handles of the partition, which are not functional yet. When submitting tasks, one can mix using the handles of the partition or the whole data. One can even partition recursively and mix using

handles at different levels of the recursion. Of course, StarPU will have to introduce coherency synchronization. `examples/filters/fmultiple_submit_implicit.c` is a complete example using this technique. One can also look at `examples/filters/fmultiple_submit_readonly.c` which contains the explicit coherency synchronization which are automatically introduced by StarPU for `examples/filters/fmultiple_submit_implicit.c`.

In short, we first register a matrix and plan the partitioning:

```
starpu_matrix_data_register(&handle, STARPU_MAIN_RAM, (uintptr_t)matrix, NX, NX, NY, sizeof(matrix[0]));
struct starpu_data_filter f_vert =
{
    .filter_func = starpu_matrix_filter_block,
    .nchildren = PARTS
};
starpu_data_partition_plan(handle, &f_vert, vert_handle);
```

`starpu_data_partition_plan()` returns the handles for the partition in `vert_handle`.

One can then submit tasks working on the main handle, and tasks working on the sub handles `vert_handle`. Between using the main handle and the handles `vert_handle`, StarPU will automatically call `starpu_data_partition_submit()` and `starpu_data_unpartition_submit()`. Or call `starpu_data_partition_submit_sequential_consistency()` and `starpu_data_unpartition_submit_sequential_consistency()` to specify the coherency to be used for the main handle, or call `starpu_data_unpartition_submit_sequential_consistency_cb()` to specify a callback function for the unpartitioning task. One can also call `starpu_data_partition_readonly_submit()` and `starpu_data_unpartition_readonly_submit()` which do not guarantee coherency if the application attempts to write to the main handle or any of its sub-handles while a task is still running. However, in read-only case we can also call `starpu_data_partition_readonly_submit_sequential_consistency()` to specify the coherency to be used for the main handle, or call `starpu_data_partition_readwrite_upgrade_submit()` to upgrade the partitioning of a data handle from read-only to read-write mode for a specific sub-handle. If users want to specify that the data won't be touched in write mode anymore and use multiple partition of the data at the same time, they can call `starpu_data_partition_readonly_downgrade_submit()`.

After the task has completed using the data partition, `starpu_data_partition_clean()` or `starpu_data_partition_clean_node()` is used to clean up a data partition on the local node or on a specific node.

All this code is asynchronous, just submitting which tasks, partitioning and unpartitioning will be done at runtime.

Planning several partitioning of the same data is also possible, StarPU will unpartition and repartition as needed when mixing accesses of different partitions. If data access is done in read-only mode, StarPU will allow the different partitioning to coexist. As soon as a data is accessed in read-write mode, StarPU will automatically unpartition everything and activate only the partitioning leading to the data being written to.

For instance, for a stencil application, one can split a subdomain into its interior and halos, and then just submit a task updating the whole subdomain, then submit MPI sends/receives to update the halos, then submit again a task updating the whole subdomain, etc. and StarPU will automatically partition/unpartition each time.

6.4 Commute Data Access

By default, the implicit dependencies computed from data access use the sequential semantic. Notably, write accesses are always serialized in the order of submission. In some applicative cases, the write contributions can actually be performed in any order without affecting the eventual result. In this case, it is useful to drop the strictly sequential semantic, to improve parallelism by allowing StarPU to reorder the write accesses. This can be done by using the data access flag `STARPU_COMMUTE`. Accesses without this flag will however properly be serialized against accesses with this flag. For instance:

```
starpu_task_insert(&c11, STARPU_R, h, STARPU_RW, handle, 0);
starpu_task_insert(&c12, STARPU_R, handle1, STARPU_RW|STARPU_COMMUTE, handle, 0);
starpu_task_insert(&c12, STARPU_R, handle2, STARPU_RW|STARPU_COMMUTE, handle, 0);
starpu_task_insert(&c13, STARPU_R, g, STARPU_RW, handle, 0);
```

The two tasks running `c12` will be able to commute: depending on whether the value of `handle1` or `handle2` becomes available first, the corresponding task running `c12` will start first. The task running `c11` will however always be run before them, and the task running `c13` will always be run after them.

`tests/datawizard/commute2.c` is a complete example using the data access flag.

If a lot of tasks use the commute access on the same set of data and a lot of them are ready at the same time, it may become interesting to use an arbiter, see [Concurrent Data Accesses](#).

6.5 Data Reduction

In various cases, some piece of data is used to accumulate intermediate results. For instances, the dot product of a vector, maximum/minimum finding, the histogram of a picture, etc. When these results are produced along the

whole machine, it would not be efficient to accumulate them in only one place, incurring data transmission each and access concurrency.

StarPU provides a mode `STARPU_REDUX`, which permits to optimize this case: it will allocate a buffer on each worker (lazily), and accumulate intermediate results there. When the data is eventually accessed in the normal mode `STARPU_R`, StarPU will collect the intermediate results in just one buffer.

The function `starpu_data_set_reduction_methods()` must be called to specify how to initialize these buffers, and how to assemble partial results. The function `starpu_data_set_reduction_methods_with_args()` can also be used to pass arguments to the reduction and init tasks.

For instance, `examples/cg/cg.c` uses that to optimize its dot product: it first defines the codelets for initialization and reduction:

```
struct starpu_codelet bzero_variable_cl =
{
    .cpu_funcs = { bzero_variable_cpu },
    .cpu_funcs_name = { "bzero_variable_cpu" },
    .cuda_funcs = { bzero_variable_cuda },
    .nbuffers = 1,
}
static void accumulate_variable_cpu(void *descr[], void *cl_arg)
{
    double *v_dst = (double *)STARPU_VARIABLE_GET_PTR(descr[0]);
    double *v_src = (double *)STARPU_VARIABLE_GET_PTR(descr[1]);
    *v_dst = *v_dst + *v_src;
}
static void accumulate_variable_cuda(void *descr[], void *cl_arg)
{
    double *v_dst = (double *)STARPU_VARIABLE_GET_PTR(descr[0]);
    double *v_src = (double *)STARPU_VARIABLE_GET_PTR(descr[1]);
    cublasaxpy(1, (double)1.0, v_src, 1, v_dst, 1);
    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
struct starpu_codelet accumulate_variable_cl =
{
    .cpu_funcs = { accumulate_variable_cpu },
    .cpu_funcs_name = { "accumulate_variable_cpu" },
    .cuda_funcs = { accumulate_variable_cuda },
    .nbuffers = 2,
    .modes = { STARPU_RW|STARPU_COMMUTE, STARPU_R },
}
```

and attaches them as reduction methods for its handle `dtq`:

```
starpu_variable_data_register(&dtq_handle, -1, NULL, sizeof(type));
starpu_data_set_reduction_methods(dtq_handle, &accumulate_variable_cl, &bzero_variable_cl);
```

and `dtq_handle` can now be used with the mode `STARPU_REDUX` for the dot products with partitioned vectors:

```
for (b = 0; b < nblocks; b++)
    starpu_task_insert(&dot_kernel_cl,
        STARPU_REDUX, dtq_handle,
        STARPU_R, starpu_data_get_sub_data(v1, 1, b),
        STARPU_R, starpu_data_get_sub_data(v2, 1, b),
        0);
```

During registration, we have here provided `NULL`, i.e. there is no initial value to be taken into account during reduction. StarPU will thus only take into account the contributions from the tasks `dot_kernel_cl`. Also, it will not allocate any memory for `dtq_handle` before the tasks `dot_kernel_cl` are ready to run.

If another dot product has to be performed, one could unregister `dtq_handle`, and re-register it. But one can also call `starpu_data_deinitialize_submit()` or even `starpu_data_invalidate_submit()` with the parameter `dtq_handle`, which will clear all data from the handle, thus resetting it back to the initial status `register(NULL)`.

The example `examples/cg/cg.c` also uses reduction for the blocked gemv kernel, leading to yet more relaxed dependencies and more parallelism.

`STARPU_REDUX` can also be passed to `starpu_mpi_task_insert()` in the MPI case. This will however not produce any MPI communication, but just pass `STARPU_REDUX` to the underlying `starpu_task_insert()`. `starpu_mpi_redux_data()` posts tasks which will reduce the partial results among MPI nodes into the MPI node which owns the data. The function can be called by users to benefit from fine-tuning such as priority setting. If users do not call this function, StarPU wraps up reduction patterns automatically. The following example shows a hypothetical application which collects partial results into data `res`, then uses it for other computation, before looping again with a new reduction where the wrap-up of the reduction pattern is explicit:

```
for (i = 0; i < 100; i++)
{
    starpu_mpi_task_insert(MPI_COMM_WORLD, &init_res, STARPU_W, res, 0);
    starpu_mpi_task_insert(MPI_COMM_WORLD, &work, STARPU_RW, A, STARPU_R, B, STARPU_REDUX, res, 0);
    starpu_mpi_redux_data(MPI_COMM_WORLD, res);
    starpu_mpi_task_insert(MPI_COMM_WORLD, &work2, STARPU_RW, B, STARPU_R, res, 0);
}
```

`starpu_mpi_redux_data()` is called automatically in various cases, including when a task reading the reduced handle is inserted through `starpu_mpi_task_insert()`. The previous example could avoid calling `starpu_mpi_redux_data()`. Default priority (0) is used. The reduction tree arity is decided based on the size of the data to reduce: a flat tree

is used with a small data (default to less than 1024 bytes), a binary tree otherwise. If the environment variable `STARPU_MPI_REDUX_ARITY_THRESHOLD` is set, the threshold between the size of a small data and a bigger data is modified. If the value is set to be negative, flat trees will always be used. If the value is set to 0, binary trees are used. Otherwise, the size of the data is compared to the size in the environment variable. Remaining distributed-memory reduction patterns are wrapped-up at the end of an application when calling `starpu_mpi_wait_for_all()`. More details about MPI reduction are show in Section MPIMpiRedux, and some examples for MPI data reduction are available in `mpi/examples/mpi_redux/`.

6.6 Concurrent Data Accesses

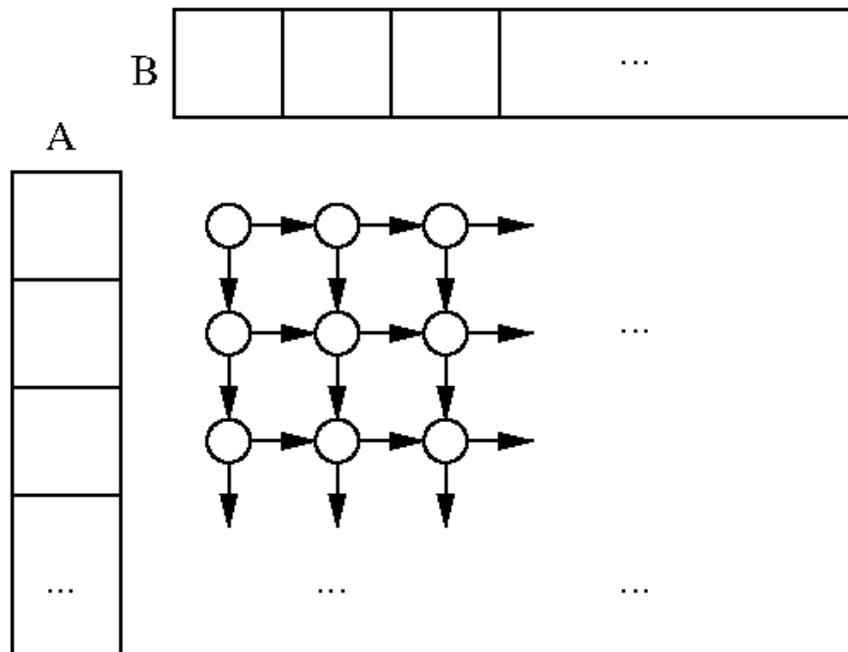
When several tasks are ready and will work on several data, StarPU is faced with the classical Dining Philosopher's problem, and has to determine the order in which it will run the tasks.

Data accesses usually use sequential ordering, so data accesses are usually already serialized, and thus by default, StarPU uses the Dijkstra solution which scales very well in terms of overhead: tasks will just acquire data one by one by data handle pointer value order.

When sequential ordering is disabled or the flag `STARPU_COMMUTE` is used, there may be a lot of concurrent accesses to the same data, and the Dijkstra solution gets only poor parallelism, typically in some pathological cases which do happen in various applications, for instance

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    task[i][j] = starpu_task_build(&cl, STARPU_RW|STARPU_COMMUTE, A[i], STARPU_RW|STARPU_COMMUTE, B[j],
    0);
```

It creates a series of tasks that are completely parallel in terms of tasks dependencies thanks to commutation, but StarPU still has to prevent two tasks from operating on the same data. The Dijkstra solution here leads to a worst-case: the `task[0][j]` tasks will wait for each other since they all access the same `A[0]`. And `task[1][0]` will wait for `task[0][0]` because they both access the same `B[0]`, `task[1][1]` will wait for `task[0][1]` because of `B[1]`, etc. In the end, no parallism is achieved:



In this case, one can use a data access arbiter `starpu_arbiter_t`, which implements the classical centralized solution for the Dining Philosophers problem. One can call `starpu_arbiter_create()` to create a data access arbiter, and `starpu_data_assign_arbiter()` to make access to handle managed by arbiter. Once the application no longer needs the arbiter, one can call `starpu_arbiter_destroy()` to destroy the arbiter after all data assigned to the arbiter have been unregistered. This is more expensive in terms of overhead since it is centralized, but it opportunisticly gets a lot of parallelism. The centralization can also be avoided by using several arbiters, thus separating sets of data for which arbitration will be done. If a task accesses data from different arbiters, it will acquire them arbiter by arbiter, in arbiter pointer value order.

See the `tests/datawizard/test_arbiter.cpp` example.

Arbiters however do not support the flag `STARPU_REDUX` yet.

6.7 Temporary Buffers

There are two kinds of temporary buffers: temporary data which just pass results from a task to another, and scratch data which are needed only internally by tasks.

6.7.1 Temporary Data

Data can be produced by a task, and consumed by another task, without being used by other parts of the application. In such case, registration can be done without prior allocation, by using the special memory node number `-1`, and passing a `NULL` pointer. StarPU will actually allocate memory only when the task creating the content gets scheduled, and destroy it on unregistration.

As the application will not use the data, it can be tedious for the application to have to unregister it. The unregistration can be done lazily by using the function `starpu_data_unregister_submit()`, which will record that no other tasks accessing the handle will be submitted, so that it can be freed as soon as the last task accessing it is completed.

The following code exemplifies both points: it registers the temporary data, submits three tasks accessing it, and records the data for automatic unregistration.

```
starpu_vector_data_register(&handle, -1, NULL, n, sizeof(float));
starpu_task_insert(&produce_data, STARPU_W, handle, 0);
starpu_task_insert(&compute_data, STARPU_RW, handle, 0);
starpu_task_insert(&summarize_data, STARPU_R, handle, STARPU_W, result_handle, 0);
starpu_data_unregister_submit(handle);
```

The application may also want for the temporary data to be initialized on the fly before being used by the task. This can be done by using `starpu_data_set_reduction_methods()` to set an initialization codelet (no redux codelet is needed).

6.7.2 Scratch Data

Some kernels sometimes need temporary data to complete the computations, like a workspace. The application could allocate it at the start of the codelet function, and free it at the end, but this would be costly. It could also allocate one buffer per worker (similarly to `HowToInitializeAComputationLibraryOnceForEachWorker`), but this would make them systematic and permanent. A more optimized way is to use the data access mode `STARPU_SCRATCH`, as exemplified below, which provides per-worker buffers without content consistency. The buffer is registered only once, using memory node `-1`, i.e. the application didn't allocate memory for it, and StarPU will allocate it on demand at task execution.

```
starpu_variable_data_register(&workspace, -1, NULL, sizeof(float));
for (i = 0; i < N; i++)
    starpu_task_insert(&compute, STARPU_R, input[i], STARPU_SCRATCH, workspace, STARPU_W, output[i], 0);
```

StarPU will make sure that the buffer is allocated before executing the task, and make this allocation per-worker: for CPU workers, notably, each worker has its own buffer. This means that each task submitted above will actually have its own workspace, which will actually be the same for all tasks running one after the other on the same worker. Also, if for instance memory becomes scarce, StarPU will notice that it can free such buffers easily, since the content does not matter.

The example `examples/pi` uses scratches for some temporary buffer.

It may be useful to additionally use the `STARPU_NOFOOTPRINT` flag, when this buffer may have various size depending e.g. on specific CUDA versions or devices, to make it simpler to use performance models for simulated execution. See for instance `examples/cholesky/cholesky_kernels.c`

Chapter 7

Scheduling

7.1 Task Scheduling Policies

The basics of the scheduling policy are the following:

- The scheduler gets to schedule tasks (`push` operation) when they become ready to be executed, i.e. they are not waiting for some tags, data dependencies or task dependencies.
- Workers pull tasks (`pop` operation) one by one from the scheduler.

This means scheduling policies usually contain at least one queue of tasks to store them between the time when they become available, and the time when a worker gets to grab them.

By default, StarPU uses the work-stealing scheduler **lws**. This is because it provides correct load balance and locality even if the application codelets do not have performance models. Other non-modelling scheduling policies can be selected among the list below, thanks to the environment variable `STARPU_SCHED`. For instance, `export STARPU_SCHED=dmda`. Use `help` to get the list of available schedulers.

The function `starpu_sched_get_predefined_policies()` returns a NULL-terminated array of all predefined scheduling policies that are available in StarPU. Functions `starpu_sched_get_sched_policy_in_ctx()` and `starpu_sched_get_sched_policy()` return the scheduling policy of a task within a specific context or a default context, respectively.

7.1.1 Non Performance Modelling Policies

- The **eager** scheduler uses a central task queue, from which all workers draw tasks to work on concurrently. 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 **random** scheduler uses a queue per worker, and distributes tasks randomly according to assumed worker overall performance.
- The **ws** (work stealing) scheduler uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from the most loaded worker.
- The **lws** (locality work stealing) scheduler uses a queue per worker, and schedules a task on the worker which released it by default. When a worker becomes idle, it steals a task from neighbor workers. It also takes priorities into account.
- The **prio** scheduler also uses a central task queue, but sorts tasks by priority specified by the application.
- The **heteroprio** scheduler uses different priorities for the different processing units. This scheduler must be configured to work correctly and to expect high-performance as described in the corresponding section.

7.1.2 Performance Model-Based Task Scheduling Policies

If (**and only if**) your **codelets have performance models** (`PerformanceModelExample`), you should change the scheduler thanks to the environment variable `STARPU_SCHED`, to select one of the policies below, in order to take

advantage of StarPU's performance modelling. For instance, `export STARPU_SCHED=dmda`. Use `help` to get the list of available schedulers.

Note: Depending on the performance model type chosen, some preliminary calibration runs may be needed for the model to converge. If the calibration has not been done, or is insufficient yet, or if no performance model is specified for a codelet, every task built from this codelet will be scheduled using an **eager** fallback policy.

Troubleshooting: Configuring and recompiling StarPU using the `configure` option `--enable-verbose` displays some statistics at the end of execution about the percentage of tasks which have been scheduled by a DM* family policy using performance model hints. A low or zero percentage may be the sign that performance models are not converging or that codelets do not have performance models enabled.

- The **dm** (deque model) scheduler takes task execution performance models into account to perform a HEFT-similar scheduling strategy: it schedules tasks where their termination time will be minimal. The difference with HEFT is that **dm** schedules tasks as soon as they become available, and thus in the order they become available, without taking priorities into account.
- The **dmda** (deque model data aware) scheduler is similar to **dm**, but it also takes data transfer time into account.
- The **dmdap** (deque model data aware prio) scheduler is similar to **dmda**, except that it sorts tasks by priority order, which allows becoming even closer to HEFT by respecting priorities after having made the scheduling decision (but it still schedules tasks in the order they become available).
- The **dmdar** (deque model data aware ready) scheduler is similar to **dmda**, but it also privileges tasks whose data buffers are already available on the target device.
- The **dmdas** combines **dmdap** and **dmdar**: it sorts tasks by priority order, but for a given priority it will privilege tasks whose data buffers are already available on the target device.
- The **dmdasd** (deque model data aware sorted decision) scheduler is similar to **dmdas**, except that when scheduling a task, it takes into account its priority when computing the minimum completion time, since this task may get executed before others, and thus the latter should be ignored.
- The **heft** (heterogeneous earliest finish time) scheduler is a deprecated alias for **dmda**.
- The **pheft** (parallel HEFT) scheduler is similar to **dmda**, it also supports parallel tasks (still experimental). It should not be used when several contexts using it are being executed simultaneously.
- The **peager** (parallel eager) scheduler is similar to **eager**, it also supports parallel tasks (still experimental). It should not be used when several contexts using it are being executed simultaneously.

7.1.3 Modularized Schedulers

StarPU provides a powerful way to implement schedulers, as documented in [DefiningANewModularSchedulingPolicy](#). It is currently shipped with the following pre-defined Modularized Schedulers :

- **modular-eager** , **modular-eager-prefetching** are eager-based Schedulers (without and with prefetching), they are naive schedulers, which try to map a task on the first available resource they find. The prefetching variant queues several tasks in advance to be able to do data prefetching. This may however degrade load balancing a bit.
- **modular-prio**, **modular-prio-prefetching**, **modular-eager-prio** are prio-based Schedulers (without / with prefetching);, similar to Eager-Based Schedulers. They can handle tasks which have a defined priority and schedule them accordingly. The **modular-eager-prio** variant integrates the eager and priority queue in a single component. This allows it to do a better job at pushing tasks.
- **modular-random**, **modular-random-prio**, **modular-random-prefetching**, **modular-random-prio-prefetching** are random-based Schedulers (without/with prefetching) : Select randomly a resource to be mapped on for each task.
- **modular-ws**) implements Work Stealing: Maps tasks to workers in round-robin, but allows workers to steal work from other workers.

- **modular-heft**, **modular-heft2**, and **modular-heft-prio** are HEFT Schedulers :
Maps tasks to workers using a heuristic very close to Heterogeneous Earliest Finish Time. It needs that every task submitted to StarPU have a defined performance model (PerformanceModelCalibration) to work efficiently, but can handle tasks without a performance model. **modular-heft** just takes tasks by order. **modular-heft2** takes at most 5 tasks of the same priority and checks which one fits best. **modular-heft-prio** is similar to **modular-heft**, but only decides the memory node, not the exact worker, just pushing tasks to one central queue per memory node. By default, they sort tasks by priorities and privilege, running first a task which has most of its data already available on the target. These can however be changed with `STARPU_SCHED_↔SORTED_ABOVE`, `STARPU_SCHED_SORTED_BELOW`, and `STARPU_SCHED_READY` .
- **modular-heteroprio** is a Heteroprio Scheduler:
Maps tasks to worker similarly to HEFT, but first attribute accelerated tasks to GPUs, then not-so-accelerated tasks to CPUs.

7.2 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 scheduler **dmda** 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 the tool `starpu_calibrate_bus`. The beta parameter defaults to 1, but it can be worth trying to tweak it by using `export STARPU_SCHED_BETA=2` (`STARPU_SCHED_BETA`) 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.

Chapter 8

Examples in StarPU Sources

We have already seen some examples in Chapter [Basic Examples](#). A tutorial is also installed in the directory `share/doc/starpu/tutorial/`.

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

incrementer/ Trivial incrementation test.

basic_examples/ Simple documented Hello world and vector/scalar product (as shown in [Basic Examples](#)), matrix product examples (as shown in `PerformanceModelExample`), 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.

native_fortran/ Example of using StarPU's native Fortran support.

fortran90/ Example of Fortran 90 bindings, using C marshalling wrappers.

fortran/ Example of Fortran 77 bindings, using C marshalling wrappers.

More advanced examples include:

filters/ Examples using filters, as shown in [Partitioning Data](#).

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

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

Part I

Appendix

Chapter 9

The 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.

1. 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 noncommercially. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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".

7. 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.

8. 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.

9. 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.

10. 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.

11. 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.

12. RELICENSING

Massive Multiauthor Collaboration Site'' (orMMC 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'' (orMMC") 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.

9.1 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.