StarPU Handbook

for StarPU 1.1.3

Generated by Doxygen *doxygenversionon*datetime

This manual documents the usage of StarPU version 1.1.3. Its contents was last updated on 20 August 2014.

Contents

Chapter 1

Introduction

1.1 Motivation

Part I

Using StarPU

Chapter 2

Building and Installing StarPU

2.1 Installing a Binary Package

One of the StarPU developers being a Debian Developer, the packages are well integrated and very uptodate. To see which packages are available, simply type:

```
$ apt-cache search starpu
```

To install what you need, type:

```
$ sudo apt-get install libstarpu-1.1 libstarpu-dev
```

2.2 Installing from Source

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

2.2.1 Optional Dependencies

The hwloc topology discovery library is not mandatory to use StarPU but strongly recommended. It allows for topology aware scheduling, which improves performance. hwloc is available in major free operating system distributions, and for most operating systems.

If hwloc is not available on your system, the option --without-hwloc should be explicitely given when calling the configure script. If hwloc is installed with a pkg-config file, no option is required, it will be detected automatically, otherwise --with-hwloc should be used to specify the location of hwloc.

2.2.2 Getting Sources

StarPU's sources can be obtained from the download page of the StarPU website.

All releases and the development tree of StarPU are freely available on INRIA's gforge under the LGPL license. Some releases are available under the BSD license.

The latest release can be downloaded from the INRIA's gforge or directly from the StarPU download page.

The latest nightly snapshot can be downloaded from the StarPU gforge website.

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

And finally, current development version is also accessible via svn. It should be used only if you need the very latest changes (i.e. less than a day!). Note that the client side of the software Subversion can be obtained from http://subversion.tigris.org. If you are running on Windows, you will probably prefer to use TortoiseSVN.

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

2.2.3 Configuring StarPU

Running `autogen.sh` is not necessary when using the tarball releases of StarPU. If you are using the source code from the svn repository, you first need to generate the configure scripts and the Makefiles. This requires the availability of `autoconf`, `automake` >= 2.60.

```
$ ./autogen.sh
```

You then need to configure StarPU. Details about options that are useful to give to `./configure` are given in Compilation Configuration.

```
$ ./configure
```

If `configure` does not detect some software or produces errors, please make sure to post the content of `config.log` when reporting the issue.

By default, the files produced during the compilation are placed in the source directory. As the compilation generates a lot of files, it is advised to to put them all in a separate directory. It is then easier to cleanup, and this allows to compile several configurations out of the same source tree. For that, simply enter the directory where you want the compilation to produce its files, and invoke the `configure` script located in the StarPU source directory.

```
$ mkdir build
$ cd build
$ ../configure
```

By default, StarPU will be installed in `/usr/local/bin`, `/usr/local/lib`, etc. You can specify an installation prefix other than `/usr/local` using the option `-prefix`, for instance:

```
$ ../configure --prefix=$HOME/starpu
```

2.2.4 Building StarPU

```
$ make
```

Once everything is built, you may want to test the result. An extensive set of regression tests is provided with StarPU. Running the tests is done by calling `make check`. These tests are run every night and the result from the main profile is publicly available.

```
$ make check
```

2.2.5 Installing StarPU

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

```
$ make install
```

Libtool interface versioning information are included in libraries names (`libstarpu-1.1.so`, `libstarpumpi-1.1.so` and `libstarpufft-1.1.so`).

2.3 Setting up Your Own Code

2.3.1 Setting Flags for Compiling, Linking and Running Applications

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

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

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

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

```
$ pkg-config --cflags starpu-1.1  # options for the compiler
$ pkg-config --libs starpu-1.1    # options for the linker
```

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

Make sure that `pkg-config -libs starpu-1.1` actually produces some output before going further: `PKG_CONFIG_PATH` has to point to the place where `starpu-1.1.pc` was installed during `make install`.

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

It is also necessary to set the environment variable `LD_LIBRARY_PATH` to locate dynamic libraries at runtime.

```
$ LD_LIBRARY_PATH=$prefix_dir/lib:$LD_LIBRARY_PATH
```

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

```
CFLAGS          +=      $$(pkg-config --cflags starpu-1.1)
LDFLAGS         +=      $$(pkg-config --libs starpu-1.1)
```

2.3.2 Running a Basic StarPU Application

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

```
$ ./examples/basic_examples/vector_scal
BEFORE: First element was 1.000000
AFTER: First element is 3.140000
```

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

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

2.3.3 Kernel Threads Started by StarPU

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

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

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

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

2.3.4 Enabling OpenCL

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

To enable OpenCL, you need either to disable CUDA when configuring StarPU:

```
$ ./configure --disable-cuda
```

or when running applications:

```
$ STARPU_NCUDA=0 ./application
```

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

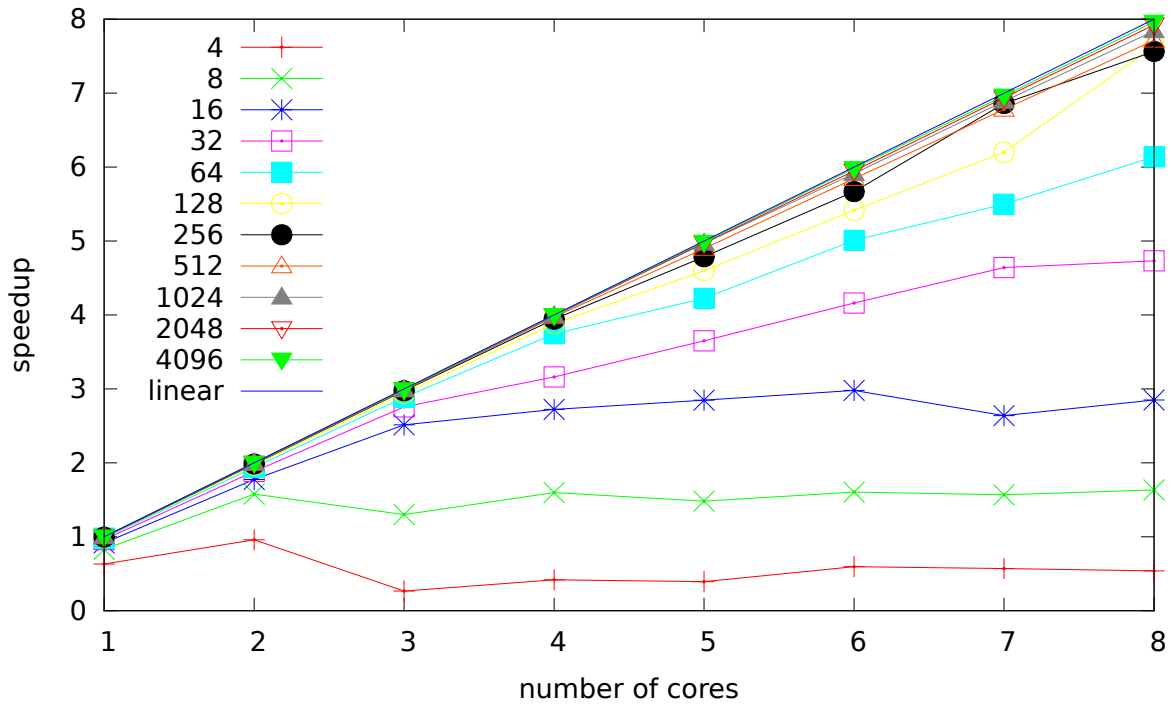
```
$ STARPU_NCUDA=2 ./application
```

2.4 Benchmarking StarPU

Some interesting benchmarks are installed among examples in `$prefix_dir/lib/starpu/examples/`. Make sure to try various schedulers, for instance `STARPU_SCHED=dmda`.

2.4.1 Task Size Overhead

This benchmark gives a glimpse into how long a task should be (in µs) for StarPU overhead to be low enough to keep efficiency. Run `tasks_size_overhead.sh`, it will generate a plot of the speedup of tasks of various sizes, depending on the number of CPUs being used.



2.4.2 Data Transfer Latency

`local_pingpong` performs a ping-pong between the first two CUDA nodes, and prints the measured latency.

2.4.3 Matrix-Matrix Multiplication

`sgemm` and `dgemm` perform a blocked matrix-matrix multiplication using BLAS and cuBLAS. They output the obtained GFlops.

2.4.4 Cholesky Factorization

`cholesky_*` perform a Cholesky factorization (single precision). They use different dependency primitives.

2.4.5 LU Factorization

`lu_*` perform an LU factorization. They use different dependency primitives.

Chapter 3

Basic Examples

3.1 Hello World Using The C Extension

This section shows how to implement a simple program that submits a task to StarPU using the StarPU C extension (C Extensions). The complete example, and additional examples, is available in the directory `gcc-plugin/examples` of the StarPU distribution. A similar example showing how to directly use the StarPU's API is shown in Hello World Using StarPU's API.

GCC from version 4.5 permit to use the StarPU GCC plug-in (C Extensions). This makes writing a task both simpler and less error-prone. In a nutshell, all it takes is to declare a task, declare and define its implementations (for CPU, OpenCL, and/or CUDA), and invoke the task like a regular C function. The example below defines `my_task` which has a single implementation for CPU:

```
#include <stdio.h>

/* Task declaration.  */
static void my_task (int x) __attribute__ ((task));

/* Definition of the CPU implementation of 'my_task'.  */
static void my_task (int x)
{
  printf ("Hello, world!  With x = %d\n", x);
}

int main ()
{
  /* Initialize StarPU. */
#pragma starpu initialize

  /* Do an asynchronous call to 'my_task'. */
  my_task (42);

  /* Wait for the call to complete.  */
#pragma starpu wait

  /* Terminate. */
#pragma starpu shutdown

  return 0;
}
```

The code can then be compiled and linked with GCC and the flag `-fplugin`:

```
$ gcc `pkg-config starpu-1.1 --cflags` hello-starpu.c \
    -fplugin=`pkg-config starpu-1.1 --variable=gccplugin` \
    `pkg-config starpu-1.1 --libs`
```

The code can also be compiled without the StarPU C extension and will behave as a normal sequential code.

```
$ gcc hello-starpu.c
hello-starpu.c:33:1: warning: 'task' attribute directive ignored [-Wattributes]
$ ./a.out
Hello, world! With x = 42
```

As can be seen above, the C extensions allows programmers to use StarPU tasks by essentially annotating “regular” C code.

3.2 Hello World Using StarPU’s API

This section shows how to achieve the same result as in the previous section using StarPU’s standard C API.

3.2.1 Required Headers

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

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

3.2.2 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 examplified above.

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 the 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, NULL },
    .nbuffers = 0
};
```

3.2.3 Submitting A Task

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

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

Chapter 4

Advanced Examples

4.1 Using Multiple Implementations Of A Codelet

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

```
#include <xmmintrin.h>

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

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

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


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

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

4.2 Enabling Implementation According To Capabilities

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

```
static int can_execute(unsigned workerid, struct starpu_task *task, unsigned nimpl)
{
  const struct cudaDeviceProp *props;
  if (starpu_worker_get_type(workerid) ==
      STARPU_CPU_WORKER)
    return 1;
  /* Cuda device */
  props = starpu_cuda_get_device_properties(workerid);
  if (props->major >= 2 || props->minor >= 3)
```

```
    /* At least compute capability 1.3, supports doubles */
    return 1;
  /* Old card, does not support doubles */
  return 0;
}

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

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

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

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

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

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

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

4.3 Task And Worker Profiling

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

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

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

/* The task is finished, get profiling information */
struct starpu_profiling_task_info *info = task->
```

```
      profiling_info;

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

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

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


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

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

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

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

4.4 Partitioning Data

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

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

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

/* Partition the vector in PARTS sub-vectors */
struct starpu_data_filter f =
{
    .filter_func = starpu_vector_filter_block,
    .nchildren = PARTS
};
starpu_data_partition(handle, &f);
```

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

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

    task->handles[0] = sub_handle;
    task->cl = &cl;
    task->synchronous = 1;
    task->cl_arg = &factor;
```

```
    task->cl_arg_size = sizeof(factor);

    starpu_task_submit(task);
}
```

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

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

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

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.

4.5 Performance Model Example

To achieve good scheduling, StarPU scheduling policies need to be able to estimate in advance the duration of a task. This is done by giving to codelets a performance model, by defining a structure starpu_perfmodel and providing its address in the field starpu_codelet::model. The fields starpu_perfmodel::symbol and starpu_perfmodel::type are mandatory, to give a name to the model, and the type of the model, since there are several kinds of performance models. For compatibility, make sure to initialize the whole structure to zero, either by using explicit memset(), or by letting the compiler implicitly do it as examplified below.

- Measured at runtime (model type STARPU_HISTORY_BASED). This assumes that for a given set of data input/output sizes, the performance will always be about the same. This is very true for regular kernels on GPUs for instance (<0.1% error), and just a bit less true on CPUs (~=1% error). This also assumes that there are few different sets of data input/output sizes. StarPU will then keep record of the average time of previous executions on the various processing units, and use it as an estimation. History is done per task size, by using a hash of the input and ouput sizes as an index. It will also save it in `$STAR↩ PU_HOME/.starpu/sampling/codelets` for further executions, and can be observed by using the tool `starpu_perfmodel_display`, or drawn by using the tool `starpu_perfmodel_plot` (Performance Model Calibration). The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. Measurements are only done when using a task scheduler which makes use of it, such as `dmda`. Measurements can also be provided explicitly by the application, by using the function starpu_perfmodel_update_↩ history().

  The following is a small code example.

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

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

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

- Measured at runtime and refined by regression (model types STARPU_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED). This still assumes performance regularity, but works with various data input sizes, by applying regression over observed execution times. STARPU_REGRESSION_BASED uses an a∗n^b regression form, STARPU_NL_REGRESSION_BASED uses an a∗n^b+c (more precise than STARPU_REGRESSION_BASED, but costs a lot more to compute).

  For instance, `tests/perfmodels/regression_based.c` uses a regression-based performance model for the function memset().

  Of course, the application has to issue tasks with varying size so that the regression can be computed. StarPU will not trust the regression unless there is at least 10% difference between the minimum and maximum observed input size. It can be useful to set the environment variable STARPU_CALIBRATE to `1` and run the application on varying input sizes with STARPU_SCHED set to `dmda` scheduler, so as to feed the performance model for a variety of inputs. The application can also provide the measurements explictly by using the function starpu_perfmodel_update_history(). The tools `starpu_perfmodel_display` and `starpu_perfmodel_plot` can be used to observe how much the performance model is calibrated (Performance Model Calibration); when their output look good, STARPU_CALIBRATE can be reset to `0` to let StarPU use the resulting performance model without recording new measures, and STARPU_SCHED can be set to `dmda` to benefit from the performance models. If the data input sizes vary a lot, it is really important to set STARPU_CALIBRATE to `0`, otherwise StarPU will continue adding the measures, and result with a very big performance model, which will take time a lot of time to load and save.

  For non-linear regression, since computing it is quite expensive, it is only done at termination of the application. This means that the first execution of the application will use only history-based performance model to perform scheduling, without using regression.

- Provided as an estimation from the application itself (model type STARPU_COMMON and field starpu_perfmodel::cost_function), see for instance `examples/common/blas_model.h` and `examples/common/blas_model.c`.

- Provided explicitly by the application (model type STARPU_PER_ARCH): the fields `.per_arch[arch][nimpl].cost_function` have to be filled with pointers to functions which return the expected duration of the task in micro-seconds, one per architecture.

For STARPU_HISTORY_BASED, STARPU_REGRESSION_BASED, and STARPU_NL_REGRESSION_BASED, the total size of task data (both input and output) is used as an index by default. The field starpu_perfmodel::size_base however permits the application to override that, when for instance some of the data do not matter for task cost (e.g. mere reference table), or when using sparse structures (in which case it is the number of non-zeros which matter), or when there is some hidden parameter such as the number of iterations, or when the application actually has a very good idea of the complexity of the algorithm, and just not the speed of the processor, etc. The example in the directory `examples/pi` uses this to include the number of iterations in the base.

StarPU will automatically determine when the performance model is calibrated, or rather, it will assume the performance model is calibrated until the application submits a task for which the performance can not be predicted. For STARPU_HISTORY_BASED, StarPU will require 10 (STARPU_CALIBRATE_MINIMUM) measurements for a given size before estimating that an average can be taken as estimation for further executions with the same size. For STARPU_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED, StarPU will require 10 (STARPU_CALIBRATE_MINIMUM) measurements, and that the minimum measured data size is smaller than 90% of the maximum measured data size (i.e. the measurement interval is large enough for a regression to have a meaning). Calibration can also be forced by setting the STARPU_CALIBRATE environment variable to `1`, or even reset by setting it to `2`.

How to use schedulers which can benefit from such performance model is explained in Task Scheduling Policy.

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

The application can request time estimations from the StarPU performance models by filling a task structure as usual without actually submitting it. The data handles can be created by calling any of the functions `starpu_*_data_register` with a `NULL` pointer and `-1` node and the desired data sizes, and need to be unregistered as usual. The functions starpu_task_expected_length() and starpu_task_expected_power() can then be called to get an estimation of the task cost on a given arch. starpu_task_footprint() can also be used to get the footprint used for indexing history-based performance models. starpu_task_destroy() needs to be called to destroy the dummy task afterwards. See `tests/perfmodels/regression_based.c` for an example.

4.6 Theoretical Lower Bound On Execution Time Example

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

The `deps` parameter tells StarPU whether to take tasks, implicit data, and tag dependencies into account. Tags released in a callback or similar are not taken into account, only tags associated with a task are. It must be understood that the linear programming problem size is quadratic with the number of tasks and thus the time to solve it will be very long, it could be minutes for just a few dozen tasks. You should probably use `lp_solve -timeout 1 test.pl -wmps test.mps` to convert the problem to MPS format and then use a better solver, `glpsol` might be better than `lp_solve` for instance (the `-pcost` option may be useful), but sometimes doesn't manage to converge. `cbc` might look slower, but it is parallel. For `lp_solve`, be sure to try at least all the `-B` options. For instance, we often just use `lp_solve -cc -B1 -Bb -Bg -Bp -Bf -Br -BG -Bd -Bs -BB -Bo -Bc -Bi` , and the `-gr` option can also be quite useful. The resulting schedule can be observed by using the tool `starpu_lp2paje`, which converts it into the Paje format.

Data transfer time can only be taken into account when `deps` is set. Only data transfers inferred from implicit data dependencies between tasks are taken into account. Other data transfers are assumed to be completely overlapped.

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

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

4.7 Insert Task Utility

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

Here the implementation of the codelet:

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

```
        float ffactor;

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

struct starpu_codelet mycodelet = {
        .where = STARPU_CPU,
        .cpu_funcs = { func_cpu, NULL },
        .nbuffers = 2,
        .modes = { STARPU_RW, STARPU_RW }
};
```

And the call to the function starpu_insert_task():

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

The call to starpu_insert_task() is equivalent to the following code:

```
struct starpu_task *task = starpu_task_create();
task->cl = &mycodelet;
task->handles[0] = data_handles[0];
task->handles[1] = data_handles[1];
char *arg_buffer;
size_t arg_buffer_size;
starpu_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);
```

Here a similar call using STARPU_DATA_ARRAY.

```
starpu_insert_task(&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_AC↩QUIRE_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_insert_task(&which_index, STARPU_W, i_handle, 0);

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

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

4.8 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 photograph, 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 that case: it will allocate a buffer on each memory node, 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.

For this to work, the user has to use the function starpu_data_set_reduction_methods() to declare how to initialize these buffers, and how to assemble partial results.

For instance, cg 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, NULL },
        .cuda_funcs = { bzero_variable_cuda, NULL },
        .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, NULL },
        .cuda_funcs = { accumulate_variable_cuda, NULL },
        .nbuffers = 1,
}
```

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 in mode STARPU_REDUX for the dot products with partitioned vectors:

```
for (b = 0; b < nblocks; b++)
    starpu_insert_task(&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 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_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 cg 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_insert_task() in the MPI case. That will however not produce any MPI communication, but just pass STARPU_REDUX to the underlying starpu_insert_task(). It is up to the application to call starpu_mpi_redux_data(), which posts tasks that will reduce the partial results among MPI nodes into the MPI node which owns the data. For instance, some hypothetical application which collects partial results into data res, then uses it for other computation, before looping again with a new reduction:

```
for (i = 0; i < 100; i++) {
    starpu_mpi_insert_task(MPI_COMM_WORLD, &init_res,
```

```
      STARPU_W, res, 0);
    starpu_mpi_insert_task(MPI_COMM_WORLD, &work,
      STARPU_RW, A,
                STARPU_R, B, STARPU_REDUX, res, 0);
    starpu_mpi_redux_data(MPI_COMM_WORLD, res);
    starpu_mpi_insert_task(MPI_COMM_WORLD, &work2,
      STARPU_RW, B, STARPU_R, res, 0);
}
```

4.9 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.

4.9.1 Temporary Data

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

In addition to that, it can be tedious for the application to have to unregister the data, since it will not use its content anyway. The unregistration can be done lazily by using the function starpu_data_unregister_submit(), which will record that no more tasks accessing the handle will be submitted, so that it can be freed as soon as the last task accessing it is over.

The following code examplifies 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, 0, n, sizeof(float));
starpu_insert_task(&produce_data, STARPU_W, handle, 0);
starpu_insert_task(&compute_data, STARPU_RW, handle, 0);
starpu_insert_task(&summarize_data, STARPU_R, handle,
      STARPU_W, result_handle, 0);
starpu_data_unregister_submit(handle);
```

The application may also want to see the temporary data 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).

4.9.2 Scratch Data

Some kernels sometimes need temporary data to achieve the computations, i.e. a workspace. The application could allocate it at the start of the codelet function, and free it at the end, but that would be costly. It could also allocate one buffer per worker (similarly to How To Initialize A Computation Library Once For Each Worker?), but that would make them systematic and permanent. A more optimized way is to use the data access mode STARPU_SCRATCH, as examplified below, which provides per-worker buffers without content consistency.

```
starpu_vector_data_register(&workspace, -1, 0, sizeof(float));
for (i = 0; i < N; i++)
    starpu_insert_task(&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 GPU 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.

4.10 Parallel Tasks

StarPU can leverage existing parallel computation libraries by the means of parallel tasks. A parallel task is a task which gets worked on by a set of CPUs (called a parallel or combined worker) at the same time, by using an existing parallel CPU implementation of the computation to be achieved. This can also be useful to improve the load balance between slow CPUs and fast GPUs: since CPUs work collectively on a single task, the completion time of tasks on CPUs become comparable to the completion time on GPUs, thus relieving from granularity discrepancy concerns. `hwloc` support needs to be enabled to get good performance, otherwise StarPU will not know how to better group cores.

Two modes of execution exist to accomodate with existing usages.

4.10.1 Fork-mode Parallel Tasks

In the Fork mode, StarPU will call the codelet function on one of the CPUs of the combined worker. The codelet function can use starpu_combined_worker_get_size() to get the number of threads it is allowed to start to achieve the computation. The CPU binding mask for the whole set of CPUs is already enforced, so that threads created by the function will inherit the mask, and thus execute where StarPU expected, the OS being in charge of choosing how to schedule threads on the corresponding CPUs. The application can also choose to bind threads by hand, using e.g. sched_getaffinity to know the CPU binding mask that StarPU chose.

For instance, using OpenMP (full source is available in `examples/openmp/vector_scal.c`):

```
void scal_cpu_func(void *buffers[], void *_args)
{
    unsigned i;
    float *factor = _args;
    struct starpu_vector_interface *vector = buffers[0];
    unsigned n = STARPU_VECTOR_GET_NX(vector);
    float *val = (float *)STARPU_VECTOR_GET_PTR(vector);

#pragma omp parallel for num_threads(starpu_combined_worker_get_size())
    for (i = 0; i < n; i++)
        val[i] *= *factor;
}

static struct starpu_codelet cl =
{
    .modes = { STARPU_RW },
    .where = STARPU_CPU,
    .type = STARPU_FORKJOIN,
    .max_parallelism = INT_MAX,
    .cpu_funcs = {scal_cpu_func, NULL},
    .nbuffers = 1,
};
```

Other examples include for instance calling a BLAS parallel CPU implementation (see `examples/mult/xgemm.c`).

4.10.2 SPMD-mode Parallel Tasks

In the SPMD mode, StarPU will call the codelet function on each CPU of the combined worker. The codelet function can use starpu_combined_worker_get_size() to get the total number of CPUs involved in the combined worker, and thus the number of calls that are made in parallel to the function, and starpu_combined_worker_get_rank() to get the rank of the current CPU within the combined worker. For instance:

```
static void func(void *buffers[], void *args)
{
    unsigned i;
    float *factor = _args;
    struct starpu_vector_interface *vector = buffers[0];
    unsigned n = STARPU_VECTOR_GET_NX(vector);
    float *val = (float *)STARPU_VECTOR_GET_PTR(vector);

    /* Compute slice to compute */
    unsigned m = starpu_combined_worker_get_size();
    unsigned j = starpu_combined_worker_get_rank();
    unsigned slice = (n+m-1)/m;
```

```
    for (i = j * slice; i < (j+1) * slice && i < n; i++)
        val[i] *= *factor;
}

static struct starpu_codelet cl =
{
    .modes = { STARPU_RW },
    .where = STARP_CPU,
    .type = STARPU_SPMD,
    .max_parallelism = INT_MAX,
    .cpu_funcs = { func, NULL },
    .nbuffers = 1,
}
```

Of course, this trivial example will not really benefit from parallel task execution, and was only meant to be simple to understand. The benefit comes when the computation to be done is so that threads have to e.g. exchange intermediate results, or write to the data in a complex but safe way in the same buffer.

4.10.3 Parallel Tasks Performance

To benefit from parallel tasks, a parallel-task-aware StarPU scheduler has to be used. When exposed to codelets with a flag STARPU_FORKJOIN or STARPU_SPMD, the schedulers `pheft` (parallel-heft) and `peager` (parallel eager) will indeed also try to execute tasks with several CPUs. It will automatically try the various available combined worker sizes (making several measurements for each worker size) and thus be able to avoid choosing a large combined worker if the codelet does not actually scale so much.

4.10.4 Combined Workers

By default, StarPU creates combined workers according to the architecture structure as detected by `hwloc`. It means that for each object of the `hwloc` topology (NUMA node, socket, cache, ...) a combined worker will be created. If some nodes of the hierarchy have a big arity (e.g. many cores in a socket without a hierarchy of shared caches), StarPU will create combined workers of intermediate sizes. The variable STARPU_SYNTHESIZE_ARITY_COMBINED_WORKER permits to tune the maximum arity between levels of combined workers.

The combined workers actually produced can be seen in the output of the tool `starpu_machine_display` (the environment variable STARPU_SCHED has to be set to a combined worker-aware scheduler such as `pheft` or `peager`).

4.10.5 Concurrent Parallel Tasks

Unfortunately, many environments and librairies do not support concurrent calls.

For instance, most OpenMP implementations (including the main ones) do not support concurrent `pragma omp parallel` statements without nesting them in another `pragma omp parallel` statement, but StarPU does not yet support creating its CPU workers by using such pragma.

Other parallel libraries are also not safe when being invoked concurrently from different threads, due to the use of global variables in their sequential sections for instance.

The solution is then to use only one combined worker at a time. This can be done by setting the field starpu_conf::single_combined_worker to `1`, or setting the environment variable STARPU_SINGLE_COMBINED_WORKER to `1`. StarPU will then run only one parallel task at a time (but other CPU and GPU tasks are not affected and can be run concurrently). The parallel task scheduler will however still however still try varying combined worker sizes to look for the most efficient ones.

4.11 Debugging

StarPU provides several tools to help debugging aplications. Execution traces can be generated and displayed graphically, see Generating Traces With FxT.

Some gdb helpers are also provided to show the whole StarPU state:

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

Valgrind can be used on StarPU: valgrind.h just needs to be found at ./configure time, to tell valgrind about some known false positives and disable host memory pinning. Other known false positives can be suppressed by giving the suppression files in tools/valgrind/ *.suppr to valgrind's –suppressions option.

The STARPU_DISABLE_KERNELS environment variable can also be set to 1 to make StarPU do everything (schedule tasks, transfer memory, etc.) except actually calling the application-provided kernel functions, i.e. the computation will not happen. This permits to quickly check that the task scheme is working properly.

The Temanejo task debugger can also be used, see Using The Temanejo Task Debugger.

4.12 The Multiformat Interface

It may be interesting to represent the same piece of data using two different data structures: one that would only be used on CPUs, and one that would only be used on GPUs. This can be done by using the multiformat interface. StarPU will be able to convert data from one data structure to the other when needed. Note that the scheduler dmda is the only one optimized for this interface. The user must provide StarPU with conversion codelets:

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

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

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

struct starpu_multiformat_data_interface_ops format_ops = {
#ifdef STARPU_USE_OPENCL
    .opencl_elemsize = 2 * sizeof(float),
    .cpu_to_opencl_cl = &cpu_to_opencl_cl,
    .opencl_to_cpu_cl = &opencl_to_cpu_cl,
#endif
    .cpu_elemsize = 2 * sizeof(float),
    ...
};

starpu_multiformat_data_register(handle, 0, &array_of_structs, NX, &
      format_ops);
```

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

```
static void
```

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

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

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

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

    ...
}
```

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

4.13 Using The Driver API

Running Drivers

```
int ret;
struct starpu_driver = {
    .type = STARPU_CUDA_WORKER,
    .id.cuda_id = 0
};
ret = starpu_driver_init(&d);
if (ret != 0)
    error();
while (some_condition) {
    ret = starpu_driver_run_once(&d);
    if (ret != 0)
        error();
}
ret = starpu_driver_deinit(&d);
if (ret != 0)
    error();
```

To add a new kind of device to the structure starpu_driver, one needs to:

1. Add a member to the union starpu_driver::id
2. Modify the internal function `_starpu_launch_drivers()` to make sure the driver is not always launched.
3. Modify the function starpu_driver_run() so that it can handle another kind of architecture.
4. Write the new function `_starpu_run_foobar()` in the corresponding driver.

4.14 Defining A New Scheduling Policy

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

See Scheduling Policy

```
static struct starpu_sched_policy dummy_sched_policy = {
    .init_sched = init_dummy_sched,
    .deinit_sched = deinit_dummy_sched,
    .add_workers = dummy_sched_add_workers,
    .remove_workers = dummy_sched_remove_workers,
    .push_task = push_task_dummy,
    .push_prio_task = NULL,
```

```
    .pop_task = pop_task_dummy,
    .post_exec_hook = NULL,
    .pop_every_task = NULL,
    .policy_name = "dummy",
    .policy_description = "dummy scheduling strategy"
};
```

4.15 On-GPU Rendering

Graphical-oriented applications need to draw the result of their computations, typically on the very GPU where these happened. Technologies such as OpenGL/CUDA interoperability permit to let CUDA directly work on the OpenGL buffers, making them thus immediately ready for drawing, by mapping OpenGL buffer, textures or renderbuffer objects into CUDA. CUDA however imposes some technical constraints: peer memcpy has to be disabled, and the thread that runs OpenGL has to be the one that runs CUDA computations for that GPU.

To achieve this with StarPU, pass the option --disable-cuda-memcpy-peer to `./configure` (TODO: make it dynamic), OpenGL/GLUT has to be initialized first, and the interoperability mode has to be enabled by using the field starpu_conf::cuda_opengl_interoperability, and the driver loop has to be run by the application, by using the field starpu_conf::not_launched_drivers to prevent StarPU from running it in a separate thread, and by using starpu_driver_run() to run the loop. The examples `gl_interop` and `gl_interop_idle` show how it articulates in a simple case, where rendering is done in task callbacks. The former uses `glutMainLoopEvent` to make GLUT progress from the StarPU driver loop, while the latter uses `glutIdleFunc` to make StarPU progress from the GLUT main loop.

Then, to use an OpenGL buffer as a CUDA data, StarPU simply needs to be given the CUDA pointer at registration, for instance:

```
/* Get the CUDA worker id */
for (workerid = 0; workerid < starpu_worker_get_count(); workerid++)
        if (starpu_worker_get_type(workerid) ==
      STARPU_CUDA_WORKER)
                break;

/* Build a CUDA pointer pointing at the OpenGL buffer */
cudaGraphicsResourceGetMappedPointer((void**)&output, &num_bytes, resource);

/* And register it to StarPU */
starpu_vector_data_register(&handle,
      starpu_worker_get_memory_node(workerid),
                          output, num_bytes / sizeof(float4), sizeof(float4));

/* The handle can now be used as usual */
starpu_insert_task(&cl, STARPU_RW, handle, 0);

/* ... */

/* This gets back data into the OpenGL buffer */
starpu_data_unregister(handle);
```

and display it e.g. in the callback function.

4.16 Defining A New Data Interface

Let's define a new data interface to manage complex numbers.

```
/* interface for complex numbers */
struct starpu_complex_interface
{
        double *real;
        double *imaginary;
        int nx;
};
```

Registering such a data to StarPU is easily done using the function starpu_data_register(). The last parameter of the function, `interface_complex_ops`, will be described below.

```
void starpu_complex_data_register(starpu_data_handle_t *handle,
     unsigned home_node, double *real, double *imaginary, int nx)
{
        struct starpu_complex_interface complex =
        {
                .real = real,
                .imaginary = imaginary,
                .nx = nx
        };

        if (interface_complex_ops.interfaceid ==
      STARPU_UNKNOWN_INTERFACE_ID)
        {
                interface_complex_ops.interfaceid =
      starpu_data_interface_get_next_id();
        }

        starpu_data_register(handleptr, home_node, &complex, &interface_complex_ops);
}
```

Different operations need to be defined for a data interface through the type starpu_data_interface_ops. We only define here the basic operations needed to run simple applications. The source code for the different functions can be found in the file `examples/interface/complex_interface.c`.

```
static struct starpu_data_interface_ops interface_complex_ops =
{
        .register_data_handle = complex_register_data_handle,
        .allocate_data_on_node = complex_allocate_data_on_node,
        .copy_methods = &complex_copy_methods,
        .get_size = complex_get_size,
        .footprint = complex_footprint,
        .interfaceid = STARPU_UNKNOWN_INTERFACE_ID,
        .interface_size = sizeof(struct starpu_complex_interface),
};
```

Functions need to be defined to access the different fields of the complex interface from a StarPU data handle.

```
double *starpu_complex_get_real(starpu_data_handle_t handle)
{
        struct starpu_complex_interface *complex_interface =
          (struct starpu_complex_interface *) starpu_data_get_interface_on_node
      (handle, 0);
        return complex_interface->real;
}

double *starpu_complex_get_imaginary(starpu_data_handle_t handle);
int starpu_complex_get_nx(starpu_data_handle_t handle);
```

Similar functions need to be defined to access the different fields of the complex interface from a `void *` pointer to be used within codelet implemetations.

```
#define STARPU_COMPLEX_GET_REAL(interface)      \
        (((struct starpu_complex_interface *)(interface))->real)
#define STARPU_COMPLEX_GET_IMAGINARY(interface) \
        (((struct starpu_complex_interface *)(interface))->imaginary)
#define STARPU_COMPLEX_GET_NX(interface)        \
        (((struct starpu_complex_interface *)(interface))->nx)
```

Complex data interfaces can then be registered to StarPU.

```
double real = 45.0;
double imaginary = 12.0;
starpu_complex_data_register(&handle1, 0, &real, &imaginary, 1);
starpu_insert_task(&cl_display, STARPU_R, handle1, 0);
```

and used by codelets.

```
void display_complex_codelet(void *descr[], __attribute__ ((unused)) void *_args)
{
        int nx = STARPU_COMPLEX_GET_NX(descr[0]);
        double *real = STARPU_COMPLEX_GET_REAL(descr[0]);
        double *imaginary = STARPU_COMPLEX_GET_IMAGINARY(descr[0]);
        int i;

        for(i=0 ; i<nx ; i++)
        {
                fprintf(stderr, "Complex[%d] = %3.2f + %3.2f i\n", i, real[i], imaginary[i]);
        }
}
```

The whole code for this complex data interface is available in the directory `examples/interface/`.

4.17 Setting The Data Handles For A Task

The number of data a task can manage is fixed by the environment variable STARPU_NMAXBUFS which has a default value which can be changed through the configure option --enable-maxbuffers.

However, it is possible to define tasks managing more data by using the field starpu_task::dyn_handles when defining a task and the field starpu_codelet::dyn_modes when defining the corresponding codelet.

```
enum starpu_data_access_mode modes[STARPU_NMAXBUFS+1] = {
        STARPU_R, STARPU_R, ...
};

struct starpu_codelet dummy_big_cl =
{
        .cuda_funcs = {dummy_big_kernel, NULL},
        .opencl_funcs = {dummy_big_kernel, NULL},
        .cpu_funcs = {dummy_big_kernel, NULL},
        .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_insert_task(&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 directory `examples/basic_↩ examples/dynamic_handles.c`.

4.18 Specifying a target node for task data

When executing a task on a GPU for instance, StarPU would normally copy all the needed data for the tasks on the embedded memory of the GPU. It may however happen that the task kernel would rather have some of the datas kept in the main memory instead of copied in the GPU, a pivoting vector for instance. This can be achieved by setting the starpu_codelet::specific_nodes flag to 1, and then fill the starpu_codelet::nodes array (or starpu_↩ codelet::dyn_nodes when starpu_codelet::nbuffers is greater than STARPU_NMAXBUFS) with the node numbers where data should be copied to, or -1 to let StarPU copy it to the memory node where the task will be executed. For instance, with the following codelet:

```
struct starpu_codelet cl =
{
        .cuda_funcs = { kernel, NULL },
        .nbuffers = 2,
        .modes = {STARPU_RW, STARPU_RW},
        .specific_nodes = 1,
        .nodes = {0, -1},
};
```

the first data of the task will be kept in the main memory, while the second data will be copied to the CUDA GPU as usual.

4.19 More Examples

More examples are 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 Performance Model Example), an example using the blocked matrix data interface, an example using the variable data interface, and an example using different formats on CPUs and GPUs.

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

`axpy/` AXPY CUBLAS operation adapted to StarPU.

`fortran/` Example of Fortran bindings.

More advanced examples include:

`filters/` Examples using filters, as shown in Partitioning Data.

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

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

Chapter 5

How To Optimize Performance With StarPU

TODO: improve!

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

5.1 Data Management

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

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

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

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

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

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

Setting the write-through mask to `~0U` can also be useful to make sure all memory nodes always have a copy of the data, so that it is never evicted when memory gets scarse.

Implicit data dependency computation can become expensive if a lot of tasks access the same piece of data. If no dependency is required on some piece of data (e.g. because it is only accessed in read-only mode, or because write accesses are actually commutative), use the function starpu_data_set_sequential_consistency_flag() to disable implicit dependencies on that data.

In the same vein, accumulation of results in the same data can become a bottleneck. The use of the mode STAR↩ PU_REDUX permits to optimize such accumulation (see Data Reduction).

Applications often need a data just for temporary results. In such a case, registration can be made without an initial value, for instance this produces a vector data:

```
starpu_vector_data_register(&handle, -1, 0, n, sizeof(float));
```

StarPU will then allocate the actual buffer only when it is actually needed, e.g. directly on the GPU without allocating in main memory.

In the same vein, once the temporary results are not useful any more, the data should be thrown away. If the handle is not to be reused, it can be unregistered:

```
starpu_data_unregister_submit(handle);
```

actual unregistration will be done after all tasks working on the handle terminate.

If the handle is to be reused, instead of unregistering it, it can simply be invalidated:

```
starpu_data_invalidate_submit(handle);
```

the buffers containing the current value will then be freed, and reallocated only when another task writes some value to the handle.

5.2 Task Granularity

Like any other runtime, StarPU has some overhead to manage tasks. Since it does smart scheduling and data management, that overhead is not always neglectable. The order of magnitude of the overhead is typically a couple of microseconds, which is actually quite smaller than the CUDA overhead itself. The amount of work that a task should do should thus be somewhat bigger, to make sure that the overhead becomes neglectible. The offline performance feedback can provide a measure of task length, which should thus be checked if bad performance are observed. To get a grasp at the scalability possibility according to task size, one can run `tests/microbenchs/tasks_size_overhead.sh` which draws curves of the speedup of independent tasks of very small sizes.

The choice of scheduler also has impact over the overhead: for instance, the scheduler `dmda` takes time to make a decision, while `eager` does not. `tasks_size_overhead.sh` can again be used to get a grasp at how much impact that has on the target machine.

5.3 Task Submission

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

5.4 Task Priorities

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

5.5 Task Scheduling Policy

The basics of the scheduling policy are that

- 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 simple greedy scheduler `eager`. This is because it provides correct load balance even if the application codelets do not have performance models. If your application codelets have performance models (Performance Model Example), you should change the scheduler thanks to the environment variable STARPU_SCHED. For instance `export STARPU_SCHED=dmda` . Use `help` to get the list of available schedulers.

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

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 **dm** (deque model) scheduler uses 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 into account data transfer time.

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

The **dmdas** (deque model data aware sorted) scheduler is similar to dmdar, except that it sorts tasks by priority order, which allows to become 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 **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). 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). Should not be used when several contexts using it are being executed simultaneously.

5.6 Performance Model Calibration

Most schedulers are based on an estimation of codelet duration on each kind of processing unit. For this to be possible, the application programmer needs to configure a performance model for the codelets of the application (see Performance Model Example for instance). History-based performance models use on-line calibration. StarPU will automatically calibrate codelets which have never been calibrated yet, and save the result in `$STARPU_HOME/.starpu/sampling/codelets`. The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. To force continuing calibration, use `export STARPU_CALIBRATE=1` . This may be necessary if your application has not-so-stable performance. StarPU will force calibration (and thus ignore the current result) until 10 (`_STARPU_CALIBRATION_MINIMUM`) measurements have been made on each architecture, to avoid badly scheduling tasks just because the first measurements were not so good. Details on the current performance model status can be obtained from the command `starpu_perfmodel_display`: the `-l` option lists the available performance models, and the `-s` option permits to choose the performance model to be displayed. The result looks like:

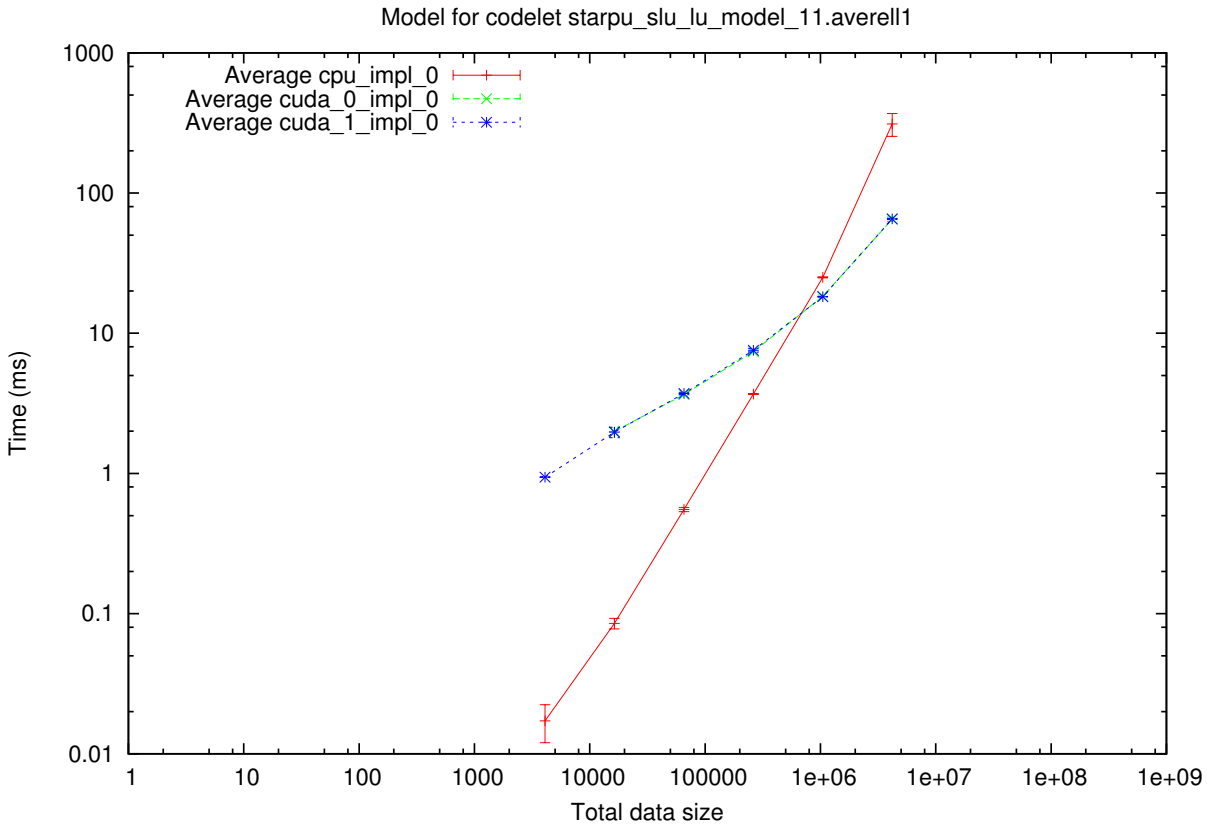
```
$ starpu_perfmodel_display -s starpu_slu_lu_model_11
performance model for cpu_impl_0
# hash    size     flops         mean          dev           n
914f3bef  1048576  0.000000e+00  2.503577e+04  1.982465e+02  8
3e921964  65536    0.000000e+00  5.527003e+02  1.848114e+01  7
e5a07e31  4096     0.000000e+00  1.717457e+01  5.190038e+00  14
```

```
...
```

Which shows that for the LU 11 kernel with a 1MiB matrix, the average execution time on CPUs was about 25ms, with a 0.2ms standard deviation, over 8 samples. It is a good idea to check this before doing actual performance measurements.

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

```
$ starpu_perfmodel_plot -s starpu_slu_lu_model_11
4096 16384 65536 262144 1048576 4194304
$ gnuplot starpu_starpu_slu_lu_model_11.gp
$ gv starpu_starpu_slu_lu_model_11.eps
```



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

Note: history-based performance models get calibrated only if a performance-model-based scheduler is chosen.

The history-based performance models can also be explicitly filled by the application without execution, if e.g. the application already has a series of measurements. This can be done by using starpu_perfmodel_update_history(), for instance:

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

struct starpu_codelet cl = {
    .where = STARPU_CUDA,
    .cuda_funcs = { cuda_func1, cuda_func2, NULL },
    .nbuffers = 1,
    .modes = {STARPU_W},
    .model = &perf_model
};

void feed(void) {
    struct my_measure *measure;
```

```
    struct starpu_task task;
    starpu_task_init(&task);

    task.cl = &cl;

    for (measure = &measures[0]; measure < measures[last]; measure++) {
        starpu_data_handle_t handle;
        starpu_vector_data_register(&handle, -1, 0, measure->size, sizeof(float)
      );
        task.handles[0] = handle;
        starpu_perfmodel_update_history(&perf_model, &task,
                                        STARPU_CUDA_DEFAULT + measure->cudadev, 0,
                                        measure->implementation, measure->time);
        starpu_task_clean(&task);
        starpu_data_unregister(handle);
    }
}
```

Measurement has to be provided in milliseconds for the completion time models, and in Joules for the energy consumption models.

5.7 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_execution + beta * T_data_transfer`, where `T_execution` is the estimated execution time of the codelet (usually accurate), and `T_data_transfer` is the estimated data transfer time. The latter is estimated based on bus calibration before execution start, i.e. with an idle machine, thus without contention. You can force bus re-calibration by running 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` for instance, since during real application execution, contention makes transfer times bigger. This is of course imprecise, but in practice, a rough estimation already gives the good results that a precise estimation would give.

5.8 Data Prefetch

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

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

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

5.9 Power-based Scheduling

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

This is however not sufficient to correctly optimize power: the scheduler would simply tend to run all computations on the most energy-conservative processing unit. To account for the consumption of the whole machine (including

idle processing units), the idle power of the machine should be given by setting `export STARPU_IDLE_POW↩ ER=200` for 200W, for instance. This value can often be obtained from the machine power supplier.

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

On-line task consumption measurement is currently only supported through the `CL_PROFILING_POWER_C↩ ONSUMED` OpenCL extension, implemented in the MoviSim simulator. Applications can however provide explicit measurements by using the function starpu_perfmodel_update_history() (examplified in Performance Model Example with the `power_model` performance model). Fine-grain measurement is often not feasible with the feedback provided by the hardware, so the user can for instance run a given task a thousand times, measure the global consumption for that series of tasks, divide it by a thousand, repeat for varying kinds of tasks and task sizes, and eventually feed StarPU with these manual measurements through starpu_perfmodel_update_history(). For instance, for CUDA devices, `nvidia-smi -q -d POWER` can be used to get the current consumption in Watt. Multiplying that value by the average duration of a single task gives the consumption of the task in Joules, which can be given to starpu_perfmodel_update_history().

5.10 Static Scheduling

In some cases, one may want to force some scheduling, for instance force a given set of tasks to GPU0, another set to GPU1, etc. while letting some other tasks be scheduled on any other device. This can indeed be useful to guide StarPU into some work distribution, while still letting some degree of dynamism. For instance, to force execution of a task on CUDA0:

```
task->execute_on_a_specific_worker = 1;
task->worker = starpu_worker_get_by_type(
      STARPU_CUDA_WORKER, 0);
```

Note however that using scheduling contexts while statically scheduling tasks on workers could be tricky. Be careful to schedule the tasks exactly on the workers of the corresponding contexts, otherwise the workers' corresponding scheduling structures may not be allocated or the execution of the application may deadlock. Moreover, the hypervisor should not be used when statically scheduling tasks.

5.11 Profiling

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

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

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

5.12 Detection Stuck Conditions

It may happen that for some reason, StarPU does not make progress for a long period of time. Reason are sometimes due to contention inside StarPU, but sometimes this is due to external reasons, such as stuck MPI driver, or CUDA driver, etc.

`export STARPU_WATCHDOG_TIMEOUT=10000`

allows to make StarPU print an error message whenever StarPU does not terminate any task for 10ms. In addition to that,

```
export STARPU_WATCHDOG_CRASH=1
```

raises SIGABRT in that condition, thus allowing to catch the situation in gdb. It can also be useful to type "handle SIGABRT nopass" in gdb to be able to let the process continue, after inspecting the state of the process.

5.13 CUDA-specific Optimizations

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

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

StarPU already does appropriate calls for the CUBLAS library.

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

5.14 Performance Debugging

To get an idea of what is happening, a lot of performance feedback is available, detailed in the next chapter. The various informations should be checked for.

- What does the Gantt diagram look like? (see Creating a Gantt Diagram)
  - If it's mostly green (tasks running in the initial context) or context specific color prevailing, then the machine is properly utilized, and perhaps the codelets are just slow. Check their performance, see Performance Of Codelets.
  - If it's mostly purple (FetchingInput), tasks keep waiting for data transfers, do you perhaps have far more communication than computation? Did you properly use CUDA streams to make sure communication can be overlapped? Did you use data-locality aware schedulers to avoid transfers as much as possible?
  - If it's mostly red (Blocked), tasks keep waiting for dependencies, do you have enough parallelism? It might be a good idea to check what the DAG looks like (see Creating a DAG With Graphviz).
  - If only some workers are completely red (Blocked), for some reason the scheduler didn't assign tasks to them. Perhaps the performance model is bogus, check it (see Performance Of Codelets). Do all your codelets have a performance model? When some of them don't, the schedulers switches to a greedy algorithm which thus performs badly.

You can also use the Temanejo task debugger (see Using The Temanejo Task Debugger) to visualize the task graph more easily.

5.15 Simulated Performance

StarPU can use Simgrid in order to simulate execution on an arbitrary platform.

5.15.1 your application for simulation.

There are a few technical details which need to be handled for an application to be simulated through Simgrid.

If the application uses `gettimeofday` to make its performance measurements, the real time will be used, which will be bogus. To get the simulated time, it has to use starpu_timing_now() which returns the virtual timestamp in us.

For some technical reason, the application's .c file which contains main() has to be recompiled with starpu.h, which in the simgrid case will #define main() into starpu_main(), and it is libstarpu which will provide the real main() and call the application's main().

To be able to test with crazy data sizes, one may want to only allocate application data if STARPU_SIMGRID is not defined. Passing a NULL pointer to starpu_data_register functions is fine, data will never be read/written to by StarPU in Simgrid mode anyway.

To be able to run the application with e.g. CUDA simulation on a system which does not have CUDA installed, one can fill the cuda_funcs with (void*)1, to express that there is a CUDA implementation, even if one does not actually provide it. StarPU will never actually run it in Simgrid mode anyway.

5.15.2 Calibration

The idea is to first compile StarPU normally, and run the application, so as to automatically benchmark the bus and the codelets.

```
$ ./configure && make
$ STARPU_SCHED=dmda ./examples/matvecmult/matvecmult
[starpu][_starpu_load_history_based_model] Warning: model matvecmult
   is not calibrated, forcing calibration for this run. Use the
   STARPU_CALIBRATE environment variable to control this.
$ ...
$ STARPU_SCHED=dmda ./examples/matvecmult/matvecmult
TEST PASSED
```

Note that we force to use the scheduler `dmda` to generate performance models for the application. The application may need to be run several times before the model is calibrated.

5.15.3 Simulation

Then, recompile StarPU, passing --enable-simgrid to `./configure`.

```
$ ./configure --enable-simgrid
```

To specify the location of SimGrid, you can either set the environment variables SIMGRID_CFLAGS and SIMGRID_LIBS, or use the configure options --with-simgrid-dir, --with-simgrid-include-dir and --with-simgrid-lib-dir, for example

```
$ ./configure --with-simgrid-dir=/opt/local/simgrid
```

You can then re-run the application.

```
$ make
$ STARPU_SCHED=dmda ./examples/matvecmult/matvecmult
TEST FAILED !!!
```

It is normal that the test fails: since the computation are not actually done (that is the whole point of simgrid), the result is wrong, of course.

If the performance model is not calibrated enough, the following error message will be displayed

```
$ STARPU_SCHED=dmda ./examples/matvecmult/matvecmult
[starpu][_starpu_load_history_based_model] Warning: model matvecmult
    is not calibrated, forcing calibration for this run. Use the
    STARPU_CALIBRATE environment variable to control this.
[starpu][_starpu_simgrid_execute_job][assert failure] Codelet
    matvecmult does not have a perfmodel, or is not calibrated enough
```

The number of devices can be chosen as usual with STARPU_NCPU, STARPU_NCUDA, and STARPU_NOPENCL, and the amount of GPU memory with STARPU_LIMIT_CUDA_MEM, STARPU_LIMIT_CUDA_devid_MEM, STARPU_LIMIT_OPENCL_MEM, and STARPU_LIMIT_OPENCL_devid_MEM.

5.15.4 Simulation On Another Machine

The simgrid support even permits to perform simulations on another machine, your desktop, typically. To achieve this, one still needs to perform the Calibration step on the actual machine to be simulated, then copy them to your desktop machine (the `$STARPU_HOME/.starpu` directory). One can then perform the Simulation step on the desktop machine, by setting the environment variable STARPU_HOSTNAME to the name of the actual machine, to make StarPU use the performance models of the simulated machine even on the desktop machine.

If the desktop machine does not have CUDA or OpenCL, StarPU is still able to use simgrid to simulate execution with CUDA/OpenCL devices, but the application source code will probably disable the CUDA and OpenCL codelets in thatcd sc case. Since during simgrid execution, the functions of the codelet are actually not called, one can use dummy functions such as the following to still permit CUDA or OpenCL execution:

```
static struct starpu_codelet cl11 =
{
        .cpu_funcs = {chol_cpu_codelet_update_u11, NULL},
#ifdef STARPU_USE_CUDA
        .cuda_funcs = {chol_cublas_codelet_update_u11, NULL},
#elif defined(STARPU_SIMGRID)
        .cuda_funcs = {(void*)1, NULL},
#endif
        .nbuffers = 1,
        .modes = {STARPU_RW},
        .model = &chol_model_11
};
```

Chapter 6

Performance Feedback

6.1 Using The Temanejo Task Debugger

StarPU can connect to Temanejo >= 1.0rc2 (see http://www.hlrs.de/temanejo), to permit nice visual task debugging. To do so, build Temanejo's `libayudame.so`, install `Ayudame.h` to e.g. `/usr/local/include`, apply the `tools/patch-ayudame` to it to fix C build, re-`./configure`, make sure that it found it, rebuild StarPU. Run the Temanejo GUI, give it the path to your application, any options you want to pass it, the path to `libayudame.so`.

Make sure to specify at least the same number of CPUs in the dialog box as your machine has, otherwise an error will happen during execution. Future versions of Temanejo should be able to tell StarPU the number of CPUs to use.

Tag numbers have to be below `4000000000000000000ULL` to be usable for Temanejo (so as to distinguish them from tasks).

6.2 On-line Performance Feedback

6.2.1 Enabling On-line Performance Monitoring

In order to enable online performance monitoring, the application can call starpu_profiling_status_set() with the parameter STARPU_PROFILING_ENABLE. It is possible to detect whether monitoring is already enabled or not by calling starpu_profiling_status_get(). Enabling monitoring also reinitialize all previously collected feedback. The environment variable STARPU_PROFILING can also be set to `1` to achieve the same effect. The function starpu_profiling_init() can also be called during the execution to reinitialize performance counters and to start the profiling if the environment variable STARPU_PROFILING is set to `1`.

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

More details about the performance monitoring API are available in Profiling.

6.2.2 Per-task Feedback

If profiling is enabled, a pointer to a structure starpu_profiling_task_info is put in the field starpu_task::profiling_info when a task terminates. This structure is automatically destroyed when the task structure is destroyed, either automatically or by calling starpu_task_destroy().

The structure starpu_profiling_task_info indicates the date when the task was submitted (starpu_profiling_task_info::submit_time), started (starpu_profiling_task_info::start_time), and terminated (starpu_profiling_task_info::end_time), relative to the initialization of StarPU with starpu_init(). It also specifies the identifier of the worker

that has executed the task (starpu_profiling_task_info::workerid). These date are stored as `timespec` structures which the user may convert into micro-seconds using the helper function starpu_timing_timespec_to_us().

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

6.2.3 Per-codelet Feedback

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

6.2.4 Per-worker Feedback

The second argument returned by the function starpu_profiling_worker_get_info() is a structure starpu_profiling_worker_info that gives statistics about the specified worker. This structure specifies when StarPU started collecting profiling information for that worker (starpu_profiling_worker_info::start_time), the duration of the profiling measurement interval (starpu_profiling_worker_info::total_time), the time spent executing kernels (starpu_profiling_worker_info::executing_time), the time spent sleeping because there is no task to execute at all (starpu_profiling_worker_info::sleeping_time), and the number of tasks that were executed while profiling was enabled. These values give an estimation of the proportion of time spent do real work, and the time spent either sleeping because there are not enough executable tasks or simply wasted in pure StarPU overhead.

Calling starpu_profiling_worker_get_info() resets the profiling information associated to a worker.

When an FxT trace is generated (see Generating Traces With FxT), it is also possible to use the tool `starpu_workers_activity` (see Monitoring Activity) to generate a graphic showing the evolution of these values during the time, for the different workers.

6.2.5 Bus-related Feedback

TODO: ajouter STARPU_BUS_STATS

Chapter 7

Tips and Tricks To Know About

7.1 How To Initialize A Computation Library Once For Each Worker?

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

Some global array stores the instanciated objects:

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

At initialisation time of libstarpu, the objects are initialized:

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

And in the codelet body, they are used:

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

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

This however is not sufficient for FFT on CUDA: initialization has to be done from the workers themselves. This can be done thanks to starpu_execute_on_each_worker(). For instance libstarpufft does the following.

```
static void fft_plan_gpu(void *args)
{
    plan plan = args;
    int n2 = plan->n2[0];
    int workerid = starpu_worker_get_id();

    cufftPlan1d(&plan->plans[workerid].plan_cuda, n, _CUFFT_C2C, 1);
    cufftSetStream(plan->plans[workerid].plan_cuda, starpu_cuda_get_local_stream
      ());
}
void starpufft_plan(void)
{
    starpu_execute_on_each_worker(fft_plan_gpu, plan,
      STARPU_CUDA);
}
```

7.2 How to limit memory per node

TODO

Talk about STARPU_LIMIT_CUDA_devid_MEM, STARPU_LIMIT_CUDA_MEM, STARPU_LIMIT_OPENCL_devid_MEM, STARPU_LIMIT_OPENCL_MEM and STARPU_LIMIT_CPU_MEM

starpu_memory_get_available()

7.3 Thread Binding on NetBSD

When using StarPU on a NetBSD machine, if the topology discovery library `hwloc` is used, thread binding will fail. To prevent the problem, you should at least use the version 1.7 of `hwloc`, and also issue the following call:

```
$ sysctl -w security.models.extensions.user_set_cpu_affinity=1
```

Or add the following line in the file `/etc/sysctl.conf`

```
security.models.extensions.user_set_cpu_affinity=1
```

7.4 Using StarPU With MKL 11 (Intel Composer XE 2013)

Some users had issues with MKL 11 and StarPU (versions 1.1rc1 and 1.0.5) on Linux with MKL, using 1 thread for MKL and doing all the parallelism using StarPU (no multithreaded tasks), setting the environment variable MKL_NUM_THREADS to 1, and using the threaded MKL library, with iomp5.

Using this configuration, StarPU uses only 1 core, no matter the value of STARPU_NCPU. The problem is actually a thread pinning issue with MKL.

The solution is to set the environment variable KMP_AFFINITY to `disabled` (http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_openmp_thread_affinity.htm).

7.5 Interleaving StarPU and non-StarPU code

If your application only partially uses StarPU, and you do not want to call starpu_init() / starpu_shutdown() at the beginning/end of each section, StarPU workers will poll for work between the sections. To avoid this behavior, you can "pause" StarPU with the starpu_pause() function. This will prevent the StarPU workers from accepting new work (tasks that are already in progress will not be frozen), and stop them from polling for more work.

Note that this does not prevent you from submitting new tasks, but they won't execute until starpu_resume() is called. Also note that StarPU must not be paused when you call starpu_shutdown(), and that this function pair works in a push/pull manner, ie you need to match the number of calls to these functions to clear their effect.

One way to use these functions could be:

```
starpu_init(NULL);
starpu_pause(); // To submit all the tasks without a single one executing
submit_some_tasks();
starpu_resume(); // The tasks start executing


starpu_task_wait_for_all();
starpu_pause(); // Stop the workers from polling

// Non-StarPU code

starpu_resume();
// ...
starpu_shutdown();
```

Chapter 8

MPI Support

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

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

8.1 Simple Example

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

```
$ pkg-config --cflags starpumpi-1.1  # options for the compiler
$ pkg-config --libs starpumpi-1.1    # options for the linker
```

You also need pass the option `-static` if the application is to be linked statically.

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

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

    starpu_task_submit(task);
}

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

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

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

    unsigned nloops = NITER;
    unsigned loop;

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

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

        if (loop == 0 && rank == 0)
        {
            token = 0;
            fprintf(stdout, "Start with token value %d\n", token);
        }
```

```
        else
        {
            starpu_mpi_irecv_detached(token_handle, (rank+size-1)%size, tag,
                    MPI_COMM_WORLD, NULL, NULL);
        }

        increment_token();

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

    starpu_task_wait_for_all();

    starpu_mpi_shutdown();
    starpu_shutdown();

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

8.2 Point To Point Communication

The standard point to point communications of MPI have been implemented. The semantic is similar to the MPI one, but adapted to the DSM provided by StarPU. A MPI request will only be submitted when the data is available in the main memory of the node submitting the request.

There is two types of asynchronous communications: the classic asynchronous communications and the detached communications. The classic asynchronous communications (starpu_mpi_isend() and starpu_mpi_irecv()) need to be followed by a call to starpu_mpi_wait() or to starpu_mpi_test() to wait for or to test the completion of the communication. Waiting for or testing the completion of detached communications is not possible, this is done internally by StarPU-MPI, on completion, the resources are automatically released. This mechanism is similar to the pthread detach state attribute which determines whether a thread will be created in a joinable or a detached state.

For any communication, the call of the function will result in the creation of a StarPU-MPI request, the function starpu_data_acquire_cb() is then called to asynchronously request StarPU to fetch the data in main memory; when the data is available in main memory, a StarPU-MPI function is called to put the new request in the list of the ready requests.

The StarPU-MPI progression thread regularly polls this list of ready requests. For each new ready request, the appropriate function is called to post the corresponding MPI call. For example, calling starpu_mpi_isend() will result in posting `MPI_Isend`. If the request is marked as detached, the request will be put in the list of detached requests.

The StarPU-MPI progression thread also polls the list of detached requests. For each detached request, it regularly tests the completion of the MPI request by calling `MPI_Test`. On completion, the data handle is released, and if a callback was defined, it is called.

Communication gives the list of all the point to point communications defined in StarPU-MPI.

8.3 Exchanging User Defined Data Interface

New data interfaces defined as explained in Defining A New Data Interface can also be used within StarPU-MPI and exchanged between nodes. Two functions needs to be defined through the type starpu_data_interface_ops. The function starpu_data_interface_ops::pack_data takes a handle and returns a contiguous memory buffer along with its size where data to be conveyed to another node should be copied. The reversed operation is implemented in the

function starpu_data_interface_ops::unpack_data which takes a contiguous memory buffer and recreates the data handle.

```
static int complex_pack_data(starpu_data_handle_t handle, unsigned node, void **ptr,
      ssize_t *count)
{
  STARPU_ASSERT(starpu_data_test_if_allocated_on_node(handle, node));

  struct starpu_complex_interface *complex_interface =
    (struct starpu_complex_interface *) starpu_data_get_interface_on_node(
      handle, node);

  *count = complex_get_size(handle);
  *ptr = malloc(*count);
  memcpy(*ptr, complex_interface->real, complex_interface->nx*sizeof(double));
  memcpy(*ptr+complex_interface->nx*sizeof(double), complex_interface->imaginary,
         complex_interface->nx*sizeof(double));

  return 0;
}

static int complex_unpack_data(starpu_data_handle_t handle, unsigned node, void *ptr,
      size_t count)
{
  STARPU_ASSERT(starpu_data_test_if_allocated_on_node(handle, node));

  struct starpu_complex_interface *complex_interface =
    (struct starpu_complex_interface *) starpu_data_get_interface_on_node(
      handle, node);

  memcpy(complex_interface->real, ptr, complex_interface->nx*sizeof(double));
  memcpy(complex_interface->imaginary, ptr+complex_interface->nx*sizeof(double),
         complex_interface->nx*sizeof(double));

  return 0;
}

static struct starpu_data_interface_ops interface_complex_ops =
{
  ...
  .pack_data = complex_pack_data,
  .unpack_data = complex_unpack_data
};
```

8.4 MPI Insert Task Utility

To save the programmer from having to explicit all communications, StarPU provides an "MPI Insert Task Utility". The principe is that the application decides a distribution of the data over the MPI nodes by allocating it and notifying StarPU of that decision, i.e. tell StarPU which MPI node "owns" which data. It also decides, for each handle, an MPI tag which will be used to exchange the content of the handle. All MPI nodes then process the whole task graph, and StarPU automatically determines which node actually execute which task, and trigger the required MPI transfers.

The list of functions is described in MPI Insert Task.

Here an stencil example showing how to use starpu_mpi_insert_task(). One first needs to define a distribution function which specifies the locality of the data. Note that the data needs to be registered to MPI by calling starpu_mpi_data_register(). This function allows to set the distribution information and the MPI tag which should be used when communicating the data. The function starpu_mpi_data_register() should be prefered to starpu_data_set_rank() and starpu_data_set_tag() as it also allows to automatically clear the MPI communication cache when unregistering the data.

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

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

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

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

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

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

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

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

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

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

This can be a concern with a growing number of nodes. To avoid this, the application can prune the task for loops according to the data distribution, so as to only submit tasks on nodes which have to care about them (either to execute them, or to send the required data).

A way to do some of this quite easily can be to just add an `if` like this:

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

This permits to drop the cost of function call argument passing and parsing.

If the `my_distrib` function can be inlined by the compiler, the latter can improve the test.

If the `size` can be made a compile-time constant, the compiler can considerably improve the test further.

If the distribution function is not too complex and the compiler is very good, the latter can even optimize the `for` loops, thus dramatically reducing the cost of task submission.

8.5 MPI cache support

StarPU-MPI automatically optimizes duplicate data transmissions: if an MPI node B needs a piece of data D from MPI node A for several tasks, only one transmission of D will take place from A to B, and the value of D will be kept on B as long as no task modifies D.

If a task modifies D, B will wait for all tasks which need the previous value of D, before invalidating the value of D. As a consequence, it releases the memory occupied by D. Whenever a task running on B needs the new value of D, allocation will take place again to receive it.

Since tasks can be submitted dynamically, StarPU-MPI can not know whether the current value of data D will again be used by a newly-submitted task before being modified by another newly-submitted task, so until a task is submitted to modify the current value, it can not decide by itself whether to flush the cache or not. The application can however explicitly tell StarPU-MPI to flush the cache by calling starpu_mpi_cache_flush() or starpu_mpi_↩ cache_flush_all_data(), for instance in case the data will not be used at all any more (see for instance the cholesky example in mpi/examples/matrix_decomposition), or at least not in the close future. If a newly-submitted task actually needs the value again, another transmission of D will be initiated from A to B.

The whole caching behavior can be disabled thanks to the ::STARPU_MPI_CACHE environment variable. The variable ::STARPU_MPI_CACHE_STATS can be set to 1 to enable the runtime to display messages when data are added or removed from the cache holding the received data.

8.6 MPI Data migration

The application can dynamically change its mind about the data distribution, to balance the load over MPI nodes for instance. This can be done very simply by requesting an explicit move and then change the registered rank. For instance, we here switch to a new distribution function `my_distrib2`: we first register any data that wasn't registered already and will be needed, then migrate the data, and register the new location.

```
for(x = 0; x < X; x++) {
    for (y = 0; y < Y; y++) {
        int mpi_rank = my_distrib2(x, y, size);
        if (!data_handles[x][y] && (mpi_rank == my_rank
              || my_rank == my_distrib(x+1, y, size) || my_rank == my_distrib(x-1, y, size)
              || my_rank == my_distrib(x, y+1, size) || my_rank == my_distrib(x, y-1, size)))
            /* Register newly-needed data */
            starpu_variable_data_register(&data_handles[x][y], -1,
                                          (uintptr_t)NULL, sizeof(unsigned));
        if (data_handles[x][y]) {
            /* Migrate the data */
            starpu_mpi_get_data_on_node_detached(MPI_COMM_WORLD,
      data_handles[x][y], mpi_rank, NULL, NULL);
            /* And register the new rank of the matrix */
            starpu_data_set_rank(data_handles[x][y], mpi_rank);
        }
    }
}
```

From then on, further tasks submissions will use the new data distribution, which will thus change both MP↩ I communications and task assignments.

Very importantly, since all nodes have to agree on which node owns which data so as to determine MPI communications and task assignments the same way, all nodes have to perform the same data migration, and at the same point among task submissions. It thus does not require a strict synchronization, just a clear separation of task submissions before and after the data redistribution.

Before data unregistration, it has to be migrated back to its original home node (the value, at least), since that is where the user-provided buffer resides. Otherwise the unregistration will complain that it does not have the latest value on the original home node.

```
for(x = 0; x < X; x++) {
```

```
    for (y = 0; y < Y; y++) {
        if (data_handles[x][y]) {
            int mpi_rank = my_distrib(x, y, size);
            /* Get back data to original place where the user-provided buffer is.  */
            starpu_mpi_get_data_on_node_detached(MPI_COMM_WORLD,
      data_handles[x][y], mpi_rank, NULL, NULL);
            /* And unregister it */
            starpu_data_unregister(data_handles[x][y]);
        }
    }
}
```

8.7 MPI Collective Operations

The functions are described in MPI Collective Operations.

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

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

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

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

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

Chapter 9

FFT Support

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

Different precisions are available, namely float, double and long double precisions, with the following fftw naming conventions:

- double precision structures and functions are named e.g. starpufft_execute()
- float precision structures and functions are named e.g. starpufftf_execute()
- long double precision structures and functions are named e.g. starpufftl_execute()

The documentation below is given with names for double precision, replace `starpufft_` with `starpufftf_` or `starpufftl_` as appropriate.

Only complex numbers are supported at the moment.

The application has to call starpu_init() before calling `starpufft` functions.

Either main memory pointers or data handles can be provided.

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

All functions are defined in FFT Support.

9.1 Compilation

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

```
$ pkg-config --cflags starpufft-1.1  # options for the compiler
$ pkg-config --libs starpufft-1.1    # options for the linker
```

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

Chapter 10

C Extensions

When GCC plug-in support is available, StarPU builds a plug-in for the GNU Compiler Collection (GCC), which defines extensions to languages of the C family (C, C++, Objective-C) that make it easier to write StarPU code. This feature is only available for GCC 4.5 and later; it is known to work with GCC 4.5, 4.6, and 4.7. You may need to install a specific `-dev` package of your distro, such as `gcc-4.6-plugin-dev` on Debian and derivatives. In addition, the plug-in's test suite is only run when `GNU Guile` is found at `configure`-time. Building the GCC plug-in can be disabled by configuring with --disable-gcc-extensions.

Those extensions include syntactic sugar for defining tasks and their implementations, invoking a task, and manipulating data buffers. Use of these extensions can be made conditional on the availability of the plug-in, leading to valid C sequential code when the plug-in is not used (Using C Extensions Conditionally).

When StarPU has been installed with its GCC plug-in, programs that use these extensions can be compiled this way:

```
$ gcc -c -fplugin=`pkg-config starpu-1.1 --variable=gccplugin` foo.c
```

When the plug-in is not available, the above `pkg-config` command returns the empty string.

In addition, the `-fplugin-arg-starpu-verbose` flag can be used to obtain feedback from the compiler as it analyzes the C extensions used in source files.

This section describes the C extensions implemented by StarPU's GCC plug-in. It does not require detailed knowledge of the StarPU library.

Note: this is still an area under development and subject to change.

10.1 Defining Tasks

The StarPU GCC plug-in views tasks as "extended" C functions:

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

Tasks and their implementations must be *declared*. These declarations are annotated with attributes (`http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html#Attribute-Syntax`): the declaration of a task is a regular C function declaration with an additional `task` attribute, and task implementations are declared with a `task_implementation` attribute.

The following function attributes are provided:

task Declare the given function as a StarPU task. Its return type must be `void`. When a function declared as `task` has a user-defined body, that body is interpreted as the implicit definition of the task's CPU implementation (see example below). In all cases, the actual definition of a task's body is automatically generated by the compiler.

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

Scalar arguments to the task are passed by value and copied to the target device if need be—technically, they are passed as the buffer starpu_task::cl_arg (Codelet and Tasks).

Pointer arguments are assumed to be registered data buffers—the handles argument of a task (starpu_task::handles) ; `const`-qualified pointer arguments are viewed as read-only buffers (STARPU_R), and non-`const`-qualified buffers are assumed to be used read-write (STARPU_RW). In addition, the `output` type attribute can be as a type qualifier for output pointer or array parameters (STARPU_W).

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

Chapter 11

SOCL OpenCL Extensions

SOCL is an OpenCL implementation based on StarPU. It gives a unified access to every available OpenCL device↩ : applications can now share entities such as Events, Contexts or Command Queues between several OpenCL implementations.

In addition, command queues that are created without specifying a device provide automatic scheduling of the submitted commands on OpenCL devices contained in the context to which the command queue is attached.

Note: this is still an area under development and subject to change.

When compiling StarPU, SOCL will be enabled if a valid OpenCL implementation is found on your system. To be able to run the SOCL test suite, the environment variable SOCL_OCL_LIB_OPENCL needs to be defined to the location of the file `libOpenCL.so` of the OCL ICD implementation. You should for example add the following line in your file `.bashrc`

```
export SOCL_OCL_LIB_OPENCL=/usr/lib/x86_64-linux-gnu/libOpenCL.so
```

You can then run the test suite in the directory `socl/examples`.

```
$ make check
...
PASS: basic/basic
PASS: testmap/testmap
PASS: clinfo/clinfo
PASS: matmul/matmul
PASS: mansched/mansched
==================
All 5 tests passed
==================
```

The environment variable OCL_ICD_VENDORS has to point to the directory where the ICD files are installed. When compiling StarPU, the files are in the directory `socl/vendors`. With an installed version of StarPU, the files are installed in the directory `$prefix/share/starpu/opencl/vendors`.

To run the tests by hand, you have to call for example,

```
$ LD_PRELOAD=$SOCL_OCL_LIB_OPENCL OCL_ICD_VENDORS=socl/vendors/ socl/examples/clinfo/clinfo
Number of platforms: 2
  Plaform Profile: FULL_PROFILE
  Plaform Version: OpenCL 1.1 CUDA 4.2.1
  Plaform Name: NVIDIA CUDA
  Plaform Vendor: NVIDIA Corporation
  Plaform Extensions: cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options cl_nv_

  Plaform Profile: FULL_PROFILE
  Plaform Version: OpenCL 1.0 SOCL Edition (0.1.0)
  Plaform Name: SOCL Platform
  Plaform Vendor: INRIA
  Plaform Extensions: cl_khr_icd
....
$
```

To enable the use of CPU cores via OpenCL, one can set the STARPU_OPENCL_ON_CPUS environment variable to 1 and STARPU_NCPUS to 0 (to avoid using CPUs both via the OpenCL driver and the normal CPU driver).

Chapter 12

Scheduling Contexts

TODO: improve!

12.1 General Ideas

Scheduling contexts represent abstracts sets of workers that allow the programmers to control the distribution of computational resources (i.e. CPUs and GPUs) to concurrent parallel kernels. The main goal is to minimize interferences between the execution of multiple parallel kernels, by partitioning the underlying pool of workers using contexts.

12.2 Creating A Context

By default, the application submits tasks to an initial context, which disposes of all the computation resources available to StarPU (all the workers). If the application programmer plans to launch several parallel kernels simultaneously, by default these kernels will be executed within this initial context, using a single scheduler policy(see Task Scheduling Policy). Meanwhile, if the application programmer is aware of the demands of these kernels and of the specificity of the machine used to execute them, the workers can be divided between several contexts. These scheduling contexts will isolate the execution of each kernel and they will permit the use of a scheduling policy proper to each one of them.

Scheduling Contexts may be created in two ways: either the programmers indicates the set of workers corresponding to each context (providing he knows the identifiers of the workers running within StarPU), or the programmer does not provide any worker list and leaves the Hypervisor assign workers to each context according to their needs (Scheduling Context Hypervisor)

Both cases require a call to the function `starpu_sched_ctx_create`, which requires as input the worker list (the exact list or a NULL pointer) and the scheduling policy. The latter one can be a character list corresponding to the name of a StarPU predefined policy or the pointer to a custom policy. The function returns an identifier of the context created which you will use to indicate the context you want to submit the tasks to.

```
/* the list of resources the context will manage */
int workerids[3] = {1, 3, 10};

/* indicate the scheduling policy to be used within the context, the list of
   workers assigned to it, the number of workers, the name of the context */
int id_ctx = starpu_sched_ctx_create("dmda", workerids, 3, "my_ctx");

/* let StarPU know that the following tasks will be submitted to this context */
starpu_sched_ctx_set_task_context(id);

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

Note: Parallel greedy and parallel heft scheduling policies do not support the existence of several disjoint contexts

on the machine. Combined workers are constructed depending on the entire topology of the machine, not only the one belonging to a context.

12.3 Modifying A Context

A scheduling context can be modified dynamically. The applications may change its requirements during the execution and the programmer can add additional workers to a context or remove if no longer needed. In the following example we have two scheduling contexts `sched_ctx1` and `sched_ctx2`. After executing a part of the tasks some of the workers of `sched_ctx1` will be moved to context `sched_ctx2`.

```
/* the list of ressources that context 1 will give away */
int workerids[3] = {1, 3, 10};

/* add the workers to context 1 */
starpu_sched_ctx_add_workers(workerids, 3, sched_ctx2);

/* remove the workers from context 2 */
starpu_sched_ctx_remove_workers(workerids, 3, sched_ctx1);
```

12.4 Submitting Tasks To A Context

The application may submit tasks to several contexts either simultaneously or sequnetially. If several threads of submission are used the function `starpu_sched_ctx_set_context` may be called just before `starpu_task_submit`. Thus StarPU considers that the current thread will submit tasks to the coresponding context.

When the application may not assign a thread of submission to each context, the id of the context must be indicated by using the function `starpu_task_submit_to_ctx` or the field `STARPU_SCHED_CTX` for `starpu_insert_task`.

12.5 Deleting A Context

When a context is no longer needed it must be deleted. The application can indicate which context should keep the resources of a deleted one. All the tasks of the context should be executed before doing this. Thus, the programmer may use either a barrier and then delete the context directly, or just indicate that other tasks will not be submitted later on to the context (such that when the last task is executed its workers will be moved to the inheritor) and delete the context at the end of the execution (when a barrier will be used eventually).

```
/* when the context 2 is deleted context 1 inherits its resources */
starpu_sched_ctx_set_inheritor(sched_ctx2, sched_ctx1);

/* submit tasks to context 2 */
for (i = 0; i < ntasks; i++)
    starpu_task_submit_to_ctx(task[i],sched_ctx2);

/* indicate that context 2 finished submitting and that */
/* as soon as the last task of context 2 finished executing */
/* its workers can be moved to the inheritor context */
starpu_sched_ctx_finished_submit(sched_ctx1);

/* wait for the tasks of both contexts to finish */
starpu_task_wait_for_all();

/* delete context 2 */
starpu_sched_ctx_delete(sched_ctx2);

/* delete context 1 */
starpu_sched_ctx_delete(sched_ctx1);
```

12.6 Emptying A Context

A context may have no resources at the begining or at a certain moment of the execution. Task can still be submitted to these contexts and they will be executed as soon as the contexts will have resources. A list of tasks pending

to be executed is kept and when workers are added to the contexts these tasks start being submitted. However, if resources are never allocated to the context the program will not terminate. If these tasks have low priority the programmer can forbid the application to submit them by calling the function `starpu_sched_ctx_stop_task_submission()`.

12.7 Contexts Sharing Workers

Contexts may share workers when a single context cannot execute efficiently enough alone on these workers or when the application decides to express a hierarchy of contexts. The workers apply an alogrithm of “Round-Robin” to chose the context on which they will “pop” next. By using the function `starpu_sched_ctx_set_turn_to_other_ctx`, the programmer can impose the `workerid` to “pop” in the context `sched_ctx_id` next.

Chapter 13

Scheduling Context Hypervisor

13.1 What Is The Hypervisor

StarPU proposes a platform to construct Scheduling Contexts, to delete and modify them dynamically. A parallel kernel, can thus be isolated into a scheduling context and interferences between several parallel kernels are avoided. If the user knows exactly how many workers each scheduling context needs, he can assign them to the contexts at their creation time or modify them during the execution of the program.

The Scheduling Context Hypervisor Plugin is available for the users who do not dispose of a regular parallelism, who cannot know in advance the exact size of the context and need to resize the contexts according to the behavior of the parallel kernels.

The Hypervisor receives information from StarPU concerning the execution of the tasks, the efficiency of the resources, etc. and it decides accordingly when and how the contexts can be resized. Basic strategies of resizing scheduling contexts already exist but a platform for implementing additional custom ones is available.

13.2 Start the Hypervisor

The Hypervisor must be initialized once at the beginning of the application. At this point a resizing policy should be indicated. This strategy depends on the information the application is able to provide to the hypervisor as well as on the accuracy needed for the resizing procedure. For example, the application may be able to provide an estimation of the workload of the contexts. In this situation the hypervisor may decide what resources the contexts need. However, if no information is provided the hypervisor evaluates the behavior of the resources and of the application and makes a guess about the future. The hypervisor resizes only the registered contexts.

13.3 Interrogate The Runtime

The runtime provides the hypervisor with information concerning the behavior of the resources and the application. This is done by using the `performance_counters` which represent callbacks indicating when the resources are idle or not efficient, when the application submits tasks or when it becomes to slow.

13.4 Trigger the Hypervisor

The resizing is triggered either when the application requires it (`sc_hypervisor_resize_ctxs`) or when the initials distribution of resources alters the performance of the application (the application is to slow or the resource are idle for too long time). If the environment variable `SC_HYPERVISOR_TRIGGER_RESIZE` is set to `speed` the monitored speed of the contexts is compared to a theoretical value computed with a linear program, and the resizing is triggered whenever the two values do not correspond. Otherwise, if the environment variable is set to `idle` the

hypervisor triggers the resizing algorithm whenever the workers are idle for a period longer than the threshold indicated by the programmer. When this happens different resizing strategy are applied that target minimizing the total execution of the application, the instant speed or the idle time of the resources.

13.5 Resizing Strategies

The plugin proposes several strategies for resizing the scheduling context.

The **Application driven** strategy uses the user's input concerning the moment when he wants to resize the contexts. Thus, the users tags the task that should trigger the resizing process. We can set directly the field starpu_task::hypervisor_tag or use the macro ::STARPU_HYPERVISOR_TAG in the function starpu_insert_task().

```
task.hypervisor_tag = 2;
```

or

```
starpu_insert_task(&codelet,
                   ...,
                   STARPU_HYPERVISOR_TAG, 2,
                   0);
```

Then the user has to indicate that when a task with the specified tag is executed the contexts should resize.

```
sc_hypervisor_resize(sched_ctx, 2);
```

The user can use the same tag to change the resizing configuration of the contexts if he considers it necessary.

```
sc_hypervisor_ctl(sched_ctx,
                  SC_HYPERVISOR_MIN_WORKERS, 6,
                  SC_HYPERVISOR_MAX_WORKERS, 12,
                  SC_HYPERVISOR_TIME_TO_APPLY, 2,
                  NULL);
```

The **Idleness** based strategy moves workers unused in a certain context to another one needing them. (see Users' Input In The Resizing Process)

```
int workerids[3] = {1, 3, 10};
int workerids2[9] = {0, 2, 4, 5, 6, 7, 8, 9, 11};
sc_hypervisor_ctl(sched_ctx_id,
            SC_HYPERVISOR_MAX_IDLE, workerids, 3, 10000.0,
            SC_HYPERVISOR_MAX_IDLE, workerids2, 9, 50000.0,
            NULL);
```

The **Gflops rate** based strategy resizes the scheduling contexts such that they all finish at the same time. The speed of each of them is computed and once one of them is significantly slower the resizing process is triggered. In order to do these computations the user has to input the total number of instructions needed to be executed by the parallel kernels and the number of instruction to be executed by each task.

The number of flops to be executed by a context are passed as parameter when they are registered to the hypervisor, (`sc_hypervisor_register_ctx(sched_ctx_id, flops)`) and the one to be executed by each task are passed when the task is submitted. The corresponding field is starpu_task::flops and the corresponding macro in the function starpu_insert_task() is STARPU_FLOPS (**Caution**: but take care of passing a double, not an integer, otherwise parameter passing will be bogus). When the task is executed the resizing process is triggered.

```
task.flops = 100;
```

or

```
starpu_insert_task(&codelet,
                   ...,
                   STARPU_FLOPS, (double) 100,
                   0);
```

The **Feft** strategy uses a linear program to predict the best distribution of resources such that the application finishes in a minimum amount of time. As for the **Gflops rate** strategy the programmers has to indicate the total number of flops to be executed when registering the context. This number of flops may be updated dynamically during the execution of the application whenever this information is not very accurate from the beginning. The function `sc_hypervisor_update_diff_total_flop` is called in order add or remove a difference to the flops left to be executed. Tasks are provided also the number of flops corresponding to each one of them. During the execution of the application the hypervisor monitors the consumed flops and recomputes the time left and the number of resources to use. The speed of each type of resource is (re)evaluated and inserter in the linear program in order to better adapt to the needs of the application.

The **Teft** strategy uses a linear program too, that considers all the types of tasks and the number of each of them and it tries to allocates resources such that the application finishes in a minimum amount of time. A previous calibration of StarPU would be useful in order to have good predictions of the execution time of each type of task.

The types of tasks may be determines directly by the hypervisor when they are submitted. However there are applications that do not expose all the graph of tasks from the beginning. In this case in order to let the hypervisor know about all the tasks the function `sc_hypervisor_set_type_of_task` will just inform the hypervisor about future tasks without submitting them right away.

The **Ispeed** strategy divides the execution of the application in several frames. For each frame the hypervisor computes the speed of the contexts and tries making them run at the same speed. The strategy requires less contribution from the user as the hypervisor requires only the size of the frame in terms of flops.

```
int workerids[3] = {1, 3, 10};
int workerids2[9] = {0, 2, 4, 5, 6, 7, 8, 9, 11};
sc_hypervisor_ctl(sched_ctx_id,
                  SC_HYPERVISOR_ISPEED_W_SAMPLE, workerids, 3, 2000000000.0,
                  SC_HYPERVISOR_ISPEED_W_SAMPLE, workerids2, 9, 200000000000.0
      ,
                  SC_HYPERVISOR_ISPEED_CTX_SAMPLE, 60000000000.0,
            NULL);
```

The **Throughput** strategy focuses on maximizing the throughput of the resources and resizes the contexts such that the machine is running at its maximum efficiency (maximum instant speed of the workers).

13.6 a new hypervisor policy

While Scheduling Context Hypervisor Plugin comes with a variety of resizing policies (see Resizing Strategies), it may sometimes be desirable to implement custom policies to address specific problems. The API described below allows users to write their own resizing policy.

Here an example of how to define a new policy

```
struct sc_hypervisor_policy dummy_policy =
{
       .handle_poped_task = dummy_handle_poped_task,
       .handle_pushed_task = dummy_handle_pushed_task,
       .handle_idle_cycle = dummy_handle_idle_cycle,
       .handle_idle_end = dummy_handle_idle_end,
       .handle_post_exec_hook = dummy_handle_post_exec_hook,
       .custom = 1,
       .name = "dummy"
};
```

Part II

Inside StarPU

Chapter 14

Execution Configuration Through Environment Variables

The behavior of the StarPU library and tools may be tuned thanks to the following environment variables.

14.1 Configuring Workers

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

STARPU_NCPUS This variable is deprecated. You should use STARPU_NCPU.

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

STARPU_NOPENCL OpenCL equivalent of the environment variable STARPU_NCUDA.

STARPU_OPENCL_ON_CPUS By default, the OpenCL driver only enables GPU and accelerator devices. By setting the environment variable STARPU_OPENCL_ON_CPUS to 1, the OpenCL driver will also enable CPU devices.

STARPU_OPENCL_ONLY_ON_CPUS By default, the OpenCL driver enables GPU and accelerator devices. By setting the environment variable STARPU_OPENCL_ONLY_ON_CPUS to 1, the OpenCL driver will ONLY enable CPU devices.

STARPU_WORKERS_NOBIND Setting it to non-zero will prevent StarPU from binding its threads to CPUs. This is for instance useful when running the testsuite in parallel.

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

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

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

This variable is ignored if the field starpu_conf::use_explicit_workers_bindid passed to starpu_init() is set.

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

This variable is ignored if the field starpu_conf::use_explicit_workers_cuda_gpuid passed to starpu_init() is set.

STARPU_WORKERS_OPENCLID OpenCL equivalent of the STARPU_WORKERS_CUDAID environment variable.

This variable is ignored if the field starpu_conf::use_explicit_workers_opencl_gpuid passed to starpu_init() is set.

STARPU_SINGLE_COMBINED_WORKER If set, StarPU will create several workers which won't be able to work concurrently. It will by default create combined workers which size goes from 1 to the total number of CPU workers in the system. STARPU_MIN_WORKERSIZE and STARPU_MAX_WORKERSIZE can be used to change this default.

STARPU_MIN_WORKERSIZE STARPU_MIN_WORKERSIZE permits to specify the minimum size of the combined workers (instead of the default 2)

STARPU_MAX_WORKERSIZE STARPU_MAX_WORKERSIZE permits to specify the minimum size of the combined workers (instead of the number of CPU workers in the system)

STARPU_SYNTHESIZE_ARITY_COMBINED_WORKER Let the user decide how many elements are allowed between combined workers created from hwloc information. For instance, in the case of sockets with 6 cores without shared L2 caches, if STARPU_SYNTHESIZE_ARITY_COMBINED_WORKER is set to 6, no combined worker will be synthesized beyond one for the socket and one per core. If it is set to 3, 3 intermediate combined workers will be synthesized, to divide the socket cores into 3 chunks of 2 cores. If it set to 2, 2 intermediate combined workers will be synthesized, to divide the the socket cores into 2 chunks of 3 cores, and then 3 additional combined workers will be synthesized, to divide the former synthesized workers into a bunch of 2 cores, and the remaining core (for which no combined worker is synthesized since there is already a normal worker for it).

The default, 2, thus makes StarPU tend to building a binary trees of combined workers.

STARPU_DISABLE_ASYNCHRONOUS_COPY Disable asynchronous copies between CPU and GPU devices. The AMD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers.

STARPU_DISABLE_ASYNCHRONOUS_CUDA_COPY Disable asynchronous copies between CPU and CUDA devices.

STARPU_DISABLE_ASYNCHRONOUS_OPENCL_COPY Disable asynchronous copies between CPU and OpenCL devices. The AMD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers.

STARPU_ENABLE_CUDA_GPU_GPU_DIRECT Enable (1) or Disable (0) direct CUDA transfers from GPU to GPU, without copying through RAM. The default is Enabled. This permits to test the performance effect of GPU-Direct.

STARPU_DISABLE_PINNING Disable (1) or Enable (0) pinning host memory allocated through starpu_malloc and friends. The default is Enabled. This permits to test the performance effect of memory pinning.

14.2 Configuring The Scheduling Engine

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

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

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

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

STARPU_CALIBRATE_MINIMUM This defines the minimum number of calibration measurements that will be made before considering that the performance model is calibrated. The default value is 10.

STARPU_BUS_CALIBRATE If this variable is set to 1, the bus is recalibrated during intialization.

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

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

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

STARPU_SCHED_GAMMA Define the execution time penalty of a joule (Power-based Scheduling).

STARPU_IDLE_POWER Define the idle power of the machine (Power-based Scheduling).

STARPU_PROFILING Enable on-line performance monitoring (Enabling On-line Performance Monitoring).

14.3 Extensions

SOCL_OCL_LIB_OPENCL THE SOCL test suite is only run when the environment variable SOCL_OCL_LIB_OPENCL is defined. It should contain the location of the file `libOpenCL.so` of the OCL ICD implementation.

OCL_ICD_VENDORS When using SOCL with OpenCL ICD (`https://forge.imag.fr/projects/ocl-icd/`), this variable may be used to point to the directory where ICD files are installed. The default directory is `/etc/OpenCL/vendors`. StarPU installs ICD files in the directory `$prefix/share/starpu/opencl/vendors`.

STARPU_COMM_STATS Communication statistics for starpumpi (MPI Support) will be enabled when the environment variable STARPU_COMM_STATS is defined to an value other than 0.

STARPU_MPI_CACHE Communication cache for starpumpi (MPI Support) will be disabled when the environment variable STARPU_MPI_CACHE is set to 0. It is enabled by default or for any other values of the variable STARPU_MPI_CACHE.

STARPU_MPI_CACHE_STATS When set to 1, statistics are enabled for the communication cache (MPI Support). For now, it prints messages on the standard output when data are added or removed from the received communication cache.

14.4 Miscellaneous And Debug

STARPU_HOME This specifies the main directory in which StarPU stores its configuration files. The default is `$HOME` on Unix environments, and `$USERPROFILE` on Windows environments.

STARPU_HOSTNAME When set, force the hostname to be used when dealing performance model files. Models are indexed by machine name. When running for example on a homogenenous cluster, it is possible to share the models between machines by setting `export STARPU_HOSTNAME=some_global_name`.

STARPU_OPENCL_PROGRAM_DIR This specifies the directory where the OpenCL codelet source files are located. The function starpu_opencl_load_program_source() looks for the codelet in the current directory, in the directory specified by the environment variable STARPU_OPENCL_PROGRAM_DIR, in the directory `share/starpu/opencl` of the installation directory of StarPU, and finally in the source directory of StarPU.

STARPU_SILENT This variable allows to disable verbose mode at runtime when StarPU has been configured with the option --enable-verbose. It also disables the display of StarPU information and warning messages.

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

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

STARPU_LIMIT_CUDA_devid_MEM This variable specifies the maximum number of megabytes that should be available to the application on the CUDA device with the identifier `devid`. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory. When defined, the variable overwrites the value of the variable STARPU_LIMIT_CUDA_MEM.

STARPU_LIMIT_CUDA_MEM This variable specifies the maximum number of megabytes that should be available to the application on each CUDA devices. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory.

STARPU_LIMIT_OPENCL_devid_MEM This variable specifies the maximum number of megabytes that should be available to the application on the OpenCL device with the identifier `devid`. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory. When defined, the variable overwrites the value of the variable STARPU_LIMIT_OPENCL_MEM.

STARPU_LIMIT_OPENCL_MEM This variable specifies the maximum number of megabytes that should be available to the application on each OpenCL devices. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory.

STARPU_LIMIT_CPU_MEM This variable specifies the maximum number of megabytes that should be available to the application on each CPU device. This variable is intended to be used for experimental purposes as it emulates devices that have a limited amount of memory.

STARPU_TRACE_BUFFER_SIZE This sets the buffer size for recording trace events in MiB. Setting it to a big size allows to avoid pauses in the trace while it is recorded on the disk. This however also consumes memory, of course. The default value is 64.

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

STARPU_MEMORY_STATS When set to 0, disable the display of memory statistics on data which have not been unregistered at the end of the execution (Memory Feedback).

STARPU_BUS_STATS When defined, statistics about data transfers will be displayed when calling starpu_shutdown() (Profiling).

STARPU_WORKER_STATS When defined, statistics about the workers will be displayed when calling starpu_shutdown() (Profiling). When combined with the environment variable STARPU_PROFILING, it displays the power consumption (Power-based Scheduling).

STARPU_STATS When set to 0, data statistics will not be displayed at the end of the execution of an application (Data Statistics).

STARPU_WATCHDOG_TIMEOUT When set to a value other than 0, allows to make StarPU print an error message whenever StarPU does not terminate any task for the given time (in µs). Should be used in combination with STARPU_WATCHDOG_CRASH (see Detection Stuck Conditions).

STARPU_WATCHDOG_CRASH When set to a value other than 0, it triggers a crash when the watch dog is reached, thus allowing to catch the situation in gdb, etc (see Detection Stuck Conditions)

STARPU_DISABLE_KERNELS When set to a value other than 1, it disables actually calling the kernel functions, thus allowing to quickly check that the task scheme is working properly, without performing the actual application-provided computation.

14.5 Configuring The Hypervisor

SC_HYPERVISOR_POLICY Choose between the different resizing policies proposed by StarPU for the hypervisor: idle, app_driven, feft_lp, teft_lp; ispeed_lp, throughput_lp etc.

Use `SC_HYPERVISOR_POLICY=help` to get the list of available policies for the hypervisor

SC_HYPERVISOR_TRIGGER_RESIZE Choose how should the hypervisor be triggered: `speed` if the resizing algorithm should be called whenever the speed of the context does not correspond to an optimal precomputed value, `idle` it the resizing algorithm should be called whenever the workers are idle for a period longer than the value indicated when configuring the hypervisor.

SC_HYPERVISOR_START_RESIZE Indicate the moment when the resizing should be available. The value correspond to the percentage of the total time of execution of the application. The default value is the resizing frame.

SC_HYPERVISOR_MAX_SPEED_GAP Indicate the ratio of speed difference between contexts that should trigger the hypervisor. This situation may occur only when a theoretical speed could not be computed and the hypervisor has no value to compare the speed to. Otherwise the resizing of a context is not influenced by the the speed of the other contexts, but only by the the value that a context should have.

SC_HYPERVISOR_STOP_PRINT By default the values of the speed of the workers is printed during the execution of the application. If the value 1 is given to this environment variable this printing is not done.

SC_HYPERVISOR_LAZY_RESIZE By default the hypervisor resizes the contexts in a lazy way, that is workers are firstly added to a new context before removing them from the previous one. Once this workers are clearly taken into account into the new context (a task was poped there) we remove them from the previous one. However if the application would like that the change in the distribution of workers should change right away this variable should be set to 0

SC_HYPERVISOR_SAMPLE_CRITERIA By default the hypervisor uses a sample of flops when computing the speed of the contexts and of the workers. If this variable is set to `time` the hypervisor uses a sample of time (10% of an aproximation of the total execution time of the application)

Chapter 15

Compilation Configuration

The behavior of the StarPU library and tools may be tuned thanks to the following configure options.

15.1 Common Configuration

–enable-debug Enable debugging messages.

–enable-spinlock-check Enable checking that spinlocks are taken and released properly.

–enable-fast Disable assertion checks, which saves computation time.

–enable-verbose Increase the verbosity of the debugging messages. This can be disabled at runtime by setting the environment variable STARPU_SILENT to any value.

```
$ STARPU_SILENT=1 ./vector_scal
```

–enable-coverage Enable flags for the coverage tool `gcov`.

–enable-quick-check Specify tests and examples should be run on a smaller data set, i.e allowing a faster execution time

–enable-long-check Enable some exhaustive checks which take a really long time.

–with-hwloc Specify `hwloc` should be used by StarPU. `hwloc` should be found by the means of the tool `pkg-config`.

–with-hwloc=`prefix` Specify `hwloc` should be used by StarPU. `hwloc` should be found in the directory specified by `prefix`

–without-hwloc Specify `hwloc` should not be used by StarPU.

–disable-build-doc Disable the creation of the documentation. This should be done on a machine which does not have the tools `doxygen` and `latex` (plus the packages `latex-xcolor` and `texlive-latex-extra`).

Additionally, the script `configure` recognize many variables, which can be listed by typing `./configure -help`. For example, `./configure NVCCFLAGS="-arch sm_13"` adds a flag for the compilation of CUDA kernels.

15.2 Configuring Workers

–enable-maxcpus=`count` Use at most `count` CPU cores. This information is then available as the macro ::STARPU_MAXCPUS.

–disable-cpu Disable the use of CPUs of the machine. Only GPUs etc. will be used.

–enable-maxcudadev=count Use at most `count` CUDA devices. This information is then available as the macro STARPU_MAXCUDADEVS.

–disable-cuda Disable the use of CUDA, even if a valid CUDA installation was detected.

–with-cuda-dir=prefix Search for CUDA under `prefix`, which should notably contain the file `include/cuda.h`.

–with-cuda-include-dir=dir Search for CUDA headers under `dir`, which should notably contain the file `cuda.h`. This defaults to `/include` appended to the value given to --with-cuda-dir.

–with-cuda-lib-dir=dir Search for CUDA libraries under `dir`, which should notably contain the CUDA shared libraries—e.g., `libcuda.so`. This defaults to `/lib` appended to the value given to --with-cuda-dir.

–disable-cuda-memcpy-peer Explicitly disable peer transfers when using CUDA 4.0.

–enable-maxopencldev=count Use at most `count` OpenCL devices. This information is then available as the macro STARPU_MAXOPENCLDEVS.

–disable-opencl Disable the use of OpenCL, even if the SDK is detected.

–with-opencl-dir=prefix Search for an OpenCL implementation under `prefix`, which should notably contain `include/CL/cl.h` (or `include/OpenCL/cl.h` on Mac OS).

–with-opencl-include-dir=dir Search for OpenCL headers under `dir`, which should notably contain `CL/cl.h` (or `OpenCL/cl.h` on Mac OS). This defaults to `/include` appended to the value given to --with-opencl-dir.

–with-opencl-lib-dir=dir Search for an OpenCL library under `dir`, which should notably contain the OpenCL shared libraries—e.g. `libOpenCL.so`. This defaults to `/lib` appended to the value given to --with-opencl-dir.

–enable-opencl-simulator Enable considering the provided OpenCL implementation as a simulator, i.e. use the kernel duration returned by OpenCL profiling information as wallclock time instead of the actual measured real time. This requires simgrid support.

–enable-maximplementations=count Allow for at most `count` codelet implementations for the same target device. This information is then available as the macro ::STARPU_MAXIMPLEMENTATIONS macro.

–enable-max-sched-ctxs=count Allow for at most `count` scheduling contexts This information is then available as the macro ::STARPU_NMAX_SCHED_CTXS.

–disable-asynchronous-copy Disable asynchronous copies between CPU and GPU devices. The AMD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers.

–disable-asynchronous-cuda-copy Disable asynchronous copies between CPU and CUDA devices.

–disable-asynchronous-opencl-copy Disable asynchronous copies between CPU and OpenCL devices. The AMD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers.

15.3 Extension Configuration

–disable-socl Disable the SOCL extension (SOCL OpenCL Extensions). By default, it is enabled when an OpenCL implementation is found.

–disable-starpu-top Disable the StarPU-Top interface (StarPU-Top Interface). By default, it is enabled when the required dependencies are found.

–disable-gcc-extensions Disable the GCC plug-in (C Extensions). By default, it is enabled when the GCC compiler provides a plug-in support.

–with-mpicc=path Use the compiler `mpicc` at `path`, for StarPU-MPI. (MPI Support).

–enable-mpi-progression-hook Enable the activity polling method for StarPU-MPI.

15.4 Advanced Configuration

–enable-perf-debug Enable performance debugging through gprof.

–enable-model-debug Enable performance model debugging.

–enable-stats (see ../../src/datawizard/datastats.c) Enable gathering of various data statistics (Data Statistics).

–enable-maxbuffers Define the maximum number of buffers that tasks will be able to take as parameters, then available as the macro STARPU_NMAXBUFS.

–enable-allocation-cache Enable the use of a data allocation cache to avoid the cost of it with CUDA. Still experimental.

–enable-opengl-render Enable the use of OpenGL for the rendering of some examples.

Chapter 16

Module Index

16.1 Modules

Here is a list of all modules:

Chapter 17

Module Documentation a.k.a StarPU's API

17.1 Versioning

Macros

- #define STARPU_MAJOR_VERSION
- #define STARPU_MINOR_VERSION
- #define STARPU_RELEASE_VERSION

Functions

- void starpu_get_version (int *major, int *minor, int *release)

17.1.1 Detailed Description

17.1.2 Macro Definition Documentation

17.1.2.1 #define STARPU_MAJOR_VERSION

Define the major version of StarPU. This is the version used when compiling the application.

17.1.2.2 #define STARPU_MINOR_VERSION

Define the minor version of StarPU. This is the version used when compiling the application.

17.1.2.3 #define STARPU_RELEASE_VERSION

Define the release version of StarPU. This is the version used when compiling the application.

17.1.3 Function Documentation

17.1.3.1 void starpu_get_version (int * *major*, int * *minor*, int * *release*)

Return as 3 integers the version of StarPU used when running the application.

17.2 Initialization and Termination

Data Structures

- struct starpu_driver
- union starpu_driver.id
- struct starpu_vector_interface
- struct starpu_conf

Functions

- int starpu_init (struct starpu_conf *conf) STARPU_WARN_UNUSED_RESULT
- int starpu_conf_init (struct starpu_conf *conf)
- void starpu_shutdown (void)
- void starpu_pause ()
- void starpu_resume ()
- int starpu_asynchronous_copy_disabled (void)
- int starpu_asynchronous_cuda_copy_disabled (void)
- int starpu_asynchronous_opencl_copy_disabled (void)
- void starpu_topology_print (FILE *f)

17.2.1 Detailed Description

17.2.2 Data Structure Documentation

17.2.2.1 struct starpu_driver

structure for a driver

Data Fields

enum starpu_worker_archtype	type	The type of the driver. Only STARPU_CPU_WORKER, STARPU_CUDA_WORKER and STARPU_OPENCL_WORKER are currently supported.
union starpu_driver	id	The identifier of the driver.

17.2.2.2 union starpu_driver.id

Data Fields

unsigned	cpu_id	
unsigned	cuda_id	
cl_device_id	opencl_id	

17.2.2.3 struct starpu_vector_interface

Vector interface

vector interface for contiguous (non-strided) buffers

Data Fields

uintptr_t	ptr	local pointer of the vector
uintptr_t	dev_handle	device handle of the vector.
size_t	offset	offset in the vector
uint32_t	nx	number of elements on the x-axis of the vector
size_t	elemsize	size of the elements of the vector

17.2.2.4 struct starpu_conf

This structure is passed to the starpu_init() function in order to configure StarPU. It has to be initialized with starpu_conf_init(). When the default value is used, StarPU automatically selects the number of processing units and takes the default scheduling policy. The environment variables overwrite the equivalent parameters.

Data Fields

int	magic	Will be initialized by starpu_conf_init(). Should not be set by hand.
const char *	sched_policy_name	This is the name of the scheduling policy. This can also be specified with the environment variable STARPU_SCHED. (default = NULL).
struct starpu_sched_policy *	sched_policy	This is the definition of the scheduling policy. This field is ignored if starpu_conf::sched_policy_name is set. (default = NULL)
int	ncpus	This is the number of CPU cores that StarPU can use. This can also be specified with the environment variable STARPU_NCPU . (default = -1)
int	ncuda	This is the number of CUDA devices that StarPU can use. This can also be specified with the environment variable STARPU_NCUDA. (default = -1)
int	nopencl	This is the number of OpenCL devices that StarPU can use. This can also be specified with the environment variable STARPU_NOPENCL. (default = -1)
unsigned	use_explicit_workers_bindid	If this flag is set, the starpu_conf::workers_bindid array indicates where the different workers are bound, otherwise StarPU automatically selects where to bind the different workers. This can also be specified with the environment variable STARPU_WORKERS_CPUID. (default = 0)
unsigned	workers_bindid[STARPU_NMAXWORKERS]	If the starpu_conf::use_explicit_workers_bindid flag is set, this array indicates where to bind the different workers. The i-th entry of the starpu_conf::workers_bindid indicates the logical identifier of the processor which should execute the i-th worker. Note that the logical ordering of the CPUs is either determined by the OS, or provided by the hwloc library in case it is available.
unsigned	use_explicit_workers_cuda_gpuid	If this flag is set, the CUDA workers will be attached to the CUDA devices specified in the starpu_conf::workers_cuda_gpuid array. Otherwise, StarPU affects the CUDA devices in a round-robin fashion. This can also be specified with the environment variable STARPU_WORKERS_CUDAID. (default = 0)
unsigned	workers_cuda_gpuid[STARPU_NMAXWORKERS]	If the starpu_conf::use_explicit_workers_cuda_gpuid flag is set, this array contains the logical identifiers of the CUDA devices (as used by cudaGetDevice()).

unsigned	use_explicit↩ _workers_↩ opencl_gpuid	If this flag is set, the OpenCL workers will be attached to the OpenCL devices specified in the starpu_conf::workers_opencl_gpuid array. Otherwise, StarPU affects the OpenCL devices in a round-robin fashion. This can also be specified with the environment variable STARPU_WORK↩ ERS_OPENCLID. (default = 0)
unsigned	workers_↩ opencl_↩ gpuid[STA↩ RPU_NMAXW↩ ORKERS]	If the starpu_conf::use_explicit_workers_opencl_gpuid flag is set, this array contains the logical identifiers of the OpenCL devices to be used.
int	bus_calibrate	If this flag is set, StarPU will recalibrate the bus. If this value is equal to -1, the default value is used. This can also be specified with the environment variable STARPU_BUS_CALIBRATE. (default = 0)
int	calibrate	If this flag is set, StarPU will calibrate the performance models when executing tasks. If this value is equal to -1, the default value is used. If the value is equal to 1, it will force continuing calibration. If the value is equal to 2, the existing performance models will be overwritten. This can also be specified with the environment variable STARPU_CALIBRATE. (default = 0)
int	single_↩ combined_↩ worker	By default, StarPU executes parallel tasks concurrently. Some parallel libraries (e.g. most OpenMP implementations) however do not support concurrent calls to parallel code. In such case, setting this flag makes StarPU only start one parallel task at a time (but other CPU and GPU tasks are not affected and can be run concurrently). The parallel task scheduler will however still however still try varying combined worker sizes to look for the most efficient ones. This can also be specified with the environment variable STARPU_SINGLE_COMBINED_WOR↩ KER. (default = 0)
int	disable_↩ asynchronous↩ _copy	This flag should be set to 1 to disable asynchronous copies between CPUs and all accelerators. This can also be specified with the environment variable STARPU_DISABLE_ASYNCHRONOUS_COPY. The A↩ MD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers. This can also be specified at compilation time by giving to the configure script the option --disable-asynchronous-copy. (default = 0)
int	disable_↩ asynchronous↩ _cuda_copy	This flag should be set to 1 to disable asynchronous copies between CPUs and CUDA accelerators. This can also be specified with the environment variable STARPU_DISABLE_ASYNCHRONOUS_CUDA_C↩ OPY. This can also be specified at compilation time by giving to the configure script the option --disable-asynchronous-cuda-copy. (default = 0)
int	disable_↩ asynchronous↩ _opencl_copy	This flag should be set to 1 to disable asynchronous copies between C↩ PUs and OpenCL accelerators. This can also be specified with the environment variable STARPU_DISABLE_ASYNCHRONOUS_OPENCL_↩ COPY. The AMD implementation of OpenCL is known to fail when copying data asynchronously. When using this implementation, it is therefore necessary to disable asynchronous data transfers. This can also be specified at compilation time by giving to the configure script the option --disable-asynchronous-opencl-copy. (default = 0)

unsigned *	cuda_opengl_↩ interoperability	Enable CUDA/OpenGL interoperation on these CUDA devices. This can be set to an array of CUDA device identifiers for which cudaGLSetGL↩ Device() should be called instead of cudaSetDevice(). Its size is specified by the starpu_conf::n_cuda_opengl_interoperability field below (default = NULL)
unsigned	n_cuda_↩ opengl_↩ interoperability	
struct starpu_driver *	not_launched_↩ drivers	Array of drivers that should not be launched by StarPU. The application will run in one of its own threads. (default = NULL)
unsigned	n_not_↩ launched_↩ drivers	The number of StarPU drivers that should not be launched by StarPU. (default = 0)
unsigned	trace_buffer_↩ size	Specifies the buffer size used for FxT tracing. Starting from FxT version 0.2.12, the buffer will automatically be flushed when it fills in, but it may still be interesting to specify a bigger value to avoid any flushing (which would disturb the trace).

17.2.3 Function Documentation

17.2.3.1 int starpu_init (struct starpu_conf * *conf*)

This is StarPU initialization method, which must be called prior to any other StarPU call. It is possible to specify StarPU's configuration (e.g. scheduling policy, number of cores, ...) by passing a non-null argument. Default configuration is used if the passed argument is NULL. Upon successful completion, this function returns 0. Otherwise, -ENODEV indicates that no worker was available (so that StarPU was not initialized).

17.2.3.2 int starpu_conf_init (struct starpu_conf * *conf*)

This function initializes the conf structure passed as argument with the default values. In case some configuration parameters are already specified through environment variables, starpu_conf_init() initializes the fields of the structure according to the environment variables. For instance if STARPU_CALIBRATE is set, its value is put in the field starpu_conf::calibrate of the structure passed as argument. Upon successful completion, this function returns 0. Otherwise, -EINVAL indicates that the argument was NULL.

17.2.3.3 void starpu_shutdown (void)

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

17.2.3.4 void starpu_pause (void)

This call is used to suspend the processing of new tasks by workers. It can be used in a program where StarPU is used during only a part of the execution. Without this call, the workers continue to poll for new tasks in a tight loop, wasting CPU time. The symmetric call to starpu_resume() should be used to unfreeze the workers.

17.2.3.5 void starpu_resume (void)

This is the symmetrical call to starpu_pause(), used to resume the workers polling for new tasks.

17.2.3.6 int starpu_asynchronous_copy_disabled (void)

Return 1 if asynchronous data transfers between CPU and accelerators are disabled.

17.2.3.7 int starpu_asynchronous_cuda_copy_disabled (void)

Return 1 if asynchronous data transfers between CPU and CUDA accelerators are disabled.

17.2.3.8 int starpu_asynchronous_opencl_copy_disabled (void)

Return 1 if asynchronous data transfers between CPU and OpenCL accelerators are disabled.

17.2.3.9 void starpu_topology_print (FILE * *f*)

Prints a description of the topology on f.

17.3 Standard Memory Library

Macros

- #define starpu_data_malloc_pinned_if_possible
- #define starpu_data_free_pinned_if_possible
- #define STARPU_MALLOC_PINNED
- #define STARPU_MALLOC_COUNT

Functions

- int starpu_malloc_flags (void **A, size_t dim, int flags)
- void starpu_malloc_set_align (size_t align)
- int starpu_malloc (void **A, size_t dim)
- int starpu_free (void *A)
- int starpu_free_flags (void *A, size_t dim, int flags)
- starpu_ssize_t starpu_memory_get_total (unsigned node)
- starpu_ssize_t starpu_memory_get_available (unsigned node)

17.3.1 Detailed Description

17.3.2 Macro Definition Documentation

17.3.2.1 #define starpu_data_malloc_pinned_if_possible

Deprecated Equivalent to starpu_malloc(). This macro is provided to avoid breaking old codes.

17.3.2.2 #define starpu_data_free_pinned_if_possible

Deprecated Equivalent to starpu_free(). This macro is provided to avoid breaking old codes.

17.3.2.3 #define STARPU_MALLOC_PINNED

Value passed to the function starpu_malloc_flags() to indicate the memory allocation should be pinned.

17.3.2.4 #define STARPU_MALLOC_COUNT

Value passed to the function starpu_malloc_flags() to indicate the memory allocation should be in the limit defined by the environment variables STARPU_LIMIT_CUDA_devid_MEM, STARPU_LIMIT_CUDA_MEM, STARPU_LIMIT_OPENCL_devid_MEM, STARPU_LIMIT_OPENCL_MEM and STARPU_LIMIT_CPU_MEM (see Section How to limit memory per node). If no memory is available, it tries to reclaim memory from StarPU. Memory allocated this way needs to be freed by calling the function starpu_free_flags() with the same flag.

17.3.3 Function Documentation

17.3.3.1 int starpu_malloc_flags (void ** *A*, size_t *dim*, int *flags*)

Performs a memory allocation based on the constraints defined by the given flag.

17.3.3.2 void starpu_malloc_set_align (size_t *align*)

This function sets an alignment constraints for starpu_malloc() allocations. align must be a power of two. This is for instance called automatically by the OpenCL driver to specify its own alignment constraints.

17.3.3.3 int starpu_malloc (void ** *A*, size_t *dim*)

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

17.3.3.4 int starpu_free (void * *A*)

This function frees memory which has previously been allocated with starpu_malloc().

17.3.3.5 int starpu_free_flags (void * *A*, size_t *dim*, int *flags*)

This function frees memory by specifying its size. The given flags should be consistent with the ones given to starpu_malloc_flags() when allocating the memory.

17.3.3.6 ssize_t starpu_memory_get_total (unsigned *node*)

If a memory limit is defined on the given node (see Section How to limit memory per node), return the amount of total memory on the node. Otherwise return -1.

17.3.3.7 ssize_t starpu_memory_get_available (unsigned *node*)

If a memory limit is defined on the given node (see Section How to limit memory per node), return the amount of available memory on the node. Otherwise return -1.

17.4 Toolbox

The following macros allow to make GCC extensions portable, and to have a code which can be compiled with any C compiler.

Macros

- #define STARPU_GNUC_PREREQ(maj, min)
- #define STARPU_UNLIKELY(expr)
- #define STARPU_LIKELY(expr)
- #define STARPU_ATTRIBUTE_UNUSED
- #define STARPU_ATTRIBUTE_INTERNAL
- #define STARPU_ATTRIBUTE_MALLOC
- #define STARPU_ATTRIBUTE_WARN_UNUSED_RESULT
- #define STARPU_ATTRIBUTE_PURE
- #define STARPU_ATTRIBUTE_ALIGNED(size)
- #define STARPU_WARN_UNUSED_RESULT
- #define STARPU_POISON_PTR
- #define STARPU_MIN(a, b)
- #define STARPU_MAX(a, b)
- #define STARPU_ASSERT(x)
- #define STARPU_ASSERT_MSG(x, msg,...)
- #define STARPU_ABORT()
- #define STARPU_ABORT_MSG(msg,...)
- #define STARPU_CHECK_RETURN_VALUE(err, message,...)
- #define STARPU_CHECK_RETURN_VALUE_IS(err, value, message,...)
- #define STARPU_RMB()
- #define STARPU_WMB()

Functions

- static __starpu_inline int starpu_get_env_number (const char *str)

17.4.1 Detailed Description

The following macros allow to make GCC extensions portable, and to have a code which can be compiled with any C compiler.

17.4.2 Macro Definition Documentation

17.4.2.1 #define STARPU_GNUC_PREREQ(*maj, min*)

Return true (non-zero) if GCC version MAJ.MIN or later is being used (macro taken from glibc.)

17.4.2.2 #define STARPU_UNLIKELY(*expr*)

When building with a GNU C Compiler, this macro allows programmers to mark an expression as unlikely.

17.4.2.3 #define STARPU_LIKELY(*expr*)

When building with a GNU C Compiler, this macro allows programmers to mark an expression as likely.

17.4.2.4 #define STARPU_ATTRIBUTE_UNUSED

When building with a GNU C Compiler, this macro is defined to __attribute__((unused))

17.4.2.5 #define STARPU_ATTRIBUTE_INTERNAL

When building with a GNU C Compiler, this macro is defined to __attribute__((visibility ("internal")))

17.4.2.6 #define STARPU_ATTRIBUTE_MALLOC

When building with a GNU C Compiler, this macro is defined to __attribute__((malloc))

17.4.2.7 #define STARPU_ATTRIBUTE_WARN_UNUSED_RESULT

When building with a GNU C Compiler, this macro is defined to __attribute__((warn_unused_result))

17.4.2.8 #define STARPU_ATTRIBUTE_PURE

When building with a GNU C Compiler, this macro is defined to __attribute__((pure))

17.4.2.9 #define STARPU_ATTRIBUTE_ALIGNED(*size*)

When building with a GNU C Compiler, this macro is defined to__attribute__((aligned(size)))

17.4.2.10 #define STARPU_WARN_UNUSED_RESULT

When building with a GNU C Compiler, this macro is defined to__attribute__((__warn_unused_result__))

17.4.2.11 #define STARPU_POISON_PTR

This macro defines a value which can be used to mark pointers as invalid values.

17.4.2.12 #define STARPU_MIN(*a, b*)

This macro returns the min of the two parameters.

17.4.2.13 #define STARPU_MAX(*a, b*)

This macro returns the max of the two parameters.

17.4.2.14 #define STARPU_ASSERT(*x*)

Unless StarPU has been configured with the option --enable-fast, this macro will abort if the expression is false.

17.4.2.15 #define STARPU_ASSERT_MSG(*x, msg, ...*)

Unless StarPU has been configured with the option --enable-fast, this macro will abort if the expression is false. The given message will be displayed.

17.4.2.16 #define STARPU_ABORT( )

This macro aborts the program.

17.4.2.17 #define STARPU_ABORT_MSG(*msg, ...*)

This macro aborts the program, and displays the given message.

17.4.2.18 #define STARPU_CHECK_RETURN_VALUE(*err, message, ...*)

If `err` has a value which is not 0, the given message is displayed before aborting.

17.4.2.19 #define STARPU_CHECK_RETURN_VALUE_IS(*err, value, message, ...*)

If `err` has a value which is not `value`, the given message is displayed before aborting.

17.4.2.20 #define STARPU_RMB( )

This macro can be used to do a synchronization.

17.4.2.21 #define STARPU_WMB( )

This macro can be used to do a synchronization.

17.4.3 Function Documentation

17.4.3.1 int starpu_get_env_number (const char * *str*) [static]

If `str` is the name of a existing environment variable which is defined to an integer, the function returns the value of the integer. It returns 0 otherwise.

17.5 Threads

This section describes the thread facilities provided by StarPU. The thread function are either implemented on top of the pthread library or the Simgrid library when the simulated performance mode is enabled (Simulated Performance).

Macros

- #define STARPU_PTHREAD_CREATE_ON(name, thread, attr, routine, arg, where)
- #define STARPU_PTHREAD_CREATE(thread, attr, routine, arg)
- #define STARPU_PTHREAD_MUTEX_INIT(mutex, attr)
- #define STARPU_PTHREAD_MUTEX_DESTROY(mutex)
- #define STARPU_PTHREAD_MUTEX_LOCK(mutex)
- #define STARPU_PTHREAD_MUTEX_UNLOCK(mutex)
- #define STARPU_PTHREAD_KEY_CREATE(key, destr)
- #define STARPU_PTHREAD_KEY_DELETE(key)
- #define STARPU_PTHREAD_SETSPECIFIC(key, ptr)
- #define STARPU_PTHREAD_GETSPECIFIC(key)
- #define STARPU_PTHREAD_RWLOCK_INIT(rwlock, attr)
- #define STARPU_PTHREAD_RWLOCK_RDLOCK(rwlock)
- #define STARPU_PTHREAD_RWLOCK_WRLOCK(rwlock)
- #define STARPU_PTHREAD_RWLOCK_UNLOCK(rwlock)
- #define STARPU_PTHREAD_RWLOCK_DESTROY(rwlock)

- #define STARPU_PTHREAD_COND_INIT(cond, attr)
- #define STARPU_PTHREAD_COND_DESTROY(cond)
- #define STARPU_PTHREAD_COND_SIGNAL(cond)
- #define STARPU_PTHREAD_COND_BROADCAST(cond)
- #define STARPU_PTHREAD_COND_WAIT(cond, mutex)
- #define STARPU_PTHREAD_BARRIER_INIT(barrier, attr, count)
- #define STARPU_PTHREAD_BARRIER_DESTROY(barrier)
- #define STARPU_PTHREAD_BARRIER_WAIT(barrier)
- #define STARPU_PTHREAD_MUTEX_INITIALIZER
- #define STARPU_PTHREAD_COND_INITIALIZER

Functions

- int **starpu_pthread_create_on** (char *name, starpu_pthread_t *thread, const starpu_pthread_attr_t *attr, void *(*start_routine)(void *), void *arg, int where)
- int starpu_pthread_create (starpu_pthread_t *thread, const starpu_pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
- int starpu_pthread_join (starpu_pthread_t thread, void **retval)
- int starpu_pthread_attr_init (starpu_pthread_attr_t *attr)
- int starpu_pthread_attr_destroy (starpu_pthread_attr_t *attr)
- int starpu_pthread_attr_setdetachstate (starpu_pthread_attr_t *attr, int detachstate)
- int starpu_pthread_mutex_init (starpu_pthread_mutex_t *mutex, const starpu_pthread_mutexattr_t *mutexattr)
- int starpu_pthread_mutex_destroy (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_lock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_unlock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_trylock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_key_create (starpu_pthread_key_t *key, void(*destr_function)(void *))
- int starpu_pthread_key_delete (starpu_pthread_key_t key)
- int starpu_pthread_setspecific (starpu_pthread_key_t key, const void *pointer)
- void * starpu_pthread_getspecific (starpu_pthread_key_t key)
- int starpu_pthread_cond_init (starpu_pthread_cond_t *cond, starpu_pthread_condattr_t *cond_attr)
- int starpu_pthread_cond_signal (starpu_pthread_cond_t *cond)
- int starpu_pthread_cond_broadcast (starpu_pthread_cond_t *cond)
- int starpu_pthread_cond_wait (starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex)
- int starpu_pthread_cond_timedwait (starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex, const struct timespec *abstime)
- int starpu_pthread_cond_destroy (starpu_pthread_cond_t *cond)
- int starpu_pthread_rwlock_init (starpu_pthread_rwlock_t *rwlock, const starpu_pthread_rwlockattr_t *attr)
- int starpu_pthread_rwlock_destroy (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_rdlock (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_wrlock (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_unlock (starpu_pthread_rwlock_t *rwlock)

17.5.1 Detailed Description

This section describes the thread facilities provided by StarPU. The thread function are either implemented on top of the pthread library or the Simgrid library when the simulated performance mode is enabled (Simulated Performance).

17.5.2 Macro Definition Documentation

17.5.2.1 #define STARPU_PTHREAD_CREATE_ON(*name, thread, attr, routine, arg, where*)

This macro calls the function starpu_pthread_create_on() and aborts on error.

17.5.2.2 #define STARPU_PTHREAD_CREATE(*thread, attr, routine, arg*)

This macro calls the function starpu_pthread_create() and aborts on error.

17.5.2.3 #define STARPU_PTHREAD_MUTEX_INIT(*mutex, attr*)

This macro calls the function starpu_pthread_mutex_init() and aborts on error.

17.5.2.4 #define STARPU_PTHREAD_MUTEX_DESTROY(*mutex*)

This macro calls the function starpu_pthread_mutex_destroy() and aborts on error.

17.5.2.5 #define STARPU_PTHREAD_MUTEX_LOCK(*mutex*)

This macro calls the function starpu_pthread_mutex_lock() and aborts on error.

17.5.2.6 #define STARPU_PTHREAD_MUTEX_UNLOCK(*mutex*)

This macro calls the function starpu_pthread_mutex_unlock() and aborts on error.

17.5.2.7 #define STARPU_PTHREAD_KEY_CREATE(*key, destr*)

This macro calls the function starpu_pthread_key_create() and aborts on error.

17.5.2.8 #define STARPU_PTHREAD_KEY_DELETE(*key*)

This macro calls the function starpu_pthread_key_delete() and aborts on error.

17.5.2.9 #define STARPU_PTHREAD_SETSPECIFIC(*key, ptr*)

This macro calls the function starpu_pthread_setspecific() and aborts on error.

17.5.2.10 #define STARPU_PTHREAD_GETSPECIFIC(*key*)

This macro calls the function starpu_pthread_getspecific() and aborts on error.

17.5.2.11 #define STARPU_PTHREAD_RWLOCK_INIT(*rwlock, attr*)

This macro calls the function starpu_pthread_rwlock_init() and aborts on error.

17.5.2.12 #define STARPU_PTHREAD_RWLOCK_RDLOCK(*rwlock*)

This macro calls the function starpu_pthread_rwlock_rdlock() and aborts on error.

17.5.2.13 #define STARPU_PTHREAD_RWLOCK_WRLOCK(*rwlock*)

This macro calls the function starpu_pthread_rwlock_wrlock() and aborts on error.

17.5.2.14 #define STARPU_PTHREAD_RWLOCK_UNLOCK(*rwlock*)

This macro calls the function starpu_pthread_rwlock_unlock() and aborts on error.

17.5.2.15 #define STARPU_PTHREAD_RWLOCK_DESTROY(*rwlock*)

This macro calls the function starpu_pthread_rwlock_destroy() and aborts on error.

17.5.2.16 #define STARPU_PTHREAD_COND_INIT(*cond, attr*)

This macro calls the function starpu_pthread_cond_init() and aborts on error.

17.5.2.17 #define STARPU_PTHREAD_COND_DESTROY(*cond*)

This macro calls the function starpu_pthread_cond_destroy() and aborts on error.

17.5.2.18 #define STARPU_PTHREAD_COND_SIGNAL(*cond*)

This macro calls the function starpu_pthread_cond_signal() and aborts on error.

17.5.2.19 #define STARPU_PTHREAD_COND_BROADCAST(*cond*)

This macro calls the function starpu_pthread_cond_broadcast() and aborts on error.

17.5.2.20 #define STARPU_PTHREAD_COND_WAIT(*cond, mutex*)

This macro calls the function starpu_pthread_cond_wait() and aborts on error.

17.5.2.21 #define STARPU_PTHREAD_BARRIER_INIT(*barrier, attr, count*)

This macro calls the function starpu_pthread_barrier_init() and aborts on error.

17.5.2.22 #define STARPU_PTHREAD_BARRIER_DESTROY(*barrier*)

This macro calls the function starpu_pthread_barrier_destroy() and aborts on error.

17.5.2.23 #define STARPU_PTHREAD_BARRIER_WAIT(*barrier*)

This macro calls the function starpu_pthread_barrier_wait() and aborts on error.

17.5.2.24 STARPU_PTHREAD_MUTEX_INITIALIZER

This macro initializes the mutex given in parameter.

17.5.2.25 STARPU_PTHREAD_COND_INITIALIZER

This macro initializes the condition variable given in parameter.

17.5.3 Function Documentation

17.5.3.1 int starpu_pthread_create (starpu_pthread_t * *thread*, const starpu_pthread_attr_t * *attr*, void *(*)(void *) *start_routine*, void * *arg*)

This function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine`; `arg` is passed as the sole argument of `start_routine`.

17.5.3.2 int starpu_pthread_join (starpu_pthread_t *thread*, void ** *retval*)

This function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then the function returns immediately. The thread specified by `thread` must be joinable.

17.5.3.3 int starpu_pthread_attr_init (starpu_pthread_attr_t * *attr*)

This function initializes the thread attributes object pointed to by `attr` with default attribute values.

It does not do anything when the simulated performance mode is enabled (Simulated Performance).

17.5.3.4 int starpu_pthread_attr_destroy (starpu_pthread_attr_t * *attr*)

This function destroys a thread attributes object which is no longer required. Destroying a thread attributes object has no effect on threads that were created using that object.

It does not do anything when the simulated performance mode is enabled (Simulated Performance).

17.5.3.5 int starpu_pthread_attr_setdetachstate (starpu_pthread_attr_t * *attr*, int *detachstate*)

This function sets the detach state attribute of the thread attributes object referred to by `attr` to the value specified in `detachstate`. The detach state attribute determines whether a thread created using the thread attributes object `attr` will be created in a joinable or a detached state.

It does not do anything when the simulated performance mode is enabled (Simulated Performance).

17.5.3.6 int starpu_pthread_mutex_init (starpu_pthread_mutex_t * *mutex*, const starpu_pthread_mutexattr_t * *mutexattr*)

This function initializes the mutex object pointed to by `mutex` according to the mutex attributes specified in `mutexattr`. If `mutexattr` is NULL, default attributes are used instead.

17.5.3.7 int starpu_pthread_mutex_destroy (starpu_pthread_mutex_t * *mutex*)

This function destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance.

17.5.3.8 int starpu_pthread_mutex_lock (starpu_pthread_mutex_t * *mutex*)

This function locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and the function returns immediately. If the mutex is already locked by another thread, the function suspends the calling thread until the mutex is unlocked.

17.5.3.9 int starpu_pthread_mutex_unlock (starpu_pthread_mutex_t * *mutex*)

This function unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to starpu_pthread_mutex_unlock().

17.5.3.10 int starpu_pthread_mutex_trylock (starpu_pthread_mutex_t * *mutex*)

This function behaves identically to starpu_pthread_mutex_lock(), except that it does not block the calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a “fast” mutex). Instead, the function returns immediately with the error code EBUSY.

17.5.3.11 int starpu_pthread_key_create (starpu_pthread_key_t * *key,* void(*)(void *) *destr_function*)

This function allocates a new TSD key. The key is stored in the location pointed to by `key`.

17.5.3.12 int starpu_pthread_key_delete (starpu_pthread_key_t *key*)

This function deallocates a TSD key. It does not check whether non-NULL values are associated with that key in the currently executing threads, nor call the destructor function associated with the key.

17.5.3.13 int starpu_pthread_setspecific (starpu_pthread_key_t *key,* const void * *pointer*)

This function changes the value associated with `key` in the calling thread, storing the given `pointer` instead.

17.5.3.14 * starpu_pthread_getspecific (starpu_pthread_key_t *key*)

This function returns the value associated with `key` on success, and NULL on error.

17.5.3.15 starpu_pthread_cond_init (starpu_pthread_cond_t * *cond,* starpu_pthread_condattr_t * *cond_attr*)

This function initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`, or default attributes if `cond_attr` is NULL.

17.5.3.16 starpu_pthread_cond_signal (starpu_pthread_cond_t * *cond*)

This function restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted, but it not specified which.

17.5.3.17 starpu_pthread_cond_broadcast (starpu_pthread_cond_t * *cond*)

This function restarts all the threads that are waiting on the condition variable `cond`. Nothing happens if no threads are waiting on cond.

17.5.3.18 starpu_pthread_cond_wait (starpu_pthread_cond_t * *cond,* starpu_pthread_mutex_t * *mutex*)

This function atomically unlocks the mutex (as per starpu_pthread_mutex_unlock()) and waits for the condition variable `cond` to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to starpu_pthread_cond_wait(). Before returning to the calling thread, the function re-acquires mutex (as per starpu_pthread_mutex_lock()).

17.5.3.19 starpu_pthread_cond_timedwait (starpu_pthread_cond_t * *cond,* starpu_pthread_mutex_t * *mutex,* const struct timespec * *abstime*)

This function atomically unlocks `mutex` and waits on `cond`, as starpu_pthread_cond_wait() does, but it also bounds the duration of the wait.

17.5.3.20 starpu_pthread_cond_destroy (starpu_pthread_cond_t * *cond*)

This function destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to the function.

17.5.3.21 starpu_pthread_rwlock_init (starpu_pthread_rwlock_t * *rwlock,* const starpu_pthread_rwlockattr_t * *attr*)

This function is the same as starpu_pthread_mutex_init().

17.5.3.22 starpu_pthread_rwlock_destroy (starpu_pthread_rwlock_t * *rwlock*)

This function is the same as starpu_pthread_mutex_destroy().

17.5.3.23 starpu_pthread_rwlock_rdlock (starpu_pthread_rwlock_t * *rwlock*)

This function is the same as starpu_pthread_mutex_lock().

17.5.3.24 starpu_pthread_rwlock_wrlock (starpu_pthread_rwlock_t * *rwlock*)

This function is the same as starpu_pthread_mutex_lock().

17.5.3.25 starpu_pthread_rwlock_unlock (starpu_pthread_rwlock_t * *rwlock*)

This function is the same as starpu_pthread_mutex_unlock().

17.6 Workers' Properties

Data Structures

- struct starpu_worker_collection
- struct starpu_sched_ctx_iterator

Macros

- #define STARPU_NMAXWORKERS

Enumerations

- enum starpu_node_kind { STARPU_UNUSED, STARPU_CPU_RAM, STARPU_CUDA_RAM, STARPU_↩ OPENCL_RAM }
- enum starpu_worker_archtype { STARPU_ANY_WORKER, STARPU_CPU_WORKER, STARPU_CUDA_↩ WORKER, STARPU_OPENCL_WORKER }
- enum starpu_worker_collection_type { STARPU_WORKER_LIST }

Functions

- unsigned starpu_worker_get_count (void)
- int starpu_worker_get_count_by_type (enum starpu_worker_archtype type)
- unsigned starpu_cpu_worker_get_count (void)
- unsigned starpu_cuda_worker_get_count (void)

- unsigned starpu_opencl_worker_get_count (void)
- int starpu_worker_get_id (void)
- int starpu_worker_get_ids_by_type (enum starpu_worker_archtype type, int *workerids, int maxsize)
- int starpu_worker_get_by_type (enum starpu_worker_archtype type, int num)
- int starpu_worker_get_by_devid (enum starpu_worker_archtype type, int devid)
- int starpu_worker_get_devid (int id)
- enum starpu_worker_archtype starpu_worker_get_type (int id)
- void starpu_worker_get_name (int id, char *dst, size_t maxlen)
- unsigned starpu_worker_get_memory_node (unsigned workerid)
- enum starpu_node_kind starpu_node_get_kind (unsigned node)

17.6.1 Detailed Description

17.6.2 Data Structure Documentation

17.6.2.1 struct starpu_worker_collection

A scheduling context manages a collection of workers that can be memorized using different data structures. Thus, a generic structure is available in order to simplify the choice of its type. Only the list data structure is available but further data structures(like tree) implementations are foreseen.

Data Fields

- void * workerids
- unsigned nworkers
- enum starpu_worker_collection_type type
- unsigned(* has_next)(struct starpu_worker_collection *workers, struct starpu_sched_ctx_iterator *it)
- int(* get_next)(struct starpu_worker_collection *workers, struct starpu_sched_ctx_iterator *it)
- int(* add)(struct starpu_worker_collection *workers, int worker)
- int(* remove)(struct starpu_worker_collection *workers, int worker)
- void(* init)(struct starpu_worker_collection *workers)
- void(* deinit)(struct starpu_worker_collection *workers)
- void(* init_iterator)(struct starpu_worker_collection *workers, struct starpu_sched_ctx_iterator *it)

17.6.2.1.1 Field Documentation

17.6.2.1.1.1 starpu_worker_collection::workerids

The workerids managed by the collection

17.6.2.1.1.2 starpu_worker_collection::nworkers

The number of workers in the collection

17.6.2.1.1.3 starpu_worker_collection::type

The type of structure (currently STARPU_WORKER_LIST is the only one available)

17.6.2.1.1.4 starpu_worker_collection::has_next

Checks if there is another element in collection

17.6.2.1.1.5 starpu_worker_collection::get_next

return the next element in the collection

17.6.2.1.1.6 starpu_worker_collection::add

add a new element in the collection

17.6.2.1.1.7 starpu_worker_collection::remove

remove an element from the collection

17.6.2.1.1.8 starpu_worker_collection::init

Initialize the collection

17.6.2.1.1.9 starpu_worker_collection::deinit

Deinitialize the colection

17.6.2.1.1.10 starpu_worker_collection::init_iterator

Initialize the cursor if there is one

17.6.2.2 struct starpu_sched_ctx_iterator

Structure needed to iterate on the collection

Data Fields

int	cursor	The index of the current worker in the collection, needed when iterating on the collection.

17.6.3 Macro Definition Documentation

17.6.3.1 #define STARPU_NMAXWORKERS

Define the maximum number of workers managed by StarPU.

17.6.4 Enumeration Type Documentation

17.6.4.1 enum starpu_node_kind

TODO

Enumerator

STARPU_UNUSED TODO

STARPU_CPU_RAM TODO

STARPU_CUDA_RAM TODO

STARPU_OPENCL_RAM TODO

17.6.4.2 enum starpu_worker_archtype

Worker Architecture Type

Enumerator

STARPU_ANY_WORKER any worker, used in the hypervisor

STARPU_CPU_WORKER CPU core

STARPU_CUDA_WORKER NVIDIA CUDA device
STARPU_OPENCL_WORKER OpenCL device

17.6.4.3 enum starpu_worker_collection_type

Types of structures the worker collection can implement

Enumerator

STARPU_WORKER_LIST The collection is an array

17.6.5 Function Documentation

17.6.5.1 unsigned starpu_worker_get_count (void)

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

17.6.5.2 int starpu_worker_get_count_by_type (enum starpu_worker_archtype *type*)

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

17.6.5.3 unsigned starpu_cpu_worker_get_count (void)

This function returns the number of CPUs controlled by StarPU. The returned value should be at most STARPU_↩ MAXCPUS.

17.6.5.4 unsigned starpu_cuda_worker_get_count (void)

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

17.6.5.5 unsigned starpu_opencl_worker_get_count (void)

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

17.6.5.6 int starpu_worker_get_id (void)

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

17.6.5.7 int starpu_worker_get_ids_by_type (enum starpu_worker_archtype *type*, int * *workerids*, int *maxsize*)

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

17.6.5.8 int starpu_worker_get_by_type (enum starpu_worker_archtype *type*, int *num*)

This returns the identifier of the num-th worker that has the specified type type. If there are no such worker, -1 is returned.

17.6.5.9 int starpu_worker_get_by_devid (enum starpu_worker_archtype *type*, int *devid*)

This returns the identifier of the worker that has the specified type `type` and device id `devid` (which may not be the n-th, if some devices are skipped for instance). If there are no such worker, -1 is returned.

17.6.5.10 int starpu_worker_get_devid (int *id*)

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

17.6.5.11 enum starpu_worker_archtype starpu_worker_get_type (int *id*)

This function returns the type of processing unit associated to a worker. The worker identifier is a value returned by the function starpu_worker_get_id()). The returned value indicates the architecture of the worker: STARPU_CP↩U_WORKER for a CPU core, STARPU_CUDA_WORKER for a CUDA device, and STARPU_OPENCL_WORKER for a OpenCL device. The value returned for an invalid identifier is unspecified.

17.6.5.12 void starpu_worker_get_name (int *id*, char * *dst*, size_t *maxlen*)

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

17.6.5.13 unsigned starpu_worker_get_memory_node (unsigned *workerid*)

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

17.6.5.14 enum starpu_node_kind starpu_node_get_kind (unsigned *node*)

Returns the type of the given node as defined by starpu_node_kind. For example, when defining a new data interface, this function should be used in the allocation function to determine on which device the memory needs to be allocated.

17.7 Data Management

This section describes the data management facilities provided by StarPU. We show how to use existing data interfaces in Data Interfaces, but developers can design their own data interfaces if required.

Typedefs

- typedef struct _starpu_data_state * starpu_data_handle_t

Enumerations

- enum starpu_data_access_mode {
  STARPU_NONE, STARPU_R, STARPU_W, STARPU_RW,
  STARPU_SCRATCH, STARPU_REDUX }

Basic Data Management API

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

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

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

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

- void starpu_data_register (starpu_data_handle_t *handleptr, unsigned home_node, void *data_interface, struct starpu_data_interface_ops *ops)
- void starpu_data_ptr_register (starpu_data_handle_t handle, unsigned node)
- void starpu_data_register_same (starpu_data_handle_t *handledst, starpu_data_handle_t handlesrc)
- void starpu_data_unregister (starpu_data_handle_t handle)
- void starpu_data_unregister_no_coherency (starpu_data_handle_t handle)
- void starpu_data_unregister_submit (starpu_data_handle_t handle)
- void starpu_data_invalidate (starpu_data_handle_t handle)
- void starpu_data_invalidate_submit (starpu_data_handle_t handle)
- void starpu_data_set_wt_mask (starpu_data_handle_t handle, uint32_t wt_mask)
- int starpu_data_prefetch_on_node (starpu_data_handle_t handle, unsigned node, unsigned async)
- starpu_data_handle_t starpu_data_lookup (const void *ptr)
- int starpu_data_request_allocation (starpu_data_handle_t handle, unsigned node)
- void starpu_data_query_status (starpu_data_handle_t handle, int memory_node, int *is_allocated, int *is_↩ valid, int *is_requested)
- void starpu_data_advise_as_important (starpu_data_handle_t handle, unsigned is_important)
- void starpu_data_set_reduction_methods (starpu_data_handle_t handle, struct starpu_codelet *redux_cl, struct starpu_codelet *init_cl)

Access registered data from the application

- #define STARPU_DATA_ACQUIRE_CB(handle, mode, code)
- int starpu_data_acquire (starpu_data_handle_t handle, enum starpu_data_access_mode mode)
- int starpu_data_acquire_cb (starpu_data_handle_t handle, enum starpu_data_access_mode mode, void(*callback)(void *), void *arg)
- void starpu_data_release (starpu_data_handle_t handle)

17.7.1 Detailed Description

This section describes the data management facilities provided by StarPU. We show how to use existing data interfaces in Data Interfaces, but developers can design their own data interfaces if required.

17.7.2 Macro Definition Documentation

17.7.2.1 #define STARPU_DATA_ACQUIRE_CB(*handle, mode, code*)

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

17.7.3 Typedef Documentation

17.7.3.1 starpu_data_handle_t

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

17.7.4 Enumeration Type Documentation

17.7.4.1 enum starpu_data_access_mode

This datatype describes a data access mode.

Enumerator

STARPU_NONE TODO

STARPU_R read-only mode.

STARPU_W write-only mode.

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

STARPU_SCRATCH A temporary buffer is allocated for the task, but StarPU does not enforce data consistency—i.e. each device has its own buffer, independently from each other (even for CPUs), and no data transfer is ever performed. This is useful for temporary variables to avoid allocating/freeing buffers inside each task. Currently, no behavior is defined concerning the relation with the STARPU_R and STARPU_W modes and the value provided at registration — i.e., the value of the scratch buffer is undefined at entry of the codelet function. It is being considered for future extensions at least to define the initial value. For now, data to be used in STARPU_SCRATCH mode should be registered with node `-1` and a `NULL` pointer, since the value of the provided buffer is simply ignored for now.

STARPU_REDUX todo

17.7.5 Function Documentation

17.7.5.1 void starpu_data_register (starpu_data_handle_t * *handleptr,* unsigned *home_node,* void * *data_interface,* struct starpu_data_interface_ops * *ops*)

Register a piece of data into the handle located at the `handleptr` address. The `data_interface` buffer contains the initial description of the data in the `home_node`. The `ops` argument is a pointer to a structure describing the different methods used to manipulate this type of interface. See starpu_data_interface_ops for more details on this structure. If `home_node` is -1, StarPU will automatically allocate the memory when it is used for the first time in write-only mode. Once such data handle has been automatically allocated, it is possible to access it using any access mode. Note that StarPU supplies a set of predefined types of interface (e.g. vector or matrix) which can be registered by the means of helper functions (e.g. starpu_vector_data_register() or starpu_matrix_data_register()).

17.7.5.2 void starpu_data_ptr_register (starpu_data_handle_t *handle*, unsigned *node*)

Register that a buffer for `handle` on `node` will be set. This is typically used by starpu_*_ptr_register helpers before setting the interface pointers for this node, to tell the core that that is now allocated.

17.7.5.3 void starpu_data_register_same (starpu_data_handle_t * *handledst*, starpu_data_handle_t *handlesrc*)

Register a new piece of data into the handle `handledst` with the same interface as the handle `handlesrc`.

17.7.5.4 void starpu_data_unregister (starpu_data_handle_t *handle*)

This function unregisters a data handle from StarPU. If the data was automatically allocated by StarPU because the home node was -1, all automatically allocated buffers are freed. Otherwise, a valid copy of the data is put back into the home node in the buffer that was initially registered. Using a data handle that has been unregistered from StarPU results in an undefined behaviour. In case we do not need to update the value of the data in the home node, we can use the function starpu_data_unregister_no_coherency() instead.

17.7.5.5 void starpu_data_unregister_no_coherency (starpu_data_handle_t *handle*)

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

17.7.5.6 void starpu_data_unregister_submit (starpu_data_handle_t *handle*)

Destroy the data handle once it is not needed anymore by any submitted task. No coherency is assumed.

17.7.5.7 void starpu_data_invalidate (starpu_data_handle_t *handle*)

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

17.7.5.8 void starpu_data_invalidate_submit (starpu_data_handle_t *handle*)

Submits invalidation of the data handle after completion of previously submitted tasks.

17.7.5.9 void starpu_data_set_wt_mask (starpu_data_handle_t *handle*, uint32_t *wt_mask*)

This function sets the write-through mask of a given data (and its children), i.e. a bitmask of nodes where the data should be always replicated after modification. It also prevents the data from being evicted from these nodes when memory gets scarse. When the data is modified, it is automatically transfered into those memory node. For instance a `1<<0` write-through mask means that the CUDA workers will commit their changes in main memory (node 0).

17.7.5.10 int starpu_data_prefetch_on_node (starpu_data_handle_t *handle*, unsigned *node*, unsigned *async*)

Issue a prefetch request for a given data to a given node, i.e. requests that the data be replicated to the given node, so that it is available there for tasks. If the `async` parameter is 0, the call will block until the transfer is achieved, else the call will return immediately, after having just queued the request. In the latter case, the request will asynchronously wait for the completion of any task writing on the data.

17.7.5.11 starpu_data_handle_t starpu_data_lookup (const void * *ptr*)

Return the handle corresponding to the data pointed to by the `ptr` host pointer.

17.7.5.12 int starpu_data_request_allocation (starpu_data_handle_t *handle*, unsigned *node*)

Explicitly ask StarPU to allocate room for a piece of data on the specified memory node.

17.7.5.13 void starpu_data_query_status (starpu_data_handle_t *handle*, int *memory_node*, int * *is_allocated*, int * *is_valid*, int * *is_requested*)

Query the status of `handle` on the specified `memory_node`.

17.7.5.14 void starpu_data_advise_as_important (starpu_data_handle_t *handle*, unsigned *is_important*)

This function allows to specify that a piece of data can be discarded without impacting the application.

17.7.5.15 void starpu_data_set_reduction_methods (starpu_data_handle_t *handle*, struct starpu_codelet * *redux_cl*, struct starpu_codelet * *init_cl*)

This sets the codelets to be used for `handle` when it is accessed in the mode STARPU_REDUX. Per-worker buffers will be initialized with the codelet `init_cl`, and reduction between per-worker buffers will be done with the codelet `redux_cl`.

17.7.5.16 int starpu_data_acquire (starpu_data_handle_t *handle*, enum starpu_data_access_mode *mode*)

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

17.7.5.17 int starpu_data_acquire_cb (starpu_data_handle_t *handle*, enum starpu_data_access_mode *mode*, void(*)(void *) *callback*, void * *arg*)

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

17.7.5.18 void starpu_data_release (starpu_data_handle_t *handle*)

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

17.8 Data Interfaces

Data Structures

- struct starpu_data_interface_ops

- struct starpu_data_copy_methods
- struct starpu_variable_interface
- struct starpu_vector_interface
- struct starpu_matrix_interface
- struct starpu_block_interface
- struct starpu_bcsr_interface
- struct starpu_csr_interface
- struct starpu_coo_interface

Enumerations

- enum starpu_data_interface_id {
  STARPU_UNKNOWN_INTERFACE_ID, STARPU_MATRIX_INTERFACE_ID, STARPU_BLOCK_INTERFACE_ID, STARPU_VECTOR_INTERFACE_ID,
  STARPU_CSR_INTERFACE_ID, STARPU_BCSR_INTERFACE_ID, STARPU_VARIABLE_INTERFACE_ID, STARPU_VOID_INTERFACE_ID,
  STARPU_MULTIFORMAT_INTERFACE_ID, STARPU_COO_INTERFACE_ID, STARPU_MAX_INTERFACE_ID }

Registering Data

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

- void starpu_void_data_register (starpu_data_handle_t *handle)
- void starpu_variable_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, size_t size)
- void starpu_vector_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t nx, size_t elemsize)
- void starpu_vector_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev_handle, size_t offset)
- void starpu_matrix_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t ld, uint32_t nx, uint32_t ny, size_t elemsize)
- void starpu_matrix_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev_handle, size_t offset, uint32_t ld)
- void starpu_block_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t ldy, uint32_t ldz, uint32_t nx, uint32_t ny, uint32_t nz, size_t elemsize)
- void starpu_block_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev_handle, size_t offset, uint32_t ldy, uint32_t ldz)
- void starpu_bcsr_data_register (starpu_data_handle_t *handle, unsigned home_node, uint32_t nnz, uint32_t nrow, uintptr_t nzval, uint32_t *colind, uint32_t *rowptr, uint32_t firstentry, uint32_t r, uint32_t c, size_t elemsize)
- void starpu_csr_data_register (starpu_data_handle_t *handle, unsigned home_node, uint32_t nnz, uint32_t nrow, uintptr_t nzval, uint32_t *colind, uint32_t *rowptr, uint32_t firstentry, size_t elemsize)
- void starpu_coo_data_register (starpu_data_handle_t *handleptr, unsigned home_node, uint32_t nx, uint32_t ny, uint32_t n_values, uint32_t *columns, uint32_t *rows, uintptr_t values, size_t elemsize)
- void * starpu_data_get_interface_on_node (starpu_data_handle_t handle, unsigned memory_node)

Accessing Data Interfaces

Each data interface is provided with a set of field access functions. The ones using a void * parameter aimed to be used in codelet implementations (see for example the code in Vector Scaling Using StarPU's API).

- void * starpu_data_handle_to_pointer (starpu_data_handle_t handle, unsigned node)

- void * starpu_data_get_local_ptr (starpu_data_handle_t handle)
- enum starpu_data_interface_id starpu_data_get_interface_id (starpu_data_handle_t handle)
- size_t starpu_data_get_size (starpu_data_handle_t handle)
- int starpu_data_pack (starpu_data_handle_t handle, void **ptr, starpu_ssize_t *count)
- int starpu_data_unpack (starpu_data_handle_t handle, void *ptr, size_t count)

Accessing Variable Data Interfaces

- #define STARPU_VARIABLE_GET_PTR(interface)
- #define STARPU_VARIABLE_GET_ELEMSIZE(interface)
- #define STARPU_VARIABLE_GET_DEV_HANDLE(interface)
- #define STARPU_VARIABLE_GET_OFFSET
- size_t starpu_variable_get_elemsize (starpu_data_handle_t handle)
- uintptr_t starpu_variable_get_local_ptr (starpu_data_handle_t handle)

Accessing Vector Data Interfaces

- #define STARPU_VECTOR_GET_PTR(interface)
- #define STARPU_VECTOR_GET_DEV_HANDLE(interface)
- #define STARPU_VECTOR_GET_OFFSET(interface)
- #define STARPU_VECTOR_GET_NX(interface)
- #define STARPU_VECTOR_GET_ELEMSIZE(interface)
- uint32_t starpu_vector_get_nx (starpu_data_handle_t handle)
- size_t starpu_vector_get_elemsize (starpu_data_handle_t handle)
- uintptr_t starpu_vector_get_local_ptr (starpu_data_handle_t handle)

Accessing Matrix Data Interfaces

- #define STARPU_MATRIX_GET_PTR(interface)
- #define STARPU_MATRIX_GET_DEV_HANDLE(interface)
- #define STARPU_MATRIX_GET_OFFSET(interface)
- #define STARPU_MATRIX_GET_NX(interface)
- #define STARPU_MATRIX_GET_NY(interface)
- #define STARPU_MATRIX_GET_LD(interface)
- #define STARPU_MATRIX_GET_ELEMSIZE(interface)
- uint32_t starpu_matrix_get_nx (starpu_data_handle_t handle)
- uint32_t starpu_matrix_get_ny (starpu_data_handle_t handle)
- uint32_t starpu_matrix_get_local_ld (starpu_data_handle_t handle)
- uintptr_t starpu_matrix_get_local_ptr (starpu_data_handle_t handle)
- size_t starpu_matrix_get_elemsize (starpu_data_handle_t handle)

Accessing Block Data Interfaces

- #define STARPU_BLOCK_GET_PTR(interface)
- #define STARPU_BLOCK_GET_DEV_HANDLE(interface)
- #define STARPU_BLOCK_GET_OFFSET(interface)
- #define STARPU_BLOCK_GET_NX(interface)
- #define STARPU_BLOCK_GET_NY(interface)
- #define STARPU_BLOCK_GET_NZ(interface)
- #define STARPU_BLOCK_GET_LDY(interface)
- #define STARPU_BLOCK_GET_LDZ(interface)
- #define STARPU_BLOCK_GET_ELEMSIZE(interface)
- uint32_t starpu_block_get_nx (starpu_data_handle_t handle)

- uint32_t starpu_block_get_ny (starpu_data_handle_t handle)
- uint32_t starpu_block_get_nz (starpu_data_handle_t handle)
- uint32_t starpu_block_get_local_ldy (starpu_data_handle_t handle)
- uint32_t starpu_block_get_local_ldz (starpu_data_handle_t handle)
- uintptr_t starpu_block_get_local_ptr (starpu_data_handle_t handle)
- size_t starpu_block_get_elemsize (starpu_data_handle_t handle)

Accessing BCSR Data Interfaces

- #define STARPU_BCSR_GET_NNZ(interface)
- #define STARPU_BCSR_GET_NZVAL(interface)
- #define STARPU_BCSR_GET_NZVAL_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_COLIND(interface)
- #define STARPU_BCSR_GET_COLIND_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_ROWPTR(interface)
- #define STARPU_BCSR_GET_ROWPTR_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_OFFSET
- uint32_t starpu_bcsr_get_nnz (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_nrow (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_firstentry (starpu_data_handle_t handle)
- uintptr_t starpu_bcsr_get_local_nzval (starpu_data_handle_t handle)
- uint32_t * starpu_bcsr_get_local_colind (starpu_data_handle_t handle)
- uint32_t * starpu_bcsr_get_local_rowptr (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_r (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_c (starpu_data_handle_t handle)
- size_t starpu_bcsr_get_elemsize (starpu_data_handle_t handle)

Accessing CSR Data Interfaces

- #define STARPU_CSR_GET_NNZ(interface)
- #define STARPU_CSR_GET_NROW(interface)
- #define STARPU_CSR_GET_NZVAL(interface)
- #define STARPU_CSR_GET_NZVAL_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_COLIND(interface)
- #define STARPU_CSR_GET_COLIND_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_ROWPTR(interface)
- #define STARPU_CSR_GET_ROWPTR_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_OFFSET
- #define STARPU_CSR_GET_FIRSTENTRY(interface)
- #define STARPU_CSR_GET_ELEMSIZE(interface)
- uint32_t starpu_csr_get_nnz (starpu_data_handle_t handle)
- uint32_t starpu_csr_get_nrow (starpu_data_handle_t handle)
- uint32_t starpu_csr_get_firstentry (starpu_data_handle_t handle)
- uintptr_t starpu_csr_get_local_nzval (starpu_data_handle_t handle)
- uint32_t * starpu_csr_get_local_colind (starpu_data_handle_t handle)
- uint32_t * starpu_csr_get_local_rowptr (starpu_data_handle_t handle)
- size_t starpu_csr_get_elemsize (starpu_data_handle_t handle)

Accessing COO Data Interfaces

- #define STARPU_COO_GET_COLUMNS(interface)
- #define STARPU_COO_GET_COLUMNS_DEV_HANDLE(interface)
- #define STARPU_COO_GET_ROWS(interface)
- #define STARPU_COO_GET_ROWS_DEV_HANDLE(interface)
- #define STARPU_COO_GET_VALUES(interface)
- #define STARPU_COO_GET_VALUES_DEV_HANDLE(interface)
- #define STARPU_COO_GET_OFFSET
- #define STARPU_COO_GET_NX(interface)
- #define STARPU_COO_GET_NY(interface)
- #define STARPU_COO_GET_NVALUES(interface)
- #define STARPU_COO_GET_ELEMSIZE(interface)

Defining Interface

Applications can provide their own interface as shown in Defining A New Data Interface.

- uintptr_t starpu_malloc_on_node (unsigned dst_node, size_t size)
- void starpu_free_on_node (unsigned dst_node, uintptr_t addr, size_t size)
- int starpu_interface_copy (uintptr_t src, size_t src_offset, unsigned src_node, uintptr_t dst, size_t dst_offset, unsigned dst_node, size_t size, void *async_data)
- uint32_t starpu_hash_crc32c_be_n (const void *input, size_t n, uint32_t inputcrc)
- uint32_t starpu_hash_crc32c_be (uint32_t input, uint32_t inputcrc)
- uint32_t starpu_hash_crc32c_string (const char *str, uint32_t inputcrc)
- int starpu_data_interface_get_next_id (void)

17.8.1 Detailed Description

17.8.2 Data Structure Documentation

17.8.2.1 struct starpu_data_interface_ops

Per-interface data transfer methods.

Data Fields

- void(* register_data_handle)(starpu_data_handle_t handle, unsigned home_node, void *data_interface)
- starpu_ssize_t(* allocate_data_on_node)(void *data_interface, unsigned node)
- void(* free_data_on_node)(void *data_interface, unsigned node)
- const struct starpu_data_copy_methods * copy_methods
- void *(* handle_to_pointer)(starpu_data_handle_t handle, unsigned node)
- size_t(* get_size)(starpu_data_handle_t handle)
- uint32_t(* footprint)(starpu_data_handle_t handle)
- int(* compare)(void *data_interface_a, void *data_interface_b)
- void(* display)(starpu_data_handle_t handle, FILE *f)
- enum starpu_data_interface_id interfaceid
- size_t interface_size
- int is_multiformat
- struct starpu_multiformat_data_interface_ops *(* get_mf_ops)(void *data_interface)
- int(* pack_data)(starpu_data_handle_t handle, unsigned node, void **ptr, starpu_ssize_t *count)
- int(* unpack_data)(starpu_data_handle_t handle, unsigned node, void *ptr, size_t count)

17.8.2.1.1 Field Documentation

17.8.2.1.1.1 starpu_data_interface_ops::register_data_handle

Register an existing interface into a data handle.

17.8.2.1.1.2 starpu_data_interface_ops::allocate_data_on_node

Allocate data for the interface on a given node.

17.8.2.1.1.3 starpu_data_interface_ops::free_data_on_node

Free data of the interface on a given node.

17.8.2.1.1.4 starpu_data_interface_ops::copy_methods

ram/cuda/opencl synchronous and asynchronous transfer methods.

17.8.2.1.1.5 starpu_data_interface_ops::handle_to_pointer

Return the current pointer (if any) for the handle on the given node.

17.8.2.1.1.6 starpu_data_interface_ops::get_size

Return an estimation of the size of data, for performance models.

17.8.2.1.1.7 starpu_data_interface_ops::footprint

Return a 32bit footprint which characterizes the data size.

17.8.2.1.1.8 starpu_data_interface_ops::compare

Compare the data size of two interfaces.

17.8.2.1.1.9 starpu_data_interface_ops::display

Dump the sizes of a handle to a file.

17.8.2.1.1.10 starpu_data_interface_ops::interfaceid

An identifier that is unique to each interface.

17.8.2.1.1.11 starpu_data_interface_ops::interface_size

The size of the interface data descriptor.

17.8.2.1.1.12 starpu_data_interface_ops::is_multiformat

todo

17.8.2.1.1.13 starpu_data_interface_ops::get_mf_ops

todo

17.8.2.1.1.14 starpu_data_interface_ops::pack_data

Pack the data handle into a contiguous buffer at the address ptr and set the size of the newly created buffer in count. If ptr is NULL, the function should not copy the data in the buffer but just set count to the size of the buffer which would have been allocated. The special value -1 indicates the size is yet unknown.

17.8.2.1.1.15 starpu_data_interface_ops::unpack_data

Unpack the data handle from the contiguous buffer at the address ptr of size count

17.8.2.2 struct starpu_data_copy_methods

Defines the per-interface methods. If the any_to_any method is provided, it will be used by default if no more specific method is provided. It can still be useful to provide more specific method in case of e.g. available particular CUDA or OpenCL support.

Data Fields

- int(∗ can_copy)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ ram_to_ram)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ ram_to_cuda)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ ram_to_opencl)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ cuda_to_ram)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ cuda_to_cuda)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ cuda_to_opencl)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ opencl_to_ram)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ opencl_to_cuda)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ opencl_to_opencl)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node)
- int(∗ ram_to_cuda_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node, cudaStream_t stream)
- int(∗ cuda_to_ram_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node, cudaStream_t stream)
- int(∗ cuda_to_cuda_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node, cudaStream_t stream)
- int(∗ ram_to_opencl_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_↩ node, cl_event ∗event)
- int(∗ opencl_to_ram_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_↩ node, cl_event ∗event)
- int(∗ opencl_to_opencl_async)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst↩ _node, cl_event ∗event)
- int(∗ any_to_any)(void ∗src_interface, unsigned src_node, void ∗dst_interface, unsigned dst_node, void ∗async_data)

17.8.2.2.1 Field Documentation

17.8.2.2.1.1 starpu_data_copy_methods::can_copy

If defined, allows the interface to declare whether it supports transferring from `src_interface` on node `src↩ _node` to `dst_interface` on node . If not defined, it is assumed that the interface supports all transfers.

17.8.2.2.1.2 starpu_data_copy_methods::ram_to_ram

Define how to copy data from the src_interface interface on the src_node CPU node to the dst_interface interface on the dst_node CPU node. Return 0 on success.

17.8.2.2.1.3 starpu_data_copy_methods::ram_to_cuda

Define how to copy data from the src_interface interface on the src_node CPU node to the dst_interface interface on the dst_node CUDA node. Return 0 on success.

17.8.2.2.1.4 starpu_data_copy_methods::ram_to_opencl

Define how to copy data from the src_interface interface on the src_node CPU node to the dst_interface interface on the dst_node OpenCL node. Return 0 on success.

17.8.2.2.1.5 starpu_data_copy_methods::cuda_to_ram

Define how to copy data from the src_interface interface on the src_node CUDA node to the dst_interface interface on the dst_node CPU node. Return 0 on success.

17.8.2.2.1.6 starpu_data_copy_methods::cuda_to_cuda

Define how to copy data from the src_interface interface on the src_node CUDA node to the dst_interface interface on the dst_node CUDA node. Return 0 on success.

17.8.2.2.1.7 starpu_data_copy_methods::cuda_to_opencl

Define how to copy data from the src_interface interface on the src_node CUDA node to the dst_interface interface on the dst_node OpenCL node. Return 0 on success.

17.8.2.2.1.8 starpu_data_copy_methods::opencl_to_ram

Define how to copy data from the src_interface interface on the src_node OpenCL node to the dst_interface interface on the dst_node CPU node. Return 0 on success.

17.8.2.2.1.9 starpu_data_copy_methods::opencl_to_cuda

Define how to copy data from the src_interface interface on the src_node OpenCL node to the dst_interface interface on the dst_node CUDA node. Return 0 on success.

17.8.2.2.1.10 starpu_data_copy_methods::opencl_to_opencl

Define how to copy data from the src_interface interface on the src_node OpenCL node to the dst_interface interface on the dst_node OpenCL node. Return 0 on success.

17.8.2.2.1.11 starpu_data_copy_methods::ram_to_cuda_async

Define how to copy data from the src_interface interface on the src_node CPU node to the dst_interface interface on the dst_node CUDA node, using the given stream. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.12 starpu_data_copy_methods::cuda_to_ram_async

Define how to copy data from the src_interface interface on the src_node CUDA node to the dst_interface interface on the dst_node CPU node, using the given stream. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.13 starpu_data_copy_methods::cuda_to_cuda_async

Define how to copy data from the src_interface interface on the src_node CUDA node to the dst_interface interface on the dst_node CUDA node, using the given stream. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.14 starpu_data_copy_methods::ram_to_opencl_async

Define how to copy data from the src_interface interface on the src_node CPU node to the dst_interface interface on the dst_node OpenCL node, by recording in event, a pointer to a cl_event, the event of the last submitted transfer. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.15 starpu_data_copy_methods::opencl_to_ram_async

Define how to copy data from the src_interface interface on the src_node OpenCL node to the dst_interface interface on the dst_node CPU node, by recording in event, a pointer to a cl_event, the event of the last submitted transfer. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.16 starpu_data_copy_methods::opencl_to_opencl_async

Define how to copy data from the src_interface interface on the src_node OpenCL node to the dst_interface interface on the dst_node OpenCL node, by recording in event, a pointer to a cl_event, the event of the last submitted transfer. Must return 0 if the transfer was actually completed completely synchronously, or -EAGAIN if at least some transfers are still ongoing and should be awaited for by the core.

17.8.2.2.1.17 starpu_data_copy_methods::any_to_any

Define how to copy data from the src_interface interface on the src_node node to the dst_interface interface on the dst_node node. This is meant to be implemented through the starpu_interface_copy() helper, to which async_↩ data should be passed as such, and will be used to manage asynchronicity. This must return -EAGAIN if any of the starpu_interface_copy() calls has returned -EAGAIN (i.e. at least some transfer is still ongoing), and return 0 otherwise.

17.8.2.3 struct starpu_variable_interface

Variable interface for a single data (not a vector, a matrix, a list, ...)

Data Fields

uintptr_t	ptr	local pointer of the variable
size_t	elemsize	size of the variable

17.8.2.4 struct starpu_vector_interface

Vector interface

vector interface for contiguous (non-strided) buffers

Data Fields

uintptr_t	ptr	local pointer of the vector
uintptr_t	dev_handle	device handle of the vector.
size_t	offset	offset in the vector
uint32_t	nx	number of elements on the x-axis of the vector
size_t	elemsize	size of the elements of the vector

17.8.2.5 struct starpu_matrix_interface

Matrix interface for dense matrices

Data Fields

uintptr_t	ptr	local pointer of the matrix
uintptr_t	dev_handle	device handle of the matrix.
size_t	offset	offset in the matrix

uint32_t	nx	number of elements on the x-axis of the matrix
uint32_t	ny	number of elements on the y-axis of the matrix
uint32_t	ld	number of elements between each row of the matrix. Maybe be equal to starpu_matrix_interface::nx when there is no padding.
size_t	elemsize	size of the elements of the matrix

17.8.2.6 struct starpu_block_interface

Block interface for 3D dense blocks

Data Fields

uintptr_t	ptr	local pointer of the block
uintptr_t	dev_handle	device handle of the block.
size_t	offset	offset in the block.
uint32_t	nx	number of elements on the x-axis of the block.
uint32_t	ny	number of elements on the y-axis of the block.
uint32_t	nz	number of elements on the z-axis of the block.
uint32_t	ldy	number of elements between two lines
uint32_t	ldz	number of elements between two planes
size_t	elemsize	size of the elements of the block.

17.8.2.7 struct starpu_bcsr_interface

BCSR interface for sparse matrices (blocked compressed sparse row representation)

Data Fields

uint32_t	nnz	number of non-zero BLOCKS
uint32_t	nrow	number of rows (in terms of BLOCKS)
uintptr_t	nzval	non-zero values
uint32_t *	colind	position of non-zero entried on the row
uint32_t *	rowptr	index (in nzval) of the first entry of the row
uint32_t	firstentry	k for k-based indexing (0 or 1 usually). Also useful when partitionning the matrix.
uint32_t	r	size of the blocks
uint32_t	c	size of the blocks
size_t	elemsize	size of the elements of the matrix

17.8.2.8 struct starpu_csr_interface

CSR interface for sparse matrices (compressed sparse row representation)

Data Fields

uint32_t	nnz	number of non-zero entries
uint32_t	nrow	number of rows
uintptr_t	nzval	non-zero values
uint32_t *	colind	position of non-zero entries on the row
uint32_t *	rowptr	index (in nzval) of the first entry of the row
uint32_t	firstentry	k for k-based indexing (0 or 1 usually). also useful when partitionning the matrix.

size_t	elemsize	size of the elements of the matrix

17.8.2.9 struct starpu_coo_interface

COO Matrices

Data Fields

uint32_t *	columns	column array of the matrix
uint32_t *	rows	row array of the matrix
uintptr_t	values	values of the matrix
uint32_t	nx	number of elements on the x-axis of the matrix
uint32_t	ny	number of elements on the y-axis of the matrix
uint32_t	n_values	number of values registered in the matrix
size_t	elemsize	size of the elements of the matrix

17.8.3 Macro Definition Documentation

17.8.3.1 #define STARPU_VARIABLE_GET_PTR(*interface*)

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

17.8.3.2 #define STARPU_VARIABLE_GET_ELEMSIZE(*interface*)

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

17.8.3.3 #define STARPU_VARIABLE_GET_DEV_HANDLE(*interface*)

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

17.8.3.4 #define STARPU_VARIABLE_GET_OFFSET

Return the offset in the variable designated by `interface`, to be used with the device handle.

17.8.3.5 #define STARPU_VECTOR_GET_PTR(*interface*)

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

17.8.3.6 #define STARPU_VECTOR_GET_DEV_HANDLE(*interface*)

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

17.8.3.7 #define STARPU_VECTOR_GET_OFFSET(*interface*)

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

17.8.3.8 #define STARPU_VECTOR_GET_NX(*interface*)

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

17.8.3.9 #define STARPU_VECTOR_GET_ELEMSIZE(*interface*)

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

17.8.3.10 #define STARPU_MATRIX_GET_PTR(*interface*)

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

17.8.3.11 #define STARPU_MATRIX_GET_DEV_HANDLE(*interface*)

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

17.8.3.12 #define STARPU_MATRIX_GET_OFFSET(*interface*)

Return the offset in the matrix designated by `interface`, to be used with the device handle.

17.8.3.13 #define STARPU_MATRIX_GET_NX(*interface*)

Return the number of elements on the x-axis of the matrix designated by `interface`.

17.8.3.14 #define STARPU_MATRIX_GET_NY(*interface*)

Return the number of elements on the y-axis of the matrix designated by `interface`.

17.8.3.15 #define STARPU_MATRIX_GET_LD(*interface*)

Return the number of elements between each row of the matrix designated by `interface`. May be equal to nx when there is no padding.

17.8.3.16 #define STARPU_MATRIX_GET_ELEMSIZE(*interface*)

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

17.8.3.17 #define STARPU_BLOCK_GET_PTR(*interface*)

Return a pointer to the block designated by `interface`.

17.8.3.18 #define STARPU_BLOCK_GET_DEV_HANDLE(*interface*)

Return a device handle for the block designated by `interface`, to be used on OpenCL. The offset document below has to be used in addition to this.

17.8.3.19 #define STARPU_BLOCK_GET_OFFSET(*interface*)

Return the offset in the block designated by `interface`, to be used with the device handle.

17.8.3.20 #define STARPU_BLOCK_GET_NX(*interface*)

Return the number of elements on the x-axis of the block designated by `interface`.

17.8.3.21 #define STARPU_BLOCK_GET_NY(*interface*)

Return the number of elements on the y-axis of the block designated by `interface`.

17.8.3.22 #define STARPU_BLOCK_GET_NZ(*interface*)

Return the number of elements on the z-axis of the block designated by `interface`.

17.8.3.23 #define STARPU_BLOCK_GET_LDY(*interface*)

Return the number of elements between each row of the block designated by `interface`. May be equal to nx when there is no padding.

17.8.3.24 #define STARPU_BLOCK_GET_LDZ(*interface*)

Return the number of elements between each z plane of the block designated by `interface`. May be equal to nx∗ny when there is no padding.

17.8.3.25 #define STARPU_BLOCK_GET_ELEMSIZE(*interface*)

Return the size of the elements of the block designated by `interface`.

17.8.3.26 #define STARPU_BCSR_GET_NNZ(*interface*)

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

17.8.3.27 #define STARPU_BCSR_GET_NZVAL(*interface*)

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

17.8.3.28 #define STARPU_BCSR_GET_NZVAL_DEV_HANDLE(*interface*)

Return a device handle for the array of non-zero values in the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.29 #define STARPU_BCSR_GET_COLIND(*interface*)

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

17.8.3.30 #define STARPU_BCSR_GET_COLIND_DEV_HANDLE(*interface*)

Return a device handle for the column index of the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.31 #define STARPU_BCSR_GET_ROWPTR(*interface*)

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

17.8.3.32 #define STARPU_BCSR_GET_ROWPTR_DEV_HANDLE(*interface*)

Return a device handle for the row pointer array of the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.33 #define STARPU_BCSR_GET_OFFSET

Return the offset in the arrays (coling, rowptr, nzval) of the matrix designated by `interface`, to be used with the device handles.

17.8.3.34 #define STARPU_CSR_GET_NNZ(*interface*)

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

17.8.3.35 #define STARPU_CSR_GET_NROW(*interface*)

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

17.8.3.36 #define STARPU_CSR_GET_NZVAL(*interface*)

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

17.8.3.37 #define STARPU_CSR_GET_NZVAL_DEV_HANDLE(*interface*)

Return a device handle for the array of non-zero values in the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.38 #define STARPU_CSR_GET_COLIND(*interface*)

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

17.8.3.39 #define STARPU_CSR_GET_COLIND_DEV_HANDLE(*interface*)

Return a device handle for the column index of the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.40 #define STARPU_CSR_GET_ROWPTR(*interface*)

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

17.8.3.41 #define STARPU_CSR_GET_ROWPTR_DEV_HANDLE(*interface*)

Return a device handle for the row pointer array of the matrix designated by `interface`. The offset documented below has to be used in addition to this.

17.8.3.42 #define STARPU_CSR_GET_OFFSET

Return the offset in the arrays (colind, rowptr, nzval) of the matrix designated by `interface`, to be used with the device handles.

17.8.3.43 #define STARPU_CSR_GET_FIRSTENTRY(*interface*)

Return the index at which all arrays (the column indexes, the row pointers...) of the `interface` start.

17.8.3.44 #define STARPU_CSR_GET_ELEMSIZE(*interface*)

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

17.8.3.45 #define STARPU_COO_GET_COLUMNS(*interface*)

Return a pointer to the column array of the matrix designated by `interface`.

17.8.3.46 #define STARPU_COO_GET_COLUMNS_DEV_HANDLE(*interface*)

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

17.8.3.47 #define STARPU_COO_GET_ROWS(*interface*)

Return a pointer to the rows array of the matrix designated by `interface`.

17.8.3.48 #define STARPU_COO_GET_ROWS_DEV_HANDLE(*interface*)

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

17.8.3.49 #define STARPU_COO_GET_VALUES(*interface*)

Return a pointer to the values array of the matrix designated by `interface`.

17.8.3.50 #define STARPU_COO_GET_VALUES_DEV_HANDLE(*interface*)

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

17.8.3.51 #define STARPU_COO_GET_OFFSET

Return the offset in the arrays of the COO matrix designated by `interface`.

17.8.3.52 #define STARPU_COO_GET_NX(*interface*)

Return the number of elements on the x-axis of the matrix designated by `interface`.

17.8.3.53 #define STARPU_COO_GET_NY(*interface*)

Return the number of elements on the y-axis of the matrix designated by `interface`.

17.8.3.54 #define STARPU_COO_GET_NVALUES(*interface*)

Return the number of values registered in the matrix designated by `interface`.

17.8.3.55 #define STARPU_COO_GET_ELEMSIZE(*interface*)

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

17.8.4 Enumeration Type Documentation

17.8.4.1 enum starpu_data_interface_id

Identifier for all predefined StarPU data interfaces

Enumerator

STARPU_UNKNOWN_INTERFACE_ID Unknown interface
STARPU_MATRIX_INTERFACE_ID Identifier for the matrix data interface
STARPU_BLOCK_INTERFACE_ID Identifier for block data interface
STARPU_VECTOR_INTERFACE_ID Identifier for the vector data interface
STARPU_CSR_INTERFACE_ID Identifier for the csr data interface
STARPU_BCSR_INTERFACE_ID Identifier for the bcsr data interface
STARPU_VARIABLE_INTERFACE_ID Identifier for the variable data interface
STARPU_VOID_INTERFACE_ID Identifier for the void data interface
STARPU_MULTIFORMAT_INTERFACE_ID Identifier for the multiformat data interface
STARPU_COO_INTERFACE_ID Identifier for the coo data interface
STARPU_MAX_INTERFACE_ID Maximum number of data interfaces

17.8.5 Function Documentation

17.8.5.1 void starpu_void_data_register (starpu_data_handle_t * *handle*)

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

17.8.5.2 void starpu_variable_data_register (starpu_data_handle_t * *handle,* unsigned *home_node,* uintptr_t *ptr,* size_t *size*)

Register the `size` byte element pointed to by `ptr`, which is typically a scalar, and initialize `handle` to represent this data item.

Here an example of how to use the function.

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

17.8.5.3 void starpu_vector_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, uintptr_t *ptr*, uint32_t *nx*, size_t *elemsize*)

Register the `nx` elemsize-byte elements pointed to by `ptr` and initialize `handle` to represent it.

Here an example of how to use the function.

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

17.8.5.4 void starpu_vector_ptr_register (starpu_data_handle_t *handle*, unsigned *node*, uintptr_t *ptr*, uintptr_t *dev_handle*, size_t *offset*)

Register into the `handle` that to store data on node `node` it should use the buffer located at `ptr`, or device handle `dev_handle` and offset `offset` (for OpenCL, notably)

17.8.5.5 void starpu_matrix_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, uintptr_t *ptr*, uint32_t *ld*, uint32_t *nx*, uint32_t *ny*, size_t *elemsize*)

Register the `nx` x `ny` 2D matrix of `elemsize-byte` elements pointed by `ptr` and initialize `handle` to represent it. `ld` specifies the number of elements between rows. a value greater than `nx` adds padding, which can be useful for alignment purposes.

Here an example of how to use the function.

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

17.8.5.6 void starpu_matrix_ptr_register (starpu_data_handle_t *handle*, unsigned *node*, uintptr_t *ptr*, uintptr_t *dev_handle*, size_t *offset*, uint32_t *ld*)

Register into the `handle` that to store data on node `node` it should use the buffer located at `ptr`, or device handle `dev_handle` and offset `offset` (for OpenCL, notably), with `ld` elements between rows.

17.8.5.7 void starpu_block_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, uintptr_t *ptr*, uint32_t *ldy*, uint32_t *ldz*, uint32_t *nx*, uint32_t *ny*, uint32_t *nz*, size_t *elemsize*)

Register the `nx` x `ny` x `nz` 3D matrix of `elemsize` byte elements pointed by `ptr` and initialize `handle` to represent it. Again, `ldy` and `ldz` specify the number of elements between rows and between z planes.

Here an example of how to use the function.

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

17.8.5.8 void starpu_block_ptr_register (starpu_data_handle_t *handle*, unsigned *node*, uintptr_t *ptr*, uintptr_t *dev_handle*, size_t *offset*, uint32_t *ldy*, uint32_t *ldz*)

Register into the `handle` that to store data on node `node` it should use the buffer located at `ptr`, or device handle `dev_handle` and offset `offset` (for OpenCL, notably), with `ldy` elements between rows and elements between z planes.

17.8.5.9 void starpu_bcsr_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, uint32_t *nnz*, uint32_t *nrow*, uintptr_t *nzval*, uint32_t * *colind*, uint32_t * *rowptr*, uint32_t *firstentry*, uint32_t *r*, uint32_t *c*, size_t *elemsize*)

This variant of starpu_data_register() uses the BCSR (Blocked Compressed Sparse Row Representation) sparse matrix interface. Register the sparse matrix made of `nnz` non-zero blocks of elements of size `elemsize` stored in `nzval` and initializes `handle` to represent it. Blocks have size `r` * `c`. `nrow` is the number of rows (in terms of blocks), `colind`[i] is the block-column index for block i in `nzval`, `rowptr`[i] is the block-index (in `nzval`) of the first block of row i. `firstentry` is the index of the first entry of the given arrays (usually 0 or 1).

17.8.5.10 void starpu_csr_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, uint32_t *nnz*, uint32_t *nrow*, uintptr_t *nzval*, uint32_t * *colind*, uint32_t * *rowptr*, uint32_t *firstentry*, size_t *elemsize*)

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

17.8.5.11 void starpu_coo_data_register (starpu_data_handle_t * *handleptr*, unsigned *home_node*, uint32_t *nx*, uint32_t *ny*, uint32_t *n_values*, uint32_t * *columns*, uint32_t * *rows*, uintptr_t *values*, size_t *elemsize*)

Register the `nx` x `ny` 2D matrix given in the COO format, using the `columns`, `rows`, `values` arrays, which must have `n_values` elements of size `elemsize`. Initialize `handleptr`.

17.8.5.12 void * starpu_data_get_interface_on_node (starpu_data_handle_t *handle*, unsigned *memory_node*)

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

17.8.5.13 void * starpu_data_handle_to_pointer (starpu_data_handle_t *handle*, unsigned *node*)

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

17.8.5.14 void * starpu_data_get_local_ptr (starpu_data_handle_t *handle*)

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

17.8.5.15 enum starpu_data_interface_id starpu_data_get_interface_id (starpu_data_handle_t *handle*)

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

17.8.5.16 size_t starpu_data_get_size (starpu_data_handle_t *handle*)

Return the size of the data associated with `handle`.

17.8.5.17 int starpu_data_pack (starpu_data_handle_t *handle*, void ** *ptr*, starpu_ssize_t * *count*)

Execute the packing operation of the interface of the data registered at `handle` (see starpu_data_interface_ops). This packing operation must allocate a buffer large enough at `ptr` and copy into the newly allocated buffer the data associated to `handle`. `count` will be set to the size of the allocated buffer. If `ptr` is NULL, the function should not copy the data in the buffer but just set `count` to the size of the buffer which would have been allocated. The special value -1 indicates the size is yet unknown.

17.8.5.18 int starpu_data_unpack (starpu_data_handle_t *handle*, void * *ptr*, size_t *count*)

Unpack in handle the data located at `ptr` of size `count` as described by the interface of the data. The interface registered at `handle` must define a unpacking operation (see starpu_data_interface_ops). The memory at the address `ptr` is freed after calling the data unpacking operation.

17.8.5.19 size_t starpu_variable_get_elemsize (starpu_data_handle_t *handle*)

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

17.8.5.20 uintptr_t starpu_variable_get_local_ptr (starpu_data_handle_t *handle*)

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

17.8.5.21 uint32_t starpu_vector_get_nx (starpu_data_handle_t *handle*)

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

17.8.5.22 size_t starpu_vector_get_elemsize (starpu_data_handle_t *handle*)

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

17.8.5.23 uintptr_t starpu_vector_get_local_ptr (starpu_data_handle_t *handle*)

Return the local pointer associated with `handle`.

17.8.5.24 uint32_t starpu_matrix_get_nx (starpu_data_handle_t *handle*)

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

17.8.5.25 uint32_t starpu_matrix_get_ny (starpu_data_handle_t *handle*)

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

17.8.5.26 uint32_t starpu_matrix_get_local_ld (starpu_data_handle_t *handle*)

Return the number of elements between each row of the matrix designated by `handle`. Maybe be equal to nx when there is no padding.

17.8.5.27 uintptr_t starpu_matrix_get_local_ptr (starpu_data_handle_t *handle*)

Return the local pointer associated with `handle`.

17.8.5.28 size_t starpu_matrix_get_elemsize (starpu_data_handle_t *handle*)

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

17.8.5.29 uint32_t starpu_block_get_nx (starpu_data_handle_t *handle*)

Return the number of elements on the x-axis of the block designated by `handle`.

17.8.5.30 uint32_t starpu_block_get_ny (starpu_data_handle_t *handle*)

Return the number of elements on the y-axis of the block designated by `handle`.

17.8.5.31 uint32_t starpu_block_get_nz (starpu_data_handle_t *handle*)

Return the number of elements on the z-axis of the block designated by `handle`.

17.8.5.32 uint32_t starpu_block_get_local_ldy (starpu_data_handle_t *handle*)

Return the number of elements between each row of the block designated by `handle`, in the format of the current memory node.

17.8.5.33 uint32_t starpu_block_get_local_ldz (starpu_data_handle_t *handle*)

Return the number of elements between each z plane of the block designated by `handle`, in the format of the current memory node.

17.8.5.34 uintptr_t starpu_block_get_local_ptr (starpu_data_handle_t *handle*)

Return the local pointer associated with `handle`.

17.8.5.35 size_t starpu_block_get_elemsize (starpu_data_handle_t *handle*)

Return the size of the elements of the block designated by `handle`.

17.8.5.36 uint32_t starpu_bcsr_get_nnz (starpu_data_handle_t *handle*)

Return the number of non-zero elements in the matrix designated by `handle`.

17.8.5.37 uint32_t starpu_bcsr_get_nrow (starpu_data_handle_t *handle*)

Return the number of rows (in terms of blocks of size r∗c) in the matrix designated by `handle`.

17.8.5.38 uint32_t starpu_bcsr_get_firstentry (starpu_data_handle_t *handle*)

Return the index at which all arrays (the column indexes, the row pointers...) of the matrix desginated by `handle`.

17.8.5.39 uintptr_t starpu_bcsr_get_local_nzval (starpu_data_handle_t *handle*)

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

17.8.5.40 uint32_t * starpu_bcsr_get_local_colind (starpu_data_handle_t *handle*)

Return a pointer to the column index, which holds the positions of the non-zero entries in the matrix designated by `handle`.

17.8.5.41 uint32_t * starpu_bcsr_get_local_rowptr (starpu_data_handle_t *handle*)

Return the row pointer array of the matrix designated by `handle`.

17.8.5.42 uint32_t starpu_bcsr_get_r (starpu_data_handle_t *handle*)

Return the number of rows in a block.

17.8.5.43 uint32_t starpu_bcsr_get_c (starpu_data_handle_t *handle*)

Return the numberof columns in a block.

17.8.5.44 size_t starpu_bcsr_get_elemsize (starpu_data_handle_t *handle*)

Return the size of the elements in the matrix designated by `handle`.

17.8.5.45 uint32_t starpu_csr_get_nnz (starpu_data_handle_t *handle*)

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

17.8.5.46 uint32_t starpu_csr_get_nrow (starpu_data_handle_t *handle*)

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

17.8.5.47 uint32_t starpu_csr_get_firstentry (starpu_data_handle_t *handle*)

Return the index at which all arrays (the column indexes, the row pointers...) of the matrix designated by `handle`.

17.8.5.48 uintptr_t starpu_csr_get_local_nzval (starpu_data_handle_t *handle*)

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

17.8.5.49 uint32_t * starpu_csr_get_local_colind (starpu_data_handle_t *handle*)

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

17.8.5.50 uint32_t * starpu_csr_get_local_rowptr (starpu_data_handle_t *handle*)

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

17.8.5.51 size_t starpu_csr_get_elemsize (starpu_data_handle_t *handle*)

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

17.8.5.52 uintptr_t starpu_malloc_on_node (unsigned *dst_node*, size_t *size*)

Allocate `size` bytes on node `dst_node`. This returns 0 if allocation failed, the allocation method should then return `-ENOMEM` as allocated size.

17.8.5.53 void starpu_free_on_node (unsigned *dst_node*, uintptr_t *addr*, size_t *size*)

Free `addr` of `size` bytes on node `dst_node`.

17.8.5.54 int starpu_interface_copy (uintptr_t *src*, size_t *src_offset*, unsigned *src_node*, uintptr_t *dst*, size_t *dst_offset*, unsigned *dst_node*, size_t *size*, void * *async_data*)

Copy `size` bytes from byte offset `src_offset` of `src` on `src_node` to byte offset `dst_offset` of `dst` on `dst_node`. This is to be used in the any_to_any() copy method, which is provided with the async_data to be passed to starpu_interface_copy(). this returns `-EAGAIN` if the transfer is still ongoing, or 0 if the transfer is already completed.

17.8.5.55 uint32_t starpu_hash_crc32c_be_n (const void * *input*, size_t *n*, uint32_t *inputcrc*)

Compute the CRC of a byte buffer seeded by the `inputcrc` *current state*. The return value should be considered as the new *current state* for future CRC computation. This is used for computing data size footprint.

17.8.5.56 uint32_t starpu_hash_crc32c_be (uint32_t *input*, uint32_t *inputcrc*)

Compute the CRC of a 32bit number seeded by the `inputcrc` *current state*. The return value should be considered as the new *current state* for future CRC computation. This is used for computing data size footprint.

17.8.5.57 uint32_t starpu_hash_crc32c_string (const char * *str*, uint32_t *inputcrc*)

Compute the CRC of a string seeded by the `inputcrc` *current state*. The return value should be considered as the new *current state* for future CRC computation. This is used for computing data size footprint.

17.8.5.58 int starpu_data_interface_get_next_id (void)

Return the next available id for a newly created data interface (Defining A New Data Interface).

17.9 Data Partition

Data Structures

- struct starpu_data_filter

Basic API

- void starpu_data_partition (starpu_data_handle_t initial_handle, struct starpu_data_filter *f)
- void starpu_data_unpartition (starpu_data_handle_t root_data, unsigned gathering_node)
- int starpu_data_get_nb_children (starpu_data_handle_t handle)
- starpu_data_handle_t starpu_data_get_child (starpu_data_handle_t handle, unsigned i)
- starpu_data_handle_t starpu_data_get_sub_data (starpu_data_handle_t root_data, unsigned depth,...)
- starpu_data_handle_t starpu_data_vget_sub_data (starpu_data_handle_t root_data, unsigned depth, va_list pa)

- void starpu_data_map_filters (starpu_data_handle_t root_data, unsigned nfilters,...)
- void starpu_data_vmap_filters (starpu_data_handle_t root_data, unsigned nfilters, va_list pa)

Predefined Vector Filter Functions

This section gives a partial list of the predefined partitioning functions for vector data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_vector_filter_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_list (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_divide_in_2 (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

Predefined Matrix Filter Functions

This section gives a partial list of the predefined partitioning functions for matrix data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_matrix_filter_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_vertical_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_vertical_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

Predefined Block Filter Functions

This section gives a partial list of the predefined partitioning functions for block data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`. A usage example is available in examples/filters/shadow3d.c

- void starpu_block_filter_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_vertical_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_vertical_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_depth_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_depth_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

Predefined BCSR Filter Functions

This section gives a partial list of the predefined partitioning functions for BCSR data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_bcsr_filter_canonical_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_csr_filter_vertical_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

17.9.1 Detailed Description

17.9.2 Data Structure Documentation

17.9.2.1 struct starpu_data_filter

The filter structure describes a data partitioning operation, to be given to the starpu_data_partition() function.

Data Fields

- void(* filter_func)(void *father_interface, void *child_interface, struct starpu_data_filter *, unsigned id, unsigned nparts)
- unsigned nchildren
- unsigned(* get_nchildren)(struct starpu_data_filter *, starpu_data_handle_t initial_handle)
- struct starpu_data_interface_ops *(* get_child_ops)(struct starpu_data_filter *, unsigned id)
- unsigned filter_arg
- void * filter_arg_ptr

17.9.2.1.1 Field Documentation

17.9.2.1.1.1 starpu_data_filter::filter_func

This function fills the child_interface structure with interface information for the id-th child of the parent father_interface (among nparts).

17.9.2.1.1.2 starpu_data_filter::nchildren

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

17.9.2.1.1.3 starpu_data_filter::get_nchildren

This returns the number of children. This can be used instead of nchildren when the number of children depends on the actual data (e.g. the number of blocks in a sparse matrix).

17.9.2.1.1.4 starpu_data_filter::get_child_ops

In case the resulting children use a different data interface, this function returns which interface is used by child number id.

17.9.2.1.1.5 starpu_data_filter::filter_arg

Allow to define an additional parameter for the filter function.

17.9.2.1.1.6 starpu_data_filter::filter_arg_ptr

Allow to define an additional pointer parameter for the filter function, such as the sizes of the different parts.

17.9.3 Function Documentation

17.9.3.1 void starpu_data_partition (starpu_data_handle_t *initial_handle*, struct starpu_data_filter * *f*)

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

Here an example of how to use the function.

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

17.9.3.2 void starpu_data_unpartition (starpu_data_handle_t *root_data*, unsigned *gathering_node*)

This unapplies one filter, thus unpartitioning the data. The pieces of data are collected back into one big piece in the `gathering_node` (usually 0). Tasks working on the partitioned data must be already finished when calling starpu_data_unpartition().

Here an example of how to use the function.

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

17.9.3.3 int starpu_data_get_nb_children (starpu_data_handle_t *handle*)

This function returns the number of children.

17.9.3.4 starpu_data_handle_t starpu_data_get_child (starpu_data_handle_t *handle*, unsigned *i*)

Return the ith child of the given `handle`, which must have been partitionned beforehand.

17.9.3.5 starpu_data_handle_t starpu_data_get_sub_data (starpu_data_handle_t *root_data*, unsigned *depth*, ...)

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

Here an example of how to use the function.

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

17.9.3.6 starpu_data_handle_t starpu_data_vget_sub_data (starpu_data_handle_t *root_data*, unsigned *depth*, va_list *pa*)

This function is similar to starpu_data_get_sub_data() but uses a va_list for the parameter list.

17.9.3.7 void starpu_data_map_filters (starpu_data_handle_t *root_data*, unsigned *nfilters*, ...)

Applies `nfilters` filters to the handle designated by `root_handle` recursively. `nfilters` pointers to variables of the type starpu_data_filter should be given.

17.9.3.8 void starpu_data_vmap_filters (starpu_data_handle_t *root_data*, unsigned *nfilters*, va_list *pa*)

Applies `nfilters` filters to the handle designated by `root_handle` recursively. It uses a va_list of pointers to variables of the type starpu_data_filter.

17.9.3.9 void starpu_vector_filter_block (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

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

17.9.3.10 void starpu_vector_filter_block_shadow (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

Return in `child_interface` the `id` th element of the vector represented by `father_interface` once partitioned in `nparts` chunks of equal size with a shadow border `filter_arg_ptr`, thus getting a vector of size (n-2*shadow)/nparts+2*shadow. The `filter_arg_ptr` field of `f` must be the shadow size casted into void*. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts. An usage example is available in examples/filters/shadow.c

17.9.3.11 void starpu_vector_filter_list (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

Return in `child_interface` the `id` th element of the vector represented by `father_interface` once partitioned into `nparts` chunks according to the `filter_arg_ptr` field of `f`. The `filter_arg_ptr` field must point to an array of `nparts` uint32_t elements, each of which specifies the number of elements in each chunk of the partition.

17.9.3.12 void starpu_vector_filter_divide_in_2 (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

Return in `child_interface` the `id` th element of the vector represented by `father_interface` once partitioned in `2` chunks of equal size, ignoring nparts. Thus, `id` must be `0` or `1`.

17.9.3.13 void starpu_matrix_filter_block (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

This partitions a dense Matrix along the x dimension, thus getting (x/`nparts` ,y) matrices. If `nparts` does not divide x, the last submatrix contains the remainder.

17.9.3.14 void starpu_matrix_filter_block_shadow (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

This partitions a dense Matrix along the x dimension, with a shadow border `filter_arg_ptr`, thus getting ((x-2*shadow)/`nparts` +2*shadow,y) matrices. If `nparts` does not divide x-2*shadow, the last submatrix contains the remainder. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts. A usage example is available in examples/filters/shadow2d.c

17.9.3.15 void starpu_matrix_filter_vertical_block (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

This partitions a dense Matrix along the y dimension, thus getting (x,y/`nparts`) matrices. If `nparts` does not divide y, the last submatrix contains the remainder.

17.9.3.16 void starpu_matrix_filter_vertical_block_shadow (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a dense Matrix along the y dimension, with a shadow border `filter_arg_ptr`, thus getting (x,(y-2*shadow)/`nparts` +2*shadow) matrices. If `nparts` does not divide y-2*shadow, the last submatrix contains the remainder. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts. A usage example is available in examples/filters/shadow2d.c

17.9.3.17 void starpu_block_filter_block (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the X dimension, thus getting (x/`nparts` ,y,z) 3D matrices. If `nparts` does not divide x, the last submatrix contains the remainder.

17.9.3.18 void starpu_block_filter_block_shadow (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the X dimension, with a shadow border `filter_arg_ptr`, thus getting ((x-2*shadow)/`nparts` +2*shadow,y,z) blocks. If `nparts` does not divide x, the last submatrix contains the remainder. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts.

17.9.3.19 void starpu_block_filter_vertical_block (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the Y dimension, thus getting (x,y/`nparts` ,z) blocks. If `nparts` does not divide y, the last submatrix contains the remainder.

17.9.3.20 void starpu_block_filter_vertical_block_shadow (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the Y dimension, with a shadow border `filter_arg_ptr`, thus getting (x,(y-2*shadow)/`nparts` +2*shadow,z) 3D matrices. If `nparts` does not divide y, the last submatrix contains the remainder. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts.

17.9.3.21 void starpu_block_filter_depth_block (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the Z dimension, thus getting (x,y,z/`nparts`) blocks. If `nparts` does not divide z, the last submatrix contains the remainder.

17.9.3.22 void starpu_block_filter_depth_block_shadow (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block along the Z dimension, with a shadow border `filter_arg_ptr`, thus getting (x,y,(z-2*shadow)/`nparts` +2*shadow) blocks. If `nparts` does not divide z, the last submatrix contains the remainder. **IMPORTANT**: This can only be used for read-only access, as no coherency is enforced for the shadowed parts.

17.9.3.23 void starpu_bcsr_filter_canonical_block (void * *father_interface,* void * *child_interface,* struct **starpu_data_filter** * *f,* unsigned *id,* unsigned *nparts*)

This partitions a block-sparse matrix into dense matrices.

17.9.3.24 void starpu_csr_filter_vertical_block (void * *father_interface*, void * *child_interface*, struct starpu_data_filter * *f*, unsigned *id*, unsigned *nparts*)

This partitions a block-sparse matrix into vertical block-sparse matrices.

17.10 Multiformat Data Interface

Data Structures

- struct starpu_multiformat_data_interface_ops
- struct starpu_multiformat_interface

Macros

- #define STARPU_MULTIFORMAT_GET_CPU_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_CUDA_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_OPENCL_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_NX(interface)

Functions

- void starpu_multiformat_data_register (starpu_data_handle_t *handle, unsigned home_node, void *ptr, uint32_t nobjects, struct starpu_multiformat_data_interface_ops *format_ops)

17.10.1 Detailed Description

17.10.2 Data Structure Documentation

17.10.2.1 struct starpu_multiformat_data_interface_ops

The different fields are:

Data Fields

size_t	cpu_elemsize	the size of each element on CPUs
size_t	opencl_elemsize	the size of each element on OpenCL devices
struct starpu_codelet *	cpu_to_opencl↩ _cl	pointer to a codelet which converts from CPU to OpenCL
struct starpu_codelet *	opencl_to_cpu↩ _cl	pointer to a codelet which converts from OpenCL to CPU
size_t	cuda_elemsize	the size of each element on CUDA devices
struct starpu_codelet *	cpu_to_cuda_cl	pointer to a codelet which converts from CPU to CUDA
struct starpu_codelet *	cuda_to_cpu_cl	pointer to a codelet which converts from CUDA to CPU

17.10.2.2 struct starpu_multiformat_interface

todo

Data Fields

void *	cpu_ptr	
void *	cuda_ptr	
void *	opencl_ptr	
uint32_t	nx	
struct starpu_multiformat_data_interface_ops *	ops	

17.10.3 Macro Definition Documentation

17.10.3.1 #define STARPU_MULTIFORMAT_GET_CPU_PTR(*interface*)

returns the local pointer to the data with CPU format.

17.10.3.2 #define STARPU_MULTIFORMAT_GET_CUDA_PTR(*interface*)

returns the local pointer to the data with CUDA format.

17.10.3.3 #define STARPU_MULTIFORMAT_GET_OPENCL_PTR(*interface*)

returns the local pointer to the data with OpenCL format.

17.10.3.4 #define STARPU_MULTIFORMAT_GET_NX(*interface*)

returns the number of elements in the data.

17.10.4 Function Documentation

17.10.4.1 void starpu_multiformat_data_register (starpu_data_handle_t * *handle*, unsigned *home_node*, void * *ptr*, uint32_t *nobjects*, struct starpu_multiformat_data_interface_ops * *format_ops*)

Register a piece of data that can be represented in different ways, depending upon the processing unit that manipulates it. It allows the programmer, for instance, to use an array of structures when working on a CPU, and a structure of arrays when working on a GPU. `nobjects` is the number of elements in the data. `format_ops` describes the format.

17.11 Codelet And Tasks

This section describes the interface to manipulate codelets and tasks.

Data Structures

- struct starpu_codelet
- struct starpu_data_descr
- struct starpu_task

Macros

- #define STARPU_CPU
- #define STARPU_CUDA
- #define STARPU_OPENCL
- #define STARPU_MULTIPLE_CPU_IMPLEMENTATIONS
- #define STARPU_MULTIPLE_CUDA_IMPLEMENTATIONS
- #define STARPU_MULTIPLE_OPENCL_IMPLEMENTATIONS
- #define STARPU_NMAXBUFS
- #define STARPU_TASK_INITIALIZER
- #define STARPU_TASK_GET_HANDLE(task, i)
- #define STARPU_TASK_SET_HANDLE(task, handle, i)
- #define STARPU_CODELET_GET_MODE(codelet, i)
- #define STARPU_CODELET_SET_MODE(codelet, mode, i)
- #define STARPU_TASK_INVALID

Typedefs

- typedef void(* starpu_cpu_func_t)(void **, void *)
- typedef void(* starpu_cuda_func_t)(void **, void *)
- typedef void(* starpu_opencl_func_t)(void **, void *)

Enumerations

- enum starpu_codelet_type { STARPU_SEQ, STARPU_SPMD, STARPU_FORKJOIN }
- enum starpu_task_status {
  STARPU_TASK_INVALID, **STARPU_TASK_INVALID**, STARPU_TASK_BLOCKED, STARPU_TASK_R↩
  EADY,
  STARPU_TASK_RUNNING, STARPU_TASK_FINISHED, STARPU_TASK_BLOCKED_ON_TAG, STARP↩
  U_TASK_BLOCKED_ON_TASK,
  STARPU_TASK_BLOCKED_ON_DATA }

Functions

- void starpu_codelet_init (struct starpu_codelet *cl)
- void starpu_task_init (struct starpu_task *task)
- struct starpu_task * starpu_task_create (void)
- struct starpu_task * starpu_task_dup (struct starpu_task *task)
- void starpu_task_clean (struct starpu_task *task)
- void starpu_task_destroy (struct starpu_task *task)
- int starpu_task_wait (struct starpu_task *task) STARPU_WARN_UNUSED_RESULT
- int starpu_task_submit (struct starpu_task *task) STARPU_WARN_UNUSED_RESULT
- int starpu_task_submit_to_ctx (struct starpu_task *task, unsigned sched_ctx_id)
- int starpu_task_wait_for_all (void)
- int starpu_task_wait_for_all_in_ctx (unsigned sched_ctx_id)
- int starpu_task_nready (void)
- int starpu_task_nsubmitted (void)
- struct starpu_task * starpu_task_get_current (void)
- void starpu_codelet_display_stats (struct starpu_codelet *cl)
- int starpu_task_wait_for_no_ready (void)
- void starpu_task_set_implementation (struct starpu_task *task, unsigned impl)
- unsigned starpu_task_get_implementation (struct starpu_task *task)
- void starpu_create_sync_task (starpu_tag_t sync_tag, unsigned ndeps, starpu_tag_t *deps, void(*callback)(void *), void *callback_arg)

17.11.1 Detailed Description

This section describes the interface to manipulate codelets and tasks.

17.11.2 Data Structure Documentation

17.11.2.1 struct starpu_codelet

The codelet structure describes a kernel that is possibly implemented on various targets. For compatibility, make sure to initialize the whole structure to zero, either by using explicit memset, or the function starpu_codelet_init(), or by letting the compiler implicitly do it in e.g. static storage case.

Data Fields

- uint32_t where
- int(∗ can_execute)(unsigned workerid, struct starpu_task ∗task, unsigned nimpl)
- enum starpu_codelet_type type
- int max_parallelism
- starpu_cpu_func_t cpu_func
- starpu_cuda_func_t cuda_func
- starpu_opencl_func_t opencl_func
- starpu_cpu_func_t cpu_funcs [STARPU_MAXIMPLEMENTATIONS]
- starpu_cuda_func_t cuda_funcs [STARPU_MAXIMPLEMENTATIONS]
- starpu_opencl_func_t opencl_funcs [STARPU_MAXIMPLEMENTATIONS]
- unsigned nbuffers
- enum starpu_data_access_mode modes [STARPU_NMAXBUFS]
- enum starpu_data_access_mode ∗ dyn_modes
- unsigned specific_nodes
- int nodes [STARPU_NMAXBUFS]
- int ∗ dyn_nodes
- struct starpu_perfmodel ∗ model
- struct starpu_perfmodel ∗ power_model
- unsigned long per_worker_stats [STARPU_NMAXWORKERS]
- const char ∗ name

17.11.2.1.1 Field Documentation

17.11.2.1.1.1 starpu_codelet::where

Optional field to indicate which types of processing units are able to execute the codelet. The different values STARPU_CPU, STARPU_CUDA, STARPU_OPENCL can be combined to specify on which types of processing units the codelet can be executed. STARPU_CPU|STARPU_CUDA for instance indicates that the codelet is implemented for both CPU cores and CUDA devices while STARPU_OPENCL indicates that it is only available on OpenCL devices. If the field is unset, its value will be automatically set based on the availability of the XXX_funcs fields defined below.

17.11.2.1.1.2 starpu_codelet::can_execute

Define a function which should return 1 if the worker designated by workerid can execute the `nimpl`th implementation of the given task, 0 otherwise.

17.11.2.1.1.3 starpu_codelet::type

Optional field to specify the type of the codelet. The default is STARPU_SEQ, i.e. usual sequential implementation. Other values (STARPU_SPMD or STARPU_FORKJOIN declare that a parallel implementation is also available. See Parallel Tasks for details.

17.11.2.1.1.4 starpu_codelet::max_parallelism

Optional field. If a parallel implementation is available, this denotes the maximum combined worker size that StarPU will use to execute parallel tasks for this codelet.

17.11.2.1.1.5 starpu_codelet::cpu_func

Deprecated Optional field which has been made deprecated. One should use instead the field starpu_codelet::cpu_funcs.

17.11.2.1.1.6 starpu_codelet::cuda_func

Deprecated Optional field which has been made deprecated. One should use instead the starpu_codelet::cuda_funcs field.

17.11.2.1.1.7 starpu_codelet::opencl_func

Deprecated Optional field which has been made deprecated. One should use instead the starpu_codelet::opencl_funcs field.

17.11.2.1.1.8 starpu_codelet::cpu_funcs

Optional array of function pointers to the CPU implementations of the codelet. It must be terminated by a NULL value. The functions prototype must be:

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

The first argument being the array of data managed by the data management library, and the second argument is a pointer to the argument passed from the field starpu_task::cl_arg. If the field starpu_codelet::where is set, then the field starpu_codelet::cpu_funcs is ignored if STARPU_CPU does not appear in the field starpu_codelet::where, it must be non-null otherwise.

17.11.2.1.1.9 starpu_codelet::cuda_funcs

Optional array of function pointers to the CUDA implementations of the codelet. It must be terminated by a NULL value. The functions must be host-functions written in the CUDA runtime API. Their prototype must be:

```
void cuda_func(void *buffers[], void *cl_arg)
```

If the field starpu_codelet::where is set, then the field starpu_codelet::cuda_funcs is ignored if STARPU_CUDA does not appear in the field starpu_codelet::where, it must be non-null otherwise.

17.11.2.1.1.10 starpu_codelet::opencl_funcs

Optional array of function pointers to the OpenCL implementations of the codelet. It must be terminated by a NULL value. The functions prototype must be:

```
void opencl_func(void *buffers[], void *cl_arg)
```

If the field starpu_codelet::where field is set, then the field starpu_codelet::opencl_funcs is ignored if STARPU_OPENCL does not appear in the field starpu_codelet::where, it must be non-null otherwise.

17.11.2.1.1.11 starpu_codelet::nbuffers

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

17.11.2.1.1.12 starpu_codelet::modes

Is an array of starpu_data_access_mode. It describes the required access modes to the data neeeded by the codelet (e.g. STARPU_RW). The number of entries in this array must be specified in the field starpu_codelet::nbuffers, and should not exceed STARPU_NMAXBUFS. If unsufficient, this value can be set with the configure option --enable-maxbuffers.

17.11.2.1.1.13 starpu_codelet::dyn_modes

Is an array of starpu_data_access_mode. It describes the required access modes to the data needed by the codelet (e.g. STARPU_RW). The number of entries in this array must be specified in the field starpu_codelet::nbuffers. This field should be used for codelets having a number of datas greater than STARPU_NMAXBUFS (see Setting The Data Handles For A Task). When defining a codelet, one should either define this field or the field starpu_codelet::modes defined above.

17.11.2.1.1.14 starpu_codelet::specific_nodes

Default value is 0. If this flag is set, StarPU will not systematically send all data to the memory node where the task will be executing, it will read the starpu_codelet::nodes or starpu_codelet::dyn_nodes array to determine, for each data, whether to send it on the memory node where the task will be executing (-1), or on a specific node (!= -1).

17.11.2.1.1.15 starpu_codelet::nodes

Optional field. When starpu_codelet::specific_nodes is 1, this specifies the memory nodes where each data should be sent to for task execution. The number of entries in this array is starpu_codelet::nbuffers, and should not exceed STARPU_NMAXBUFS.

17.11.2.1.1.16 starpu_codelet::dyn_nodes

Optional field. When starpu_codelet::specific_nodes is 1, this specifies the memory nodes where each data should be sent to for task execution. The number of entries in this array is starpu_codelet::nbuffers. This field should be used for codelets having a number of datas greater than STARPU_NMAXBUFS (see Setting The Data Handles For A Task). When defining a codelet, one should either define this field or the field starpu_codelet::nodes defined above.

17.11.2.1.1.17 starpu_codelet::model

Optional pointer to the task duration performance model associated to this codelet. This optional field is ignored when set to `NULL` or when its field starpu_perfmodel::symbol is not set.

17.11.2.1.1.18 starpu_codelet::power_model

Optional pointer to the task power consumption performance model associated to this codelet. This optional field is ignored when set to `NULL` or when its field starpu_perfmodel::field is not set. In the case of parallel codelets, this has to account for all processing units involved in the parallel execution.

17.11.2.1.1.19 starpu_codelet::per_worker_stats

Optional array for statistics collected at runtime: this is filled by StarPU and should not be accessed directly, but for example by calling the function starpu_codelet_display_stats() (See starpu_codelet_display_stats() for details).

17.11.2.1.1.20 starpu_codelet::name

Optional name of the codelet. This can be useful for debugging purposes.

17.11.2.2 struct starpu_data_descr

This type is used to describe a data handle along with an access mode.

Data Fields

starpu_data_↩ handle_t	handle	describes a data
enum starpu_data_↩ access_mode	mode	describes its access mode

17.11.2.3 struct starpu_task

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

Data Fields

- struct starpu_codelet * cl
- struct starpu_data_descr buffers [STARPU_NMAXBUFS]
- starpu_data_handle_t handles [STARPU_NMAXBUFS]
- void * interfaces [STARPU_NMAXBUFS]
- starpu_data_handle_t * dyn_handles
- void ** dyn_interfaces
- void * cl_arg
- size_t cl_arg_size
- unsigned cl_arg_free
- void(* callback_func)(void *)
- void * callback_arg
- unsigned callback_arg_free
- void(* prologue_callback_func)(void *)
- void * prologue_callback_arg
- unsigned prologue_callback_arg_free
- unsigned use_tag
- starpu_tag_t tag_id
- unsigned sequential_consistency
- unsigned synchronous
- int priority
- unsigned execute_on_a_specific_worker
- unsigned workerid
- starpu_task_bundle_t bundle
- int detach
- int destroy
- int regenerate
- enum starpu_task_status status
- struct starpu_profiling_task_info * profiling_info
- double predicted
- double predicted_transfer
- unsigned int mf_skip
- struct starpu_task * prev
- struct starpu_task * next
- void * starpu_private
- int magic
- unsigned sched_ctx
- int hypervisor_tag
- double flops
- unsigned scheduled

17.11.2.3.1 Field Documentation

17.11.2.3.1.1 starpu_task::cl

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

17.11.2.3.1.2 starpu_task::buffers

Deprecated This field has been made deprecated. One should use instead the field starpu_task::handles to specify the data handles accessed by the task. The access modes are now defined in the field starpu_codelet::modes.

17.11.2.3.1.3 starpu_task::handles

Is an array of starpu_data_handle_t. It specifies the handles to the different pieces of data accessed by the task. The number of entries in this array must be specified in the field starpu_codelet::nbuffers, and should not exceed STARPU_NMAXBUFS. If unsufficient, this value can be set with the configure option --enable-maxbuffers.

17.11.2.3.1.4 starpu_task::interfaces

The actual data pointers to the memory node where execution will happen, managed by the DSM.

17.11.2.3.1.5 starpu_task::dyn_handles

Is an array of starpu_data_handle_t. It specifies the handles to the different pieces of data accessed by the task. The number of entries in this array must be specified in the field starpu_codelet::nbuffers. This field should be used for tasks having a number of datas greater than STARPU_NMAXBUFS (see Setting The Data Handles For A Task). When defining a task, one should either define this field or the field starpu_task::handles defined above.

17.11.2.3.1.6 starpu_task::dyn_interfaces

The actual data pointers to the memory node where execution will happen, managed by the DSM. Is used when the field starpu_task::dyn_handles is defined.

17.11.2.3.1.7 starpu_task::cl_arg

Optional pointer which is passed to the codelet through the second argument of the codelet implementation (e.g. starpu_codelet::cpu_func or starpu_codelet::cuda_func). The default value is `NULL`.

17.11.2.3.1.8 starpu_task::cl_arg_size

Optional field. For some specific drivers, the pointer starpu_task::cl_arg cannot not be directly given to the driver function. A buffer of size starpu_task::cl_arg_size needs to be allocated on the driver. This buffer is then filled with the starpu_task::cl_arg_size bytes starting at address starpu_task::cl_arg. In this case, the argument given to the codelet is therefore not the starpu_task::cl_arg pointer, but the address of the buffer in local store (LS) instead. This field is ignored for CPU, CUDA and OpenCL codelets, where the starpu_task::cl_arg pointer is given as such.

17.11.2.3.1.9 starpu_task::cl_arg_free

Optional field. In case starpu_task::cl_arg was allocated by the application through `malloc()`, setting starpu_task::cl_arg_free to 1 makes StarPU automatically call `free(cl_arg)` when destroying the task. This saves the user from defining a callback just for that. This is mostly useful when targetting devices where the codelet does not execute in the same memory space as the main thread.

17.11.2.3.1.10 starpu_task::callback_func

Optional field, the default value is `NULL`. This is a function pointer of prototype `void (*f)(void *)` which specifies a possible callback. If this pointer is non-null, the callback function is executed on the host after the

execution of the task. Tasks which depend on it might already be executing. The callback is passed the value contained in the starpu_task::callback_arg field. No callback is executed if the field is set to NULL.

17.11.2.3.1.11 starpu_task::callback_arg

Optional field, the default value is `NULL`. This is the pointer passed to the callback function. This field is ignored if the field starpu_task::callback_func is set to `NULL`.

17.11.2.3.1.12 starpu_task::callback_arg_free

Optional field. In case starpu_task::callback_arg was allocated by the application through `malloc()`, setting starpu_task::callback_arg_free to 1 makes StarPU automatically call `free(callback_arg)` when destroying the task.

17.11.2.3.1.13 starpu_task::prologue_callback_func

Optional field, the default value is `NULL`. This is a function pointer of prototype `void (*f)(void *)` which specifies a possible callback. If this pointer is non-null, the callback function is executed on the host when the task becomes ready for execution, before getting scheduled. The callback is passed the value contained in the starpu_task::prologue_callback_arg field. No callback is executed if the field is set to NULL.

17.11.2.3.1.14 starpu_task::prologue_callback_arg

Optional field, the default value is `NULL`. This is the pointer passed to the prologue callback function. This field is ignored if the field starpu_task::prologue_callback_func is set to `NULL`.

17.11.2.3.1.15 starpu_task::prologue_callback_arg_free

Optional field. In case starpu_task::prologue_callback_arg was allocated by the application through `malloc()`, setting starpu_task::prologue_callback_arg_free to 1 makes StarPU automatically call `free(prologue_↩ callback_arg)` when destroying the task.

17.11.2.3.1.16 starpu_task::use_tag

Optional field, the default value is 0. If set, this flag indicates that the task should be associated with the tag contained in the starpu_task::tag_id field. Tag allow the application to synchronize with the task and to express task dependencies easily.

17.11.2.3.1.17 starpu_task::tag_id

This optional field contains the tag associated to the task if the field starpu_task::use_tag is set, it is ignored otherwise.

17.11.2.3.1.18 starpu_task::sequential_consistency

If this flag is set (which is the default), sequential consistency is enforced for the data parameters of this task for which sequential consistency is enabled. Clearing this flag permits to disable sequential consistency for this task, even if data have it enabled.

17.11.2.3.1.19 starpu_task::synchronous

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

17.11.2.3.1.20 starpu_task::priority

Optional field, the default value is STARPU_DEFAULT_PRIO. This field indicates a level of priority for the task. This is an integer value that must be set between the return values of the function starpu_sched_get_min_priority() for the least important tasks, and that of the function starpu_sched_get_max_priority() for the most important tasks (included). The STARPU_MIN_PRIO and STARPU_MAX_PRIO macros are provided for convenience and respectively returns the value of starpu_sched_get_min_priority() and starpu_sched_get_max_priority(). Default priority

is STARPU_DEFAULT_PRIO, which is always defined as 0 in order to allow static task initialization. Scheduling strategies that take priorities into account can use this parameter to take better scheduling decisions, but the scheduling policy may also ignore it.

17.11.2.3.1.21 starpu_task::execute_on_a_specific_worker

Default value is 0. If this flag is set, StarPU will bypass the scheduler and directly affect this task to the worker specified by the field starpu_task::workerid.

17.11.2.3.1.22 starpu_task::workerid

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

17.11.2.3.1.23 starpu_task::bundle

Optional field. The bundle that includes this task. If no bundle is used, this should be NULL.

17.11.2.3.1.24 starpu_task::detach

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

17.11.2.3.1.25 starpu_task::destroy

Optional value. Default value is 0 for starpu_task_init(), and 1 for starpu_task_create(). If this flag is set, the task structure will automatically be freed, either after the execution of the callback if the task is detached, or during starpu_task_wait() otherwise. If this flag is not set, dynamically allocated data structures will not be freed until starpu_task_destroy() is called explicitly. Setting this flag for a statically allocated task structure will result in undefined behaviour. The flag is set to 1 when the task is created by calling starpu_task_create(). Note that starpu_task_wait_for_all() will not free any task.

17.11.2.3.1.26 starpu_task::regenerate

Optional field. If this flag is set, the task will be re-submitted to StarPU once it has been executed. This flag must not be set if the flag starpu_task::destroy is set.

17.11.2.3.1.27 starpu_task::status

Optional field. Current state of the task.

17.11.2.3.1.28 starpu_task::profiling_info

Optional field. Profiling information for the task.

17.11.2.3.1.29 starpu_task::predicted

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

17.11.2.3.1.30 starpu_task::predicted_transfer

Optional field. Predicted data transfer duration for the task in microseconds. This field is only valid if the scheduling strategy uses performance models.

17.11.2.3.1.31 starpu_task::mf_skip

This is only used for tasks that use multiformat handle. This should only be used by StarPU.

17.11.2.3.1.32 starpu_task::prev

A pointer to the previous task. This should only be used by StarPU.

17.11.2.3.1.33 starpu_task::next

A pointer to the next task. This should only be used by StarPU.

17.11.2.3.1.34 starpu_task::starpu_private

This is private to StarPU, do not modify. If the task is allocated by hand (without starpu_task_create()), this field should be set to NULL.

17.11.2.3.1.35 starpu_task::magic

This field is set when initializing a task. The function starpu_task_submit() will fail if the field does not have the right value. This will hence avoid submitting tasks which have not been properly initialised.

17.11.2.3.1.36 starpu_task::sched_ctx

Scheduling context.

17.11.2.3.1.37 starpu_task::hypervisor_tag

Helps the hypervisor monitor the execution of this task.

17.11.2.3.1.38 starpu_task::flops

This can be set to the number of floating points operations that the task will have to achieve. This is useful for easily getting GFlops curves from the tool `starpu_perfmodel_plot`, and for the hypervisor load balancing.

17.11.2.3.1.39 starpu_task::scheduled

Whether the scheduler has pushed the task on some queue

17.11.3 Macro Definition Documentation

17.11.3.1 #define STARPU_CPU

This macro is used when setting the field starpu_codelet::where to specify the codelet may be executed on a CPU processing unit.

17.11.3.2 #define STARPU_CUDA

This macro is used when setting the field starpu_codelet::where to specify the codelet may be executed on a CUDA processing unit.

17.11.3.3 #define STARPU_OPENCL

This macro is used when setting the field starpu_codelet::where to specify the codelet may be executed on a OpenCL processing unit.

17.11.3.4 #define STARPU_MULTIPLE_CPU_IMPLEMENTATIONS

Deprecated Setting the field starpu_codelet::cpu_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::cpu_funcs.

17.11.3.5 #define STARPU_MULTIPLE_CUDA_IMPLEMENTATIONS

Deprecated Setting the field starpu_codelet::cuda_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::cuda_funcs.

17.11.3.6 #define STARPU_MULTIPLE_OPENCL_IMPLEMENTATIONS

Deprecated Setting the field starpu_codelet::opencl_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::opencl_funcs.

17.11.3.7 #define STARPU_NMAXBUFS

Defines the maximum number of buffers that tasks will be able to take as parameters. The default value is 8, it can be changed by using the configure option --enable-maxbuffers.

17.11.3.8 #define STARPU_TASK_INITIALIZER

It is possible to initialize statically allocated tasks with this value. This is equivalent to initializing a structure starpu_task with the function starpu_task_init() function.

17.11.3.9 #define STARPU_TASK_GET_HANDLE(*task, i*)

Return the `i` th data handle of the given task. If the task is defined with a static or dynamic number of handles, will either return the `i` th element of the field starpu_task::handles or the `i` th element of the field starpu_task::dyn_handles (see Setting The Data Handles For A Task)

17.11.3.10 #define STARPU_TASK_SET_HANDLE(*task, handle, i*)

Set the `i` th data handle of the given task with the given dat handle. If the task is defined with a static or dynamic number of handles, will either set the `i` th element of the field starpu_task::handles or the `i` th element of the field starpu_task::dyn_handles (see Setting The Data Handles For A Task)

17.11.3.11 #define STARPU_CODELET_GET_MODE(*codelet, i*)

Return the access mode of the `i` th data handle of the given codelet. If the codelet is defined with a static or dynamic number of handles, will either return the `i` th element of the field starpu_codelet::modes or the `i` th element of the field starpu_codelet::dyn_modes (see Setting The Data Handles For A Task)

17.11.3.12 #define STARPU_CODELET_SET_MODE(*codelet, mode, i*)

Set the access mode of the `i` th data handle of the given codelet. If the codelet is defined with a static or dynamic number of handles, will either set the `i` th element of the field starpu_codelet::modes or the `i` th element of the field starpu_codelet::dyn_modes (see Setting The Data Handles For A Task)

17.11.3.13 starpu_task_status::STARPU_TASK_INVALID

The task has just been initialized.

17.11.4 Typedef Documentation

17.11.4.1 starpu_cpu_func_t

CPU implementation of a codelet.

17.11.4.2 starpu_cuda_func_t

CUDA implementation of a codelet.

17.11.4.3 starpu_opencl_func_t

OpenCL implementation of a codelet.

17.11.5 Enumeration Type Documentation

17.11.5.1 enum starpu_codelet_type

Describes the type of parallel task. See Parallel Tasks for details.

Enumerator

STARPU_SEQ (default) for classical sequential tasks.

STARPU_SPMD for a parallel task whose threads are handled by StarPU, the code has to use starpu_combined_worker_get_size() and starpu_combined_worker_get_rank() to distribute the work.

STARPU_FORKJOIN for a parallel task whose threads are started by the codelet function, which has to use starpu_combined_worker_get_size() to determine how many threads should be started.

17.11.5.2 enum starpu_task_status

Task status

Enumerator

STARPU_TASK_BLOCKED The task has just been submitted, and its dependencies has not been checked yet.

STARPU_TASK_READY The task is ready for execution.

STARPU_TASK_RUNNING The task is running on some worker.

STARPU_TASK_FINISHED The task is finished executing.

STARPU_TASK_BLOCKED_ON_TAG The task is waiting for a tag.

STARPU_TASK_BLOCKED_ON_TASK The task is waiting for a task.

STARPU_TASK_BLOCKED_ON_DATA The task is waiting for some data.

17.11.6 Function Documentation

17.11.6.1 void starpu_codelet_init (struct starpu_codelet * cl)

Initialize `cl` with default values. Codelets should preferably be initialized statically as shown in Defining A Codelet. However such a initialisation is not always possible, e.g. when using C++.

17.11.6.2 void starpu_task_init (struct starpu_task * *task*)

Initialize task with default values. This function is implicitly called by starpu_task_create(). By default, tasks initialized with starpu_task_init() must be deinitialized explicitly with starpu_task_clean(). Tasks can also be initialized statically, using STARPU_TASK_INITIALIZER.

17.11.6.3 struct starpu_task * starpu_task_create (void)

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

17.11.6.4 struct starpu_task * starpu_task_dup (struct starpu_task * *task*)

Allocate a task structure which is the exact duplicate of the given task.

17.11.6.5 void starpu_task_clean (struct starpu_task * *task*)

Release all the structures automatically allocated to execute task, but not the task structure itself and values set by the user remain unchanged. It is thus useful for statically allocated tasks for instance. It is also useful when users want to execute the same operation several times with as least overhead as possible. It is called automatically by starpu_task_destroy(). It has to be called only after explicitly waiting for the task or after starpu_shutdown() (waiting for the callback is not enough, since StarPU still manipulates the task after calling the callback).

17.11.6.6 void starpu_task_destroy (struct starpu_task * *task*)

Free the resource allocated during starpu_task_create() and associated with task. This function is already called automatically after the execution of a task when the field starpu_task::destroy is set, which is the default for tasks created by starpu_task_create(). Calling this function on a statically allocated task results in an undefined behaviour.

17.11.6.7 int starpu_task_wait (struct starpu_task * *task*)

This function blocks until `task` has been executed. It is not possible to synchronize with a task more than once. It is not possible to wait for synchronous or detached tasks. Upon successful completion, this function returns 0. Otherwise, `-EINVAL` indicates that the specified task was either synchronous or detached.

17.11.6.8 int starpu_task_submit (struct starpu_task * *task*)

This function submits task to StarPU. Calling this function does not mean that the task will be executed immediately as there can be data or task (tag) dependencies that are not fulfilled yet: StarPU will take care of scheduling this task with respect to such dependencies. This function returns immediately if the field starpu_task::synchronous is set to 0, and block until the termination of the task otherwise. It is also possible to synchronize the application with asynchronous tasks by the means of tags, using the function starpu_tag_wait() function for instance. In case of success, this function returns 0, a return value of `-ENODEV` means that there is no worker able to process this task (e.g. there is no GPU available and this task is only implemented for CUDA devices). starpu_task_submit() can be called from anywhere, including codelet functions and callbacks, provided that the field starpu_task::synchronous is set to 0.

17.11.6.9 int starpu_task_submit_to_ctx (struct starpu_task * task, unsigned sched_ctx_id)

This function submits a task to StarPU to the context `sched_ctx_id` . By default starpu_task_submit submits the task to a global context that is created automatically by StarPU.

17.11.6.10 int starpu_task_wait_for_all (void)

This function blocks until all the tasks that were submitted (to the current context or the global one if there aren't any) are terminated. It does not destroy these tasks.

17.11.6.11 int starpu_task_wait_for_all_in_ctx (unsigned sched_ctx_id)

This function waits until all the tasks that were already submitted to the context `sched_ctx_id` have been executed

17.11.6.12 int starpu_task_nready (void)

TODO

Return the number of submitted tasks which are ready for execution are already executing. It thus does not include tasks waiting for dependencies.

17.11.6.13 int starpu_task_nsubmitted (void)

Return the number of submitted tasks which have not completed yet.

17.11.6.14 struct starpu_task * starpu_task_get_current (void)

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

17.11.6.15 void starpu_codelet_display_stats (struct starpu_codelet * cl)

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

17.11.6.16 int starpu_task_wait_for_no_ready (void)

This function waits until there is no more ready task.

17.11.6.17 void starpu_task_set_implementation (struct starpu_task * task, unsigned impl)

This function should be called by schedulers to specify the codelet implementation to be executed when executing the task.

17.11.6.18 unsigned starpu_task_get_implementation (struct starpu_task * task)

This function return the codelet implementation to be executed when executing the task.

17.11.6.19 void starpu_create_sync_task (starpu_tag_t *sync_tag*, unsigned *ndeps*, starpu_tag_t * *deps*, void(*)(void *) *callback*, void * *callback_arg*)

This creates (and submits) an empty task that unlocks a tag once all its dependencies are fulfilled.

17.12 Insert_Task

Macros

- #define STARPU_VALUE
- #define STARPU_CALLBACK
- #define STARPU_CALLBACK_WITH_ARG
- #define STARPU_CALLBACK_ARG
- #define STARPU_PRIORITY
- #define STARPU_DATA_ARRAY
- #define STARPU_EXECUTE_ON_WORKER
- #define STARPU_TAG
- #define STARPU_FLOPS
- #define STARPU_SCHED_CTX

Functions

- int starpu_insert_task (struct starpu_codelet *cl,...)
- void starpu_codelet_pack_args (void **arg_buffer, size_t *arg_buffer_size,...)
- void starpu_codelet_unpack_args (void *cl_arg,...)
- struct starpu_task * starpu_task_build (struct starpu_codelet *cl,...)

17.12.1 Detailed Description

17.12.2 Macro Definition Documentation

17.12.2.1 #define STARPU_VALUE

this macro is used when calling starpu_insert_task(), and must be followed by a pointer to a constant value and the size of the constant

17.12.2.2 #define STARPU_CALLBACK

this macro is used when calling starpu_insert_task(), and must be followed by a pointer to a callback function

17.12.2.3 #define STARPU_CALLBACK_WITH_ARG

this macro is used when calling starpu_insert_task(), and must be followed by two pointers: one to a callback function, and the other to be given as an argument to the callback function; this is equivalent to using both STARPU_CALLBACK and STARPU_CALLBACK_WITH_ARG.

17.12.2.4 #define STARPU_CALLBACK_ARG

this macro is used when calling starpu_insert_task(), and must be followed by a pointer to be given as an argument to the callback function

17.12.2.5 #define STARPU_PRIORITY

this macro is used when calling starpu_insert_task(), and must be followed by a integer defining a priority level

17.12.2.6 #define STARPU_DATA_ARRAY

TODO

17.12.2.7 #define STARPU_EXECUTE_ON_WORKER

this macro is used when calling starpu_task_insert(), and must be followed by an integer value specifying the worker on which to execute the task (as specified by starpu_task::execute_on_a_specific_worker)

17.12.2.8 #define STARPU_TAG

this macro is used when calling starpu_insert_task(), and must be followed by a tag.

17.12.2.9 #define STARPU_FLOPS

this macro is used when calling starpu_insert_task(), and must be followed by an amount of floating point operations, as a double. Users **MUST** explicitly cast into double, otherwise parameter passing will not work.

17.12.2.10 #define STARPU_SCHED_CTX

this macro is used when calling starpu_insert_task(), and must be followed by the id of the scheduling context to which we want to submit the task.

17.12.3 Function Documentation

17.12.3.1 int starpu_insert_task (struct starpu_codelet * *cl*, ...)

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

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

- STARPU_R, STARPU_W, STARPU_RW, STARPU_SCRATCH, STARPU_REDUX an access mode followed by a data handle;
- STARPU_DATA_ARRAY followed by an array of data handles and its number of elements;
- STARPU_EXECUTE_ON_WORKER followed by an integer value specifying the worker on which to execute the task (as specified by starpu_task::execute_on_a_specific_worker)
- the specific values STARPU_VALUE, STARPU_CALLBACK, STARPU_CALLBACK_ARG, STARPU_CALLBACK_WITH_ARG, STARPU_PRIORITY, STARPU_TAG, STARPU_FLOPS, STARPU_SCHED_CTX followed by the appropriated objects as defined elsewhere.

When using STARPU_DATA_ARRAY, the access mode of the data handles is not defined.

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

17.12.3.2 void starpu_codelet_pack_args (void ** *arg_buffer*, size_t * *arg_buffer_size*, ...)

Pack arguments of type STARPU_VALUE into a buffer which can be given to a codelet and later unpacked with the function starpu_codelet_unpack_args().

17.12.3.3 void starpu_codelet_unpack_args (void * *cl_arg*, ...)

Retrieve the arguments of type STARPU_VALUE associated to a task automatically created using the function starpu_insert_task().

17.12.3.4 struct starpu_task * starpu_task_build (struct starpu_codelet * *cl*, ...)

Create a task corresponding to `cl` with the following arguments. The argument list must be zero-terminated. The arguments following the codelet are the same as the ones for the function starpu_insert_task().

17.13 Explicit Dependencies

Typedefs

- typedef uint64_t starpu_tag_t

Functions

- void starpu_task_declare_deps_array (struct starpu_task *task, unsigned ndeps, struct starpu_task *task_array[])
- void starpu_tag_declare_deps (starpu_tag_t id, unsigned ndeps,...)
- void starpu_tag_declare_deps_array (starpu_tag_t id, unsigned ndeps, starpu_tag_t *array)
- int starpu_tag_wait (starpu_tag_t id)
- int starpu_tag_wait_array (unsigned ntags, starpu_tag_t *id)
- void starpu_tag_restart (starpu_tag_t id)
- void starpu_tag_remove (starpu_tag_t id)
- void starpu_tag_notify_from_apps (starpu_tag_t id)

17.13.1 Detailed Description

17.13.2 Typedef Documentation

17.13.2.1 starpu_tag_t

This type defines a task logical identifer. It is possible to associate a task with a unique *tag* chosen by the application, and to express dependencies between tasks by the means of those tags. To do so, fill the field starpu_task::tag_id with a tag number (can be arbitrary) and set the field starpu_task::use_tag to 1. If starpu_tag_declare_deps() is called with this tag number, the task will not be started until the tasks which holds the declared dependency tags are completed.

17.13.3 Function Documentation

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

Declare task dependencies between a `task` and an array of tasks of length `ndeps`. This function must be called prior to the submission of the task, but it may called after the submission or the execution of the tasks in the array,

provided the tasks are still valid (i.e. they were not automatically destroyed). Calling this function on a task that was already submitted or with an entry of `task_array` that is no longer a valid task results in an undefined behaviour. If `ndeps` is 0, no dependency is added. It is possible to call starpu_task_declare_deps_array() several times on the same task, in this case, the dependencies are added. It is possible to have redundancy in the task dependencies.

17.13.3.2 void starpu_tag_declare_deps (starpu_tag_t *id*, unsigned *ndeps*, ...)

Specify the dependencies of the task identified by tag `id`. The first argument specifies the tag which is configured, the second argument gives the number of tag(s) on which `id` depends. The following arguments are the tags which have to be terminated to unlock the task. This function must be called before the associated task is submitted to StarPU with starpu_task_submit().

WARNING! Use with caution. Because of the variable arity of starpu_tag_declare_deps(), note that the last arguments must be of type starpu_tag_t : constant values typically need to be explicitly casted. Otherwise, due to integer sizes and argument passing on the stack, the C compiler might consider the tag `0x200000003` instead of `0x2` and `0x3` when calling `starpu_tag_declare_deps(0x1, 2, 0x2, 0x3)`. Using the starpu_tag_declare_deps_array() function avoids this hazard.

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

17.13.3.3 void starpu_tag_declare_deps_array (starpu_tag_t *id*, unsigned *ndeps*, starpu_tag_t * *array*)

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

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

17.13.3.4 int starpu_tag_wait (starpu_tag_t *id*)

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

17.13.3.5 int starpu_tag_wait_array (unsigned *ntags*, starpu_tag_t * *id*)

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

17.13.3.6 void starpu_tag_restart (starpu_tag_t *id*)

This function can be used to clear the *already notified* status of a tag which is not associated with a task. Before that, calling starpu_tag_notify_from_apps() again will not notify the successors. After that, the next call to starpu_tag_notify_from_apps() will notify the successors.

17.13.3.7 void starpu_tag_remove (starpu_tag_t *id*)

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

17.13.3.8 void starpu_tag_notify_from_apps (starpu_tag_t *id*)

This function explicitly unlocks tag `id`. It may be useful in the case of applications which execute part of their computation outside StarPU tasks (e.g. third-party libraries). It is also provided as a convenient tool for the programmer, for instance to entirely construct the task DAG before actually giving StarPU the opportunity to execute the tasks. When called several times on the same tag, notification will be done only on first call, thus implementing "OR" dependencies, until the tag is restarted using starpu_tag_restart().

17.14 Implicit Data Dependencies

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

Functions

- void starpu_data_set_default_sequential_consistency_flag (unsigned flag)
- unsigned starpu_data_get_default_sequential_consistency_flag (void)
- void starpu_data_set_sequential_consistency_flag (starpu_data_handle_t handle, unsigned flag)

17.14.1 Detailed Description

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

17.14.2 Function Documentation

17.14.2.1 starpu_data_set_default_sequential_consistency_flag (unsigned *flag*)

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

17.14.2.2 unsigned starpu_data_get_default_sequential_consistency_flag (void)

Return the default sequential consistency flag

17.14.2.3 void starpu_data_set_sequential_consistency_flag (starpu_data_handle_t *handle,* unsigned *flag*)

Set the data consistency mode associated to a data handle. The consistency mode set using this function has the priority over the default mode which can be set with starpu_data_set_default_sequential_consistency_flag().

17.15 Performance Model

Data Structures

- struct starpu_perfmodel
- struct starpu_perfmodel_regression_model
- struct starpu_perfmodel_per_arch
- struct starpu_perfmodel_history_list
- struct starpu_perfmodel_history_entry

Enumerations

- enum starpu_perfmodel_archtype { STARPU_CPU_DEFAULT, STARPU_CUDA_DEFAULT, STARPU_OPENCL_DEFAULT }
- enum starpu_perfmodel_type {
  STARPU_PER_ARCH, STARPU_COMMON, STARPU_HISTORY_BASED, STARPU_REGRESSION_BASED,
  STARPU_NL_REGRESSION_BASED }

Functions

- int starpu_perfmodel_load_symbol (const char *symbol, struct starpu_perfmodel *model)
- int starpu_perfmodel_unload_model (struct starpu_perfmodel *model)
- void starpu_perfmodel_debugfilepath (struct starpu_perfmodel *model, enum starpu_perfmodel_archtype arch, char *path, size_t maxlen, unsigned nimpl)
- void starpu_perfmodel_get_arch_name (enum starpu_perfmodel_archtype arch, char *archname, size_t maxlen, unsigned nimpl)
- enum starpu_perfmodel_archtype starpu_worker_get_perf_archtype (int workerid)
- int starpu_perfmodel_list (FILE *output)
- void starpu_perfmodel_directory (FILE *output)
- void starpu_perfmodel_print (struct starpu_perfmodel *model, enum starpu_perfmodel_archtype arch, unsigned nimpl, char *parameter, uint32_t *footprint, FILE *output)
- int starpu_perfmodel_print_all (struct starpu_perfmodel *model, char *arch, char *parameter, uint32_t *footprint, FILE *output)
- void starpu_bus_print_bandwidth (FILE *f)
- void starpu_bus_print_affinity (FILE *f)
- void starpu_perfmodel_update_history (struct starpu_perfmodel *model, struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned cpuid, unsigned nimpl, double measured)
- double starpu_transfer_bandwidth (unsigned src_node, unsigned dst_node)
- double starpu_transfer_latency (unsigned src_node, unsigned dst_node)
- double starpu_transfer_predict (unsigned src_node, unsigned dst_node, size_t size)

17.15.1 Detailed Description

17.15.2 Data Structure Documentation

17.15.2.1 struct starpu_perfmodel

Contains all information about a performance model. At least the type and symbol fields have to be filled when defining a performance model for a codelet. For compatibility, make sure to initialize the whole structure to zero, either by using explicit memset, or by letting the compiler implicitly do it in e.g. static storage case. If not provided, other fields have to be zero.

Data Fields

- enum starpu_perfmodel_type type
- double(∗ cost_model)(struct starpu_data_descr ∗)
- double(∗ cost_function)(struct starpu_task ∗, unsigned nimpl)
- size_t(∗ size_base)(struct starpu_task ∗, unsigned nimpl)
- struct starpu_perfmodel_per_arch per_arch [STARPU_NARCH_VARIATIONS][STARPU_MAXIMPLEMENTATIONS]
- const char ∗ symbol
- unsigned is_loaded
- unsigned benchmarking
- starpu_pthread_rwlock_t model_rwlock

17.15.2.1.1 Field Documentation

17.15.2.1.1.1 starpu_perfmodel::type

is the type of performance model

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

17.15.2.1.1.2 starpu_perfmodel::cost_model

Deprecated This field is deprecated. Use instead the field starpu_perfmodel::cost_function field.

17.15.2.1.1.3 starpu_perfmodel::cost_function

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

17.15.2.1.1.4 starpu_perfmodel::size_base

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

17.15.2.1.1.5 starpu_perfmodel::per_arch

Used by STARPU_PER_ARCH: array of structures starpu_per_arch_perfmodel

17.15.2.1.1.6 starpu_perfmodel::symbol

is the symbol name for the performance model, which will be used as file name to store the model. It must be set otherwise the model will be ignored.

17.15.2.1.1.7 starpu_perfmodel::is_loaded

Whether the performance model is already loaded from the disk.

17.15.2.1.1.8 starpu_perfmodel::benchmarking

Whether the performance model is still being calibrated.

17.15.2.1.1.9 starpu_perfmodel::model_rwlock

Lock to protect concurrency between loading from disk (W), updating the values (W), and making a performance estimation (R).

17.15.2.2 struct starpu_perfmodel_regression_model

...

Data Fields

double	sumlny	sum of ln(measured)
double	sumlnx	sum of ln(size)
double	sumlnx2	sum of ln(size)^2
unsigned long	minx	minimum size
unsigned long	maxx	maximum size
double	sumlnxlny	sum of ln(size)*ln(measured)
double	alpha	estimated = alpha * size ^ beta
double	beta	estimated = alpha * size ^ beta
unsigned	valid	whether the linear regression model is valid (i.e. enough measures)
double	a	estimated = a size ^b + c
double	b	estimated = a size ^b + c
double	c	estimated = a size ^b + c
unsigned	nl_valid	whether the non-linear regression model is valid (i.e. enough measures)
unsigned	nsample	number of sample values for non-linear regression

17.15.2.3 struct starpu_perfmodel_per_arch

contains information about the performance model of a given arch.

Data Fields

- double(* cost_model)(struct starpu_data_descr *t)
- double(* cost_function)(struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- size_t(* size_base)(struct starpu_task *, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- struct starpu_perfmodel_history_table * history
- struct starpu_perfmodel_history_list * list
- struct starpu_perfmodel_regression_model regression

17.15.2.3.1 Field Documentation

17.15.2.3.1.1 starpu_perfmodel_per_arch::cost_model

Deprecated This field is deprecated. Use instead the field starpu_perfmodel_per_arch::cost_function.

17.15.2.3.1.2 starpu_perfmodel_per_arch::cost_function

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

17.15.2.3.1.3 starpu_perfmodel_per_arch::size_base

Same as in structure starpu_perfmodel, but per-arch, in case it depends on the architecture-specific implementation.

17.15.2.3.1.4 starpu_perfmodel_per_arch::history

The history of performance measurements.

17.15.2.3.1.5 starpu_perfmodel_per_arch::list

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

17.15.2.3.1.6 starpu_perfmodel_per_arch::regression

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

17.15.2.4 struct starpu_perfmodel_history_list

todo

Data Fields

struct starpu_perfmodel_history_list *	next	todo
struct starpu_perfmodel_history_entry *	entry	todo

17.15.2.5 struct starpu_perfmodel_history_entry

todo

Data Fields

double	mean	mean_n = 1/n sum
double	deviation	n dev_n = sum2 - 1/n (sum)^2
double	sum	sum of samples (in µs)
double	sum2	sum of samples^2
unsigned	nsample	number of samples
uint32_t	footprint	data footprint
size_t	size	in bytes
double	flops	Provided by the application

17.15.3 Enumeration Type Documentation

17.15.3.1 enum starpu_perfmodel_archtype

Enumerates the various types of architectures.

it is possible that we have multiple versions of the same kind of workers, for instance multiple GPUs or even different CPUs within the same machine so we do not use the archtype enum type directly for performance models.

- CPU types range within STARPU_CPU_DEFAULT (1 CPU), STARPU_CPU_DEFAULT+1 (2 CPUs), ... STARPU_CPU_DEFAULT + STARPU_MAXCPUS - 1 (STARPU_MAXCPUS CPUs).
- CUDA types range within STARPU_CUDA_DEFAULT (GPU number 0), STARPU_CUDA_DEFAULT + 1 (GPU number 1), ..., STARPU_CUDA_DEFAULT + STARPU_MAXCUDADEVS - 1 (GPU number STARPU_MAXCUDADEVS - 1).
- OpenCL types range within STARPU_OPENCL_DEFAULT (GPU number 0), STARPU_OPENCL_DEFAULT + 1 (GPU number 1), ..., STARPU_OPENCL_DEFAULT + STARPU_MAXOPENCLDEVS - 1 (GPU number STARPU_MAXOPENCLDEVS - 1).

Enumerator

STARPU_CPU_DEFAULT CPU combined workers between 0 and STARPU_MAXCPUS-1

STARPU_CUDA_DEFAULT CUDA workers

STARPU_OPENCL_DEFAULT OpenCL workers

17.15.3.2 enum starpu_perfmodel_type

TODO

Enumerator

STARPU_PER_ARCH Application-provided per-arch cost model function

STARPU_COMMON Application-provided common cost model function, with per-arch factor

STARPU_HISTORY_BASED Automatic history-based cost model

STARPU_REGRESSION_BASED Automatic linear regression-based cost model (alpha ∗ size ^ beta)

STARPU_NL_REGRESSION_BASED Automatic non-linear regression-based cost model (a ∗ size ^ b + c)

17.15.4 Function Documentation

17.15.4.1 int starpu_perfmodel_load_symbol (const char ∗ *symbol*, struct starpu_perfmodel ∗ *model*)

loads a given performance model. The model structure has to be completely zero, and will be filled with the information saved in `$STARPU_HOME/.starpu`. The function is intended to be used by external tools that should read the performance model files.

17.15.4.2 int starpu_perfmodel_unload_model (struct starpu_perfmodel ∗ *model*)

unloads the given model which has been previously loaded through the function starpu_perfmodel_load_symbol()

17.15.4.3 void starpu_perfmodel_debugfilepath (struct starpu_perfmodel ∗ *model*, enum starpu_perfmodel_archtype *arch*, char ∗ *path*, size_t *maxlen*, unsigned *nimpl*)

returns the path to the debugging information for the performance model.

17.15.4.4 void starpu_perfmodel_get_arch_name (enum starpu_perfmodel_archtype *arch*, char ∗ *archname*, size_t *maxlen*, unsigned *nimpl*)

returns the architecture name for `arch`

17.15.4.5 enum starpu_perfmodel_archtype starpu_worker_get_perf_archtype (int *workerid*)

returns the architecture type of a given worker.

17.15.4.6 int starpu_perfmodel_list (FILE * *output*)

prints a list of all performance models on `output`

17.15.4.7 int starpu_perfmodel_directory (FILE * *output*)

prints the directory name storing performance models on `output`

17.15.4.8 void starpu_perfmodel_print (struct starpu_perfmodel * *model*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*, char * *parameter*, uint32_t * *footprint*, FILE * *output*)

todo

17.15.4.9 int starpu_perfmodel_print_all (struct starpu_perfmodel * *model*, char * *arch*, char * *parameter*, uint32_t * *footprint*, FILE * *output*)

todo

17.15.4.10 void starpu_bus_print_bandwidth (FILE * *f*)

prints a matrix of bus bandwidths on `f`.

17.15.4.11 void starpu_bus_print_affinity (FILE * *f*)

prints the affinity devices on `f`.

17.15.4.12 void starpu_perfmodel_update_history (struct starpu_perfmodel * *model*, struct starpu_task * *task*, enum starpu_perfmodel_archtype *arch*, unsigned *cpuid*, unsigned *nimpl*, double *measured*)

This feeds the performance model model with an explicit measurement measured (in µs), in addition to measurements done by StarPU itself. This can be useful when the application already has an existing set of measurements done in good conditions, that StarPU could benefit from instead of doing on-line measurements. And example of use can be seen in Performance Model Example.

17.15.4.13 double starpu_transfer_bandwidth (unsigned *src_node*, unsigned *dst_node*)

Return the bandwidth of data transfer between two memory nodes

17.15.4.14 double starpu_transfer_latency (unsigned *src_node*, unsigned *dst_node*)

Return the latency of data transfer between two memory nodes

17.15.4.15 double starpu_transfer_predict (unsigned *src_node*, unsigned *dst_node*, size_t *size*)

Return the estimated time to transfer a given size between two memory nodes.

17.16 Profiling

Data Structures

- struct starpu_profiling_task_info
- struct starpu_profiling_worker_info
- struct starpu_profiling_bus_info

Macros

- #define STARPU_PROFILING_DISABLE
- #define STARPU_PROFILING_ENABLE

Functions

- int starpu_profiling_status_set (int status)
- int starpu_profiling_status_get (void)
- void starpu_profiling_init ()
- void starpu_profiling_set_id (int new_id)
- int starpu_profiling_worker_get_info (int workerid, struct starpu_profiling_worker_info *worker_info)
- int starpu_bus_get_profiling_info (int busid, struct starpu_profiling_bus_info *bus_info)
- int starpu_bus_get_count (void)
- int starpu_bus_get_id (int src, int dst)
- int starpu_bus_get_src (int busid)
- int starpu_bus_get_dst (int busid)
- double starpu_timing_timespec_delay_us (struct timespec *start, struct timespec *end)
- double starpu_timing_timespec_to_us (struct timespec *ts)
- void starpu_profiling_bus_helper_display_summary (void)
- void starpu_profiling_worker_helper_display_summary (void)
- void starpu_data_display_memory_stats ()

17.16.1 Detailed Description

17.16.2 Data Structure Documentation

17.16.2.1 struct starpu_profiling_task_info

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

Data Fields

struct timespec	submit_time	Date of task submission (relative to the initialization of StarPU).
struct timespec	push_start_time	Time when the task was submitted to the scheduler.
struct timespec	push_end_time	Time when the scheduler finished with the task submission.
struct timespec	pop_start_time	Time when the scheduler started to be requested for a task, and eventually gave that task.
struct timespec	pop_end_time	Time when the scheduler finished providing the task for execution.
struct timespec	acquire_data_start_time	Time when the worker started fetching input data.

struct timespec	acquire_data_↩ end_time	Time when the worker finished fetching input data.
struct timespec	start_time	Date of task execution beginning (relative to the initialization of StarPU).
struct timespec	end_time	Date of task execution termination (relative to the initialization of StarPU).
struct timespec	release_data_↩ start_time	Time when the worker started releasing data.
struct timespec	release_data_↩ end_time	Time when the worker finished releasing data.
struct timespec	callback_start_↩ time	Time when the worker started the application callback for the task.
struct timespec	callback_end_↩ time	Time when the worker finished the application callback for the task.
int	workerid	Identifier of the worker which has executed the task.
uint64_t	used_cycles	Number of cycles used by the task, only available in the MoviSim
uint64_t	stall_cycles	Number of cycles stalled within the task, only available in the MoviSim
double	power_↩ consumed	Power consumed by the task, only available in the MoviSim

17.16.2.2 struct starpu_profiling_worker_info

This structure contains the profiling information associated to a worker. The timing is provided since the previous call to starpu_profiling_worker_get_info()

Data Fields

struct timespec	start_time	Starting date for the reported profiling measurements.
struct timespec	total_time	Duration of the profiling measurement interval.
struct timespec	executing_time	Time spent by the worker to execute tasks during the profiling measurement interval.
struct timespec	sleeping_time	Time spent idling by the worker during the profiling measurement interval.
int	executed_tasks	Number of tasks executed by the worker during the profiling measurement interval.
uint64_t	used_cycles	Number of cycles used by the worker, only available in the MoviSim
uint64_t	stall_cycles	Number of cycles stalled within the worker, only available in the MoviSim
double	power_↩ consumed	Power consumed by the worker, only available in the MoviSim

17.16.2.3 struct starpu_profiling_bus_info

todo

Data Fields

struct timespec	start_time	Time of bus profiling startup.
struct timespec	total_time	Total time of bus profiling.
int long long	transferred_↩ bytes	Number of bytes transferred during profiling.
int	transfer_count	Number of transfers during profiling.

17.16.3 Macro Definition Documentation

17.16.3.1 STARPU_PROFILING_DISABLE

This value is used when calling the function starpu_profiling_status_set() to disable profiling.

17.16.3.2 STARPU_PROFILING_ENABLE

This value is used when calling the function starpu_profiling_status_set() to enable profiling.

17.16.4 Function Documentation

17.16.4.1 int starpu_profiling_status_set (int *status*)

This function sets the profiling status. Profiling is activated by passing STARPU_PROFILING_ENABLE in status. Passing STARPU_PROFILING_DISABLE disables profiling. Calling this function resets all profiling measurements. When profiling is enabled, the field starpu_task::profiling_info points to a valid structure starpu_profiling_task_info containing information about the execution of the task. Negative return values indicate an error, otherwise the previous status is returned.

17.16.4.2 int starpu_profiling_status_get (void)

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

17.16.4.3 int starpu_profiling_init (void)

This function resets performance counters and enable profiling if the environment variable STARPU_PROFILING is set to a positive value.

17.16.4.4 void starpu_profiling_set_id (int *new_id*)

This function sets the ID used for profiling trace filename. It needs to be called before starpu_init().

17.16.4.5 int starpu_profiling_worker_get_info (int *workerid,* struct **starpu_profiling_worker_info** * *worker_info*)

Get the profiling info associated to the worker identified by `workerid`, and reset the profiling measurements. If the argument `worker_info` is NULL, only reset the counters associated to worker `workerid`. Upon successful completion, this function returns 0. Otherwise, a negative value is returned.

17.16.4.6 int starpu_bus_get_profiling_info (int *busid,* struct **starpu_profiling_bus_info** * *bus_info*)

todo

17.16.4.7 int starpu_bus_get_count (void)

Return the number of buses in the machine

17.16.4.8 int starpu_bus_get_id (int *src,* int *dst*)

Return the identifier of the bus between `src` and `dst`

17.16.4.9 int starpu_bus_get_src (int *busid*)

Return the source point of bus `busid`

17.16.4.10 int starpu_bus_get_dst (int *busid*)

Return the destination point of bus `busid`

17.16.4.11 double starpu_timing_timespec_delay_us (struct timespec * *start*, struct timespec * *end*)

Returns the time elapsed between `start` and `end` in microseconds.

17.16.4.12 double starpu_timing_timespec_to_us (struct timespec * *ts*)

Converts the given timespec `ts` into microseconds

17.16.4.13 void starpu_profiling_bus_helper_display_summary (void)

Displays statistics about the bus on stderr. if the environment variable STARPU_BUS_STATS is defined. The function is called automatically by starpu_shutdown().

17.16.4.14 void starpu_profiling_worker_helper_display_summary (void)

Displays statistics about the workers on stderr if the environment variable STARPU_WORKER_STATS is defined. The function is called automatically by starpu_shutdown().

17.16.4.15 void starpu_data_display_memory_stats ( )

Display statistics about the current data handles registered within StarPU. StarPU must have been configured with the configure option --enable-memory-stats (see Memory Feedback).

17.17 Theoretical Lower Bound on Execution Time

Compute theoretical upper computation efficiency bound corresponding to some actual execution.

Functions

- void starpu_bound_start (int deps, int prio)
- void starpu_bound_stop (void)
- void starpu_bound_print_dot (FILE *output)
- void starpu_bound_compute (double *res, double *integer_res, int integer)
- void starpu_bound_print_lp (FILE *output)
- void starpu_bound_print_mps (FILE *output)
- void starpu_bound_print (FILE *output, int integer)

17.17.1 Detailed Description

Compute theoretical upper computation efficiency bound corresponding to some actual execution.

17.17.2 Function Documentation

17.17.2.1 void starpu_bound_start (int *deps*, int *prio*)

Start recording tasks (resets stats). `deps` tells whether dependencies should be recorded too (this is quite expensive)

17.17.2.2 void starpu_bound_stop (void)

Stop recording tasks

17.17.2.3 void starpu_bound_print_dot (FILE * *output*)

Print the DAG that was recorded

17.17.2.4 void starpu_bound_compute (double * *res*, double * *integer_res*, int *integer*)

Get theoretical upper bound (in ms) (needs glpk support detected by configure script). It returns 0 if some performance models are not calibrated.

17.17.2.5 void starpu_bound_print_lp (FILE * *output*)

Emit the Linear Programming system on `output` for the recorded tasks, in the lp format

17.17.2.6 void starpu_bound_print_mps (FILE * *output*)

Emit the Linear Programming system on `output` for the recorded tasks, in the mps format

17.17.2.7 void starpu_bound_print (FILE * *output*, int *integer*)

Emit statistics of actual execution vs theoretical upper bound. `integer` permits to choose between integer solving (which takes a long time but is correct), and relaxed solving (which provides an approximate solution).

17.18 CUDA Extensions

Macros

- #define STARPU_USE_CUDA
- #define STARPU_MAXCUDADEVS
- #define STARPU_CUDA_REPORT_ERROR(status)
- #define STARPU_CUBLAS_REPORT_ERROR(status)

Functions

- cudaStream_t starpu_cuda_get_local_stream (void)
- const struct cudaDeviceProp * starpu_cuda_get_device_properties (unsigned workerid)
- void starpu_cuda_report_error (const char *func, const char *file, int line, cudaError_t status)
- int starpu_cuda_copy_async_sync (void *src_ptr, unsigned src_node, void *dst_ptr, unsigned dst_node, size_t ssize, cudaStream_t stream, enum cudaMemcpyKind kind)
- void starpu_cuda_set_device (unsigned devid)

- void starpu_cublas_init (void)
- void starpu_cublas_shutdown (void)
- void starpu_cublas_report_error (const char *func, const char *file, int line, cublasStatus status)

17.18.1 Detailed Description

17.18.2 Macro Definition Documentation

17.18.2.1 #define STARPU_USE_CUDA

This macro is defined when StarPU has been installed with CUDA support. It should be used in your code to detect the availability of CUDA as shown in Full source code for the 'Scaling a Vector' example.

17.18.2.2 #define STARPU_MAXCUDADEVS

This macro defines the maximum number of CUDA devices that are supported by StarPU.

17.18.2.3 #define STARPU_CUDA_REPORT_ERROR(*status*)

Calls starpu_cuda_report_error(), passing the current function, file and line position.

17.18.2.4 #define STARPU_CUBLAS_REPORT_ERROR(*status*)

Calls starpu_cublas_report_error(), passing the current function, file and line position.

17.18.3 Function Documentation

17.18.3.1 cudaStream_t starpu_cuda_get_local_stream (void)

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

17.18.3.2 const struct cudaDeviceProp * starpu_cuda_get_device_properties (unsigned *workerid*)

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

17.18.3.3 void starpu_cuda_report_error (const char * *func*, const char * *file*, int *line*, cudaError_t *status*)

Report a CUDA error.

17.18.3.4 int starpu_cuda_copy_async_sync (void * *src_ptr*, unsigned *src_node*, void * *dst_ptr*, unsigned *dst_node*, size_t *ssize*, cudaStream_t *stream*, enum cudaMemcpyKind *kind*)

Copy `ssize` bytes from the pointer `src_ptr` on `src_node` to the pointer `dst_ptr` on `dst_node`. The function first tries to copy the data asynchronous (unless stream is `NULL`). If the asynchronous copy fails or if stream is `NULL`, it copies the data synchronously. The function returns `-EAGAIN` if the asynchronous launch was successfull. It returns 0 if the synchronous copy was successful, or fails otherwise.

17.18.3.5 void starpu_cuda_set_device (unsigned *devid*)

Calls cudaSetDevice(devid) or cudaGLSetGLDevice(devid), according to whether `devid` is among the field starpu_conf::cuda_opengl_interoperability.

17.18.3.6 void starpu_cublas_init (void)

This function initializes CUBLAS on every CUDA device. The CUBLAS library must be initialized prior to any CUBLAS call. Calling starpu_cublas_init() will initialize CUBLAS on every CUDA device controlled by StarPU. This call blocks until CUBLAS has been properly initialized on every device.

17.18.3.7 void starpu_cublas_shutdown (void)

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

17.18.3.8 void starpu_cublas_report_error (const char * *func*, const char * *file*, int *line*, cublasStatus *status*)

Report a cublas error.

17.19 OpenCL Extensions

Data Structures

- struct starpu_opencl_program

Macros

- #define STARPU_USE_OPENCL
- #define STARPU_MAXOPENCLDEVS
- #define STARPU_OPENCL_DATADIR

Writing OpenCL kernels

- void starpu_opencl_get_context (int devid, cl_context *context)
- void starpu_opencl_get_device (int devid, cl_device_id *device)
- void starpu_opencl_get_queue (int devid, cl_command_queue *queue)
- void starpu_opencl_get_current_context (cl_context *context)
- void starpu_opencl_get_current_queue (cl_command_queue *queue)
- int starpu_opencl_set_kernel_args (cl_int *err, cl_kernel *kernel,...)

Compiling OpenCL kernels

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

- int starpu_opencl_load_opencl_from_file (const char *source_file_name, struct starpu_opencl_program *opencl_programs, const char *build_options)

- int starpu_opencl_load_opencl_from_string (const char *opencl_program_source, struct starpu_opencl_↩ program *opencl_programs, const char *build_options)
- int starpu_opencl_unload_opencl (struct starpu_opencl_program *opencl_programs)
- void starpu_opencl_load_program_source (const char *source_file_name, char *located_file_name, char *located_dir_name, char *opencl_program_source)
- int starpu_opencl_compile_opencl_from_file (const char *source_file_name, const char *build_options)
- int starpu_opencl_compile_opencl_from_string (const char *opencl_program_source, const char *file_name, const char *build_options)
- int starpu_opencl_load_binary_opencl (const char *kernel_id, struct starpu_opencl_program *opencl_↩ programs)

Loading OpenCL kernels

- int starpu_opencl_load_kernel (cl_kernel *kernel, cl_command_queue *queue, struct starpu_opencl_↩ program *opencl_programs, const char *kernel_name, int devid)
- int starpu_opencl_release_kernel (cl_kernel kernel)

OpenCL statistics

- int starpu_opencl_collect_stats (cl_event event)

OpenCL utilities

- #define STARPU_OPENCL_DISPLAY_ERROR(status)
- #define STARPU_OPENCL_REPORT_ERROR(status)
- #define STARPU_OPENCL_REPORT_ERROR_WITH_MSG(msg, status)
- const char * starpu_opencl_error_string (cl_int status)
- void starpu_opencl_display_error (const char *func, const char *file, int line, const char *msg, cl_int status)
- static __starpu_inline void starpu_opencl_report_error (const char *func, const char *file, int line, const char *msg, cl_int status)
- cl_int starpu_opencl_allocate_memory (cl_mem *addr, size_t size, cl_mem_flags flags)
- cl_int starpu_opencl_copy_ram_to_opencl (void *ptr, unsigned src_node, cl_mem buffer, unsigned dst_node, size_t size, size_t offset, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_opencl_to_ram (cl_mem buffer, unsigned src_node, void *ptr, unsigned dst_node, size_t size, size_t offset, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_opencl_to_opencl (cl_mem src, unsigned src_node, size_t src_offset, cl_mem dst, unsigned dst_node, size_t dst_offset, size_t size, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_async_sync (uintptr_t src, size_t src_offset, unsigned src_node, uintptr_t dst, size_t dst_offset, unsigned dst_node, size_t size, cl_event *event)

17.19.1 Detailed Description

17.19.2 Data Structure Documentation

17.19.2.1 struct starpu_opencl_program

Stores the OpenCL programs as compiled for the different OpenCL devices.

Data Fields

cl_program	programs[STARPU_MAXOPENCLDEVS]	Stores each program for each OpenCL device.

17.19.3 Macro Definition Documentation

17.19.3.1 #define STARPU_USE_OPENCL

This macro is defined when StarPU has been installed with OpenCL support. It should be used in your code to detect the availability of OpenCL as shown in Full source code for the 'Scaling a Vector' example.

17.19.3.2 #define STARPU_MAXOPENCLDEVS

This macro defines the maximum number of OpenCL devices that are supported by StarPU.

17.19.3.3 #define STARPU_OPENCL_DATADIR

This macro defines the directory in which the OpenCL codelets of the applications provided with StarPU have been installed.

17.19.3.4 #define STARPU_OPENCL_DISPLAY_ERROR(*status*)

Call the function starpu_opencl_display_error() with the given error `status`, the current function name, current file and line number, and a empty message.

17.19.3.5 #define STARPU_OPENCL_REPORT_ERROR(*status*)

Call the function starpu_opencl_report_error() with the given error `status`, with the current function name, current file and line number, and a empty message.

17.19.3.6 #define STARPU_OPENCL_REPORT_ERROR_WITH_MSG(*msg, status*)

Call the function starpu_opencl_report_error() with the given `msg` and the given error `status`, with the current function name, current file and line number.

17.19.4 Function Documentation

17.19.4.1 void starpu_opencl_get_context (int *devid,* cl_context * *context*)

Places the OpenCL context of the device designated by `devid` into `context`.

17.19.4.2 void starpu_opencl_get_device (int *devid,* cl_device_id * *device*)

Places the cl_device_id corresponding to `devid` in `device`.

17.19.4.3 void starpu_opencl_get_queue (int *devid,* cl_command_queue * *queue*)

Places the command queue of the device designated by `devid` into `queue`.

17.19.4.4 void starpu_opencl_get_current_context (cl_context * *context*)

Return the context of the current worker.

17.19.4.5 void starpu_opencl_get_current_queue (cl_command_queue * *queue*)

Return the computation kernel command queue of the current worker.

17.19.4.6 int starpu_opencl_set_kernel_args (cl_int * *err*, cl_kernel * *kernel*, ...)

Sets the arguments of a given kernel. The list of arguments must be given as `(size_t size_of_the_argument, cl_mem * pointer_to_the_argument)`. The last argument must be 0. Returns the number of arguments that were successfully set. In case of failure, returns the id of the argument that could not be set and err is set to the error returned by OpenCL. Otherwise, returns the number of arguments that were set.

Here an example:

```
int n;
cl_int err;
cl_kernel kernel;
n = starpu_opencl_set_kernel_args(&err, 2, &kernel,
                                  sizeof(foo), &foo,
                                  sizeof(bar), &bar,
                                  0);
if (n != 2)
   fprintf(stderr, "Error : %d\n", err);
```

17.19.4.7 int starpu_opencl_load_opencl_from_file (const char * *source_file_name*, struct **starpu_opencl_program** * *opencl_programs*, const char * *build_options*)

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

17.19.4.8 int starpu_opencl_load_opencl_from_string (const char * *opencl_program_source*, struct **starpu_opencl_program** * *opencl_programs*, const char * *build_options*)

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

17.19.4.9 int starpu_opencl_unload_opencl (struct **starpu_opencl_program** * *opencl_programs*)

This function unloads an OpenCL compiled code.

17.19.4.10 void starpu_opencl_load_program_source (const char * *source_file_name*, char * *located_file_name*, char * *located_dir_name*, char * *opencl_program_source*)

Store the contents of the file `source_file_name` in the buffer `opencl_program_source`. The file `source_file_name` can be located in the current directory, or in the directory specified by the environment variable STARPU_OPENCL_PROGRAM_DIR, or in the directory `share/starpu/opencl` of the installation directory of StarPU, or in the source directory of StarPU. When the file is found, `located_file_name` is the full name of the file as it has been located on the system, `located_dir_name` the directory where it has been located. Otherwise, they are both set to the empty string.

17.19.4.11 int starpu_opencl_compile_opencl_from_file (const char * *source_file_name*, const char * *build_options*)

Compile the OpenCL kernel stored in the file `source_file_name` with the given options `build_options` and stores the result in the directory `$STARPU_HOME/.starpu/opencl` with the same filename as `source_`

file_name. The compilation is done for every OpenCL device, and the filename is suffixed with the vendor id and the device id of the OpenCL device.

17.19.4.12 int starpu_opencl_compile_opencl_from_string (const char * *opencl_program_source,* const char * *file_name,* const char * *build_options*)

Compile the OpenCL kernel in the string opencl_program_source with the given options build_options and stores the result in the directory $STARPU_HOME/.starpu/opencl with the filename file_name. The compilation is done for every OpenCL device, and the filename is suffixed with the vendor id and the device id of the OpenCL device.

17.19.4.13 int starpu_opencl_load_binary_opencl (const char * *kernel_id,* struct **starpu_opencl_program** * *opencl_programs*)

Compile the binary OpenCL kernel identified with kernel_id. For every OpenCL device, the binary OpenCL kernel will be loaded from the file $STARPU_HOME/.starpu/opencl/<kernel_id>.<device_type>.vendor_id_<vendor_id>_device_id_<device_id>.

17.19.4.14 int starpu_opencl_load_kernel (cl_kernel * *kernel,* cl_command_queue * *queue,* struct **starpu_opencl_program** * *opencl_programs,* const char * *kernel_name,* int *devid*)

Create a kernel kernel for device devid, on its computation command queue returned in queue, using program opencl_programs and name kernel_name.

17.19.4.15 int starpu_opencl_release_kernel (cl_kernel *kernel*)

Release the given kernel, to be called after kernel execution.

17.19.4.16 int starpu_opencl_collect_stats (cl_event *event*)

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

17.19.4.17 const char * starpu_opencl_error_string (cl_int *status*)

Return the error message in English corresponding to status, an OpenCL error code.

17.19.4.18 void starpu_opencl_display_error (const char * *func,* const char * *file,* int *line,* const char * *msg,* cl_int *status*)

Given a valid error status, prints the corresponding error message on stdout, along with the given function name func, the given filename file, the given line number line and the given message msg.

17.19.4.19 void starpu_opencl_report_error (const char * *func,* const char * *file,* int *line,* const char * *msg,* cl_int *status*) [static]

Call the function starpu_opencl_display_error() and abort.

17.19.4.20 cl_int starpu_opencl_allocate_memory (cl_mem * *addr,* size_t *size,* cl_mem_flags *flags*)

Allocate size bytes of memory, stored in addr. flags must be a valid combination of cl_mem_flags values.

17.19.4.21 cl_int starpu_opencl_copy_ram_to_opencl (void * *ptr*, unsigned *src_node*, cl_mem *buffer*, unsigned *dst_node*, size_t *size*, size_t *offset*, cl_event * *event*, int * *ret*)

Copy `size` bytes from the given `ptr` on RAM `src_node` to the given `buffer` on OpenCL `dst_node`. `offset` is the offset, in bytes, in `buffer`. if `event` is `NULL`, the copy is synchronous, i.e the queue is synchronised before returning. If not `NULL`, `event` can be used after the call to wait for this particular copy to complete. This function returns `CL_SUCCESS` if the copy was successful, or a valid OpenCL error code otherwise. The integer pointed to by `ret` is set to `-EAGAIN` if the asynchronous launch was successful, or to 0 if `event` was `NULL`.

17.19.4.22 cl_int starpu_opencl_copy_opencl_to_ram (cl_mem *buffer*, unsigned *src_node*, void * *ptr*, unsigned *dst_node*, size_t *size*, size_t *offset*, cl_event * *event*, int * *ret*)

Copy `size` bytes asynchronously from the given `buffer` on OpenCL `src_node` to the given `ptr` on RAM `dst_node`. `offset` is the offset, in bytes, in `buffer`. if `event` is `NULL`, the copy is synchronous, i.e the queue is synchronised before returning. If not `NULL`, `event` can be used after the call to wait for this particular copy to complete. This function returns `CL_SUCCESS` if the copy was successful, or a valid OpenCL error code otherwise. The integer pointed to by `ret` is set to `-EAGAIN` if the asynchronous launch was successful, or to 0 if `event` was `NULL`.

17.19.4.23 cl_int starpu_opencl_copy_opencl_to_opencl (cl_mem *src*, unsigned *src_node*, size_t *src_offset*, cl_mem *dst*, unsigned *dst_node*, size_t *dst_offset*, size_t *size*, cl_event * *event*, int * *ret*)

Copy `size` bytes asynchronously from byte offset `src_offset` of `src` on OpenCL `src_node` to byte offset `dst_offset` of `dst` on OpenCL `dst_node`. if `event` is `NULL`, the copy is synchronous, i.e. the queue is synchronised before returning. If not `NULL`, `event` can be used after the call to wait for this particular copy to complete. This function returns `CL_SUCCESS` if the copy was successful, or a valid OpenCL error code otherwise. The integer pointed to by `ret` is set to `-EAGAIN` if the asynchronous launch was successful, or to 0 if `event` was `NULL`.

17.19.4.24 cl_int starpu_opencl_copy_async_sync (uintptr_t *src*, size_t *src_offset*, unsigned *src_node*, uintptr_t *dst*, size_t *dst_offset*, unsigned *dst_node*, size_t *size*, cl_event * *event*)

Copy `size` bytes from byte offset `src_offset` of `src` on `src_node` to byte offset `dst_offset` of `dst` on `dst_node`. if `event` is `NULL`, the copy is synchronous, i.e. the queue is synchronised before returning. If not `NULL`, `event` can be used after the call to wait for this particular copy to complete. The function returns `-EAGAIN` if the asynchronous launch was successfull. It returns 0 if the synchronous copy was successful, or fails otherwise.

17.20 Miscellaneous Helpers

Functions

- int starpu_data_cpy (starpu_data_handle_t dst_handle, starpu_data_handle_t src_handle, int asynchronous, void(*callback_func)(void *), void *callback_arg)
- void starpu_execute_on_each_worker (void(*func)(void *), void *arg, uint32_t where)

17.20.1 Detailed Description

17.20.2 Function Documentation

17.20.2.1 int starpu_data_cpy (starpu_data_handle_t *dst_handle*, starpu_data_handle_t *src_handle*, int *asynchronous*, void(∗)(void ∗) *callback_func*, void ∗ *callback_arg*)

Copy the content of `src_handle` into `dst_handle`. The parameter `asynchronous` indicates whether the function should block or not. In the case of an asynchronous call, it is possible to synchronize with the termination of this operation either by the means of implicit dependencies (if enabled) or by calling starpu_task_wait_for_all(). If `callback_func` is not NULL, this callback function is executed after the handle has been copied, and it is given the pointer `callback_arg` as argument.

17.20.2.2 void starpu_execute_on_each_worker (void(∗)(void ∗) *func*, void ∗ *arg*, uint32_t *where*)

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

17.21 FxT Support

Data Structures

- struct starpu_fxt_codelet_event
- struct starpu_fxt_options

Functions

- void starpu_fxt_options_init (struct starpu_fxt_options ∗options)
- void starpu_fxt_generate_trace (struct starpu_fxt_options ∗options)
- void starpu_fxt_start_profiling (void)
- void starpu_fxt_stop_profiling (void)

17.21.1 Detailed Description

17.21.2 Data Structure Documentation

17.21.2.1 struct starpu_fxt_codelet_event

todo

Data Fields

char	symbol[256]	name of the codelet
int	workerid	
enum starpu_↩ perfmodel_↩ archtype	archtype	

uint32_t	hash	
size_t	size	
float	time	

17.21.2.2 struct starpu_fxt_options

todo

Data Fields

unsigned	per_task_colour	
unsigned	no_counter	
unsigned	no_bus	
unsigned	ninputfiles	
char *	filenames[STARPU_FXT_MAX_FILES]	
char *	out_paje_path	
char *	distrib_time_path	
char *	activity_path	
char *	dag_path	
char *	file_prefix	In case we are going to gather multiple traces (e.g in the case of MPI processes), we may need to prefix the name of the containers.
uint64_t	file_offset	In case we are going to gather multiple traces (e.g in the case of MPI processes), we may need to prefix the name of the containers.
int	file_rank	In case we are going to gather multiple traces (e.g in the case of MPI processes), we may need to prefix the name of the containers.
char	worker_names[STARPU_NMAXWORKERS][256]	Output parameters
enum starpu_perfmodel_archtype	worker_archtypes[STARPU_NMAXWORKERS]	Output parameters
int	nworkers	Output parameters
struct starpu_fxt_codelet_event **	dumped_codelets	In case we want to dump the list of codelets to an external tool
long	dumped_codelets_count	In case we want to dump the list of codelets to an external tool

17.21.3 Function Documentation

17.21.3.1 void starpu_fxt_options_init (struct starpu_fxt_options * *options*)

todo

17.21.3.2 void starpu_fxt_generate_trace (struct starpu_fxt_options * *options*)

todo

17.21.3.3 void starpu_fxt_start_profiling (void)

Start recording the trace. The trace is by default started from starpu_init() call, but can be paused by using starpu_fxt_stop_profiling(), in which case starpu_fxt_start_profiling() should be called to resume recording events.

17.21.3.4 void starpu_fxt_stop_profiling (void)

Stop recording the trace. The trace is by default stopped when calling starpu_shutdown(). starpu_fxt_stop_profiling() can however be used to stop it earlier. starpu_fxt_start_profiling() can then be called to start recording it again, etc.

17.22 FFT Support

Functions

- void * starpufft_malloc (size_t n)
- void starpufft_free (void *p)
- starpufft_plan starpufft_plan_dft_1d (int n, int sign, unsigned flags)
- starpufft_plan starpufft_plan_dft_2d (int n, int m, int sign, unsigned flags)
- struct starpu_task * starpufft_start (starpufft_plan p, void *in, void *out)
- struct starpu_task * starpufft_start_handle (starpufft_plan p, starpu_data_handle_t in, starpu_data_handle_t out)
- int starpufft_execute (starpufft_plan p, void *in, void *out)
- int starpufft_execute_handle (starpufft_plan p, starpu_data_handle_t in, starpu_data_handle_t out)
- void starpufft_cleanup (starpufft_plan p)
- void starpufft_destroy_plan (starpufft_plan p)

17.22.1 Detailed Description

17.22.2 Function Documentation

17.22.2.1 void * starpufft_malloc (size_t *n*)

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

17.22.2.2 void * starpufft_free (void * *p*)

Release memory previously allocated.

17.22.2.3 struct starpufft_plan * starpufft_plan_dft_1d (int *n*, int *sign*, unsigned *flags*)

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

17.22.2.4 struct starpufft_plan * starpufft_plan_dft_2d (int *n*, int *m*, int *sign*, unsigned *flags*)

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

17.22.2.5 struct starpu_task * starpufft_start (starpufft_plan *p*, void * *in*, void * *out*)

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

17.22.2.6 struct starpu_task * starpufft_start_handle (starpufft_plan *p*, starpu_data_handle_t *in*, starpu_data_handle_t *out*)

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

17.22.2.7 void starpufft_execute (starpufft_plan *p*, void * *in*, void * *out*)

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

17.22.2.8 void starpufft_execute_handle (starpufft_plan *p*, starpu_data_handle_t *in*, starpu_data_handle_t *out*)

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

17.22.2.9 void starpufft_cleanup (starpufft_plan *p*)

Releases data for plan p, in the starpufft_start() case.

17.22.2.10 void starpufft_destroy_plan (starpufft_plan *p*)

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

17.23 MPI Support

Initialisation

- #define STARPU_USE_MPI
- int starpu_mpi_init (int *argc, char ***argv, int initialize_mpi)
- int starpu_mpi_initialize (void)
- int starpu_mpi_initialize_extended (int *rank, int *world_size)
- int starpu_mpi_shutdown (void)
- void starpu_mpi_comm_amounts_retrieve (size_t *comm_amounts)

Communication

- int starpu_mpi_send (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm)
- int starpu_mpi_recv (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, MPI_↩ Status *status)
- int starpu_mpi_isend (starpu_data_handle_t data_handle, starpu_mpi_req *req, int dest, int mpi_tag, MPI↩ _Comm comm)
- int starpu_mpi_irecv (starpu_data_handle_t data_handle, starpu_mpi_req *req, int source, int mpi_tag, M↩ PI_Comm comm)
- int starpu_mpi_isend_detached (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm, void(*callback)(void *), void *arg)

- int starpu_mpi_irecv_detached (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, void(*callback)(void *), void *arg)
- int starpu_mpi_wait (starpu_mpi_req *req, MPI_Status *status)
- int starpu_mpi_test (starpu_mpi_req *req, int *flag, MPI_Status *status)
- int starpu_mpi_barrier (MPI_Comm comm)
- int starpu_mpi_isend_detached_unlock_tag (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm, starpu_tag_t tag)
- int starpu_mpi_irecv_detached_unlock_tag (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, starpu_tag_t tag)
- int starpu_mpi_isend_array_detached_unlock_tag (unsigned array_size, starpu_data_handle_t *data_handle, int *dest, int *mpi_tag, MPI_Comm *comm, starpu_tag_t tag)
- int starpu_mpi_irecv_array_detached_unlock_tag (unsigned array_size, starpu_data_handle_t *data_handle, int *source, int *mpi_tag, MPI_Comm *comm, starpu_tag_t tag)

Communication Cache

- void starpu_mpi_cache_flush (MPI_Comm comm, starpu_data_handle_t data_handle)
- void starpu_mpi_cache_flush_all_data (MPI_Comm comm)

MPI Insert Task

- #define STARPU_EXECUTE_ON_NODE
- #define STARPU_EXECUTE_ON_DATA
- int starpu_data_set_tag (starpu_data_handle_t handle, int tag)
- int starpu_data_get_tag (starpu_data_handle_t handle)
- void starpu_mpi_data_register (starpu_data_handle_t data_handle, int tag, int rank)
- int starpu_data_set_rank (starpu_data_handle_t handle, int rank)
- int starpu_data_get_rank (starpu_data_handle_t handle)
- int starpu_mpi_insert_task (MPI_Comm comm, struct starpu_codelet *codelet,...)
- void starpu_mpi_get_data_on_node (MPI_Comm comm, starpu_data_handle_t data_handle, int node)
- void starpu_mpi_get_data_on_node_detached (MPI_Comm comm, starpu_data_handle_t data_handle, int node, void(*callback)(void *), void *arg)

Collective Operations

- void starpu_mpi_redux_data (MPI_Comm comm, starpu_data_handle_t data_handle)
- int starpu_mpi_scatter_detached (starpu_data_handle_t *data_handles, int count, int root, MPI_Comm comm, void(*scallback)(void *), void *sarg, void(*rcallback)(void *), void *rarg)
- int starpu_mpi_gather_detached (starpu_data_handle_t *data_handles, int count, int root, MPI_Comm comm, void(*scallback)(void *), void *sarg, void(*rcallback)(void *), void *rarg)

17.23.1 Detailed Description

17.23.2 Macro Definition Documentation

17.23.2.1 #define STARPU_USE_MPI

This macro is defined when StarPU has been installed with MPI support. It should be used in your code to detect the availability of MPI.

17.23.2.2 #define STARPU_EXECUTE_ON_NODE

this macro is used when calling starpu_mpi_insert_task(), and must be followed by a integer value which specified the node on which to execute the codelet.

17.23.2.3 #define STARPU_EXECUTE_ON_DATA

this macro is used when calling starpu_mpi_insert_task(), and must be followed by a data handle to specify that the node owning the given data will execute the codelet.

17.23.3 Function Documentation

17.23.3.1 int starpu_mpi_init (int * *argc*, char *** *argv*, int *initialize_mpi*)

Initializes the starpumpi library. `initialize_mpi` indicates if MPI should be initialized or not by StarPU. If the value is not 0, MPI will be initialized by calling `MPI_Init_Thread(argc, argv, MPI_THREAD_SERIALIZED, ...)`.

17.23.3.2 int starpu_mpi_initialize (void)

Deprecated This function has been made deprecated. One should use instead the function starpu_mpi_init(). This function does not call MPI_Init(), it should be called beforehand.

17.23.3.3 int starpu_mpi_initialize_extended (int * *rank*, int * *world_size*)

Deprecated This function has been made deprecated. One should use instead the function starpu_mpi_init(). MPI will be initialized by starpumpi by calling `MPI_Init_Thread(argc, argv, MPI_THREAD_SERIALIZED, ...)`.

17.23.3.4 int starpu_mpi_shutdown (void)

Cleans the starpumpi library. This must be called between calling starpu_mpi functions and starpu_shutdown(). MPI_Finalize() will be called if StarPU-MPI has been initialized by starpu_mpi_init().

17.23.3.5 void starpu_mpi_comm_amounts_retrieve (size_t * *comm_amounts*)

Retrieve the current amount of communications from the current node in the array `comm_amounts` which must have a size greater or equal to the world size. Communications statistics must be enabled (see STARPU_COMM_STATS).

17.23.3.6 int starpu_mpi_send (starpu_data_handle_t *data_handle*, int *dest*, int *mpi_tag*, MPI_Comm *comm*)

Performs a standard-mode, blocking send of `data_handle` to the node `dest` using the message tag `mpi_tag` within the communicator `comm`.

17.23.3.7 int starpu_mpi_recv (starpu_data_handle_t *data_handle*, int *source*, int *mpi_tag*, MPI_Comm *comm*, MPI_Status * *status*)

Performs a standard-mode, blocking receive in `data_handle` from the node `source` using the message tag `mpi_tag` within the communicator `comm`.

17.23.3.8 int starpu_mpi_isend (starpu_data_handle_t *data_handle,* starpu_mpi_req * *req,* int *dest,* int *mpi_tag,* MPI_Comm *comm*)

Posts a standard-mode, non blocking send of `data_handle` to the node `dest` using the message tag `mpi_tag` within the communicator `comm`. After the call, the pointer to the request `req` can be used to test or to wait for the completion of the communication.

17.23.3.9 int starpu_mpi_irecv (starpu_data_handle_t *data_handle,* starpu_mpi_req * *req,* int *source,* int *mpi_tag,* MPI_Comm *comm*)

Posts a nonblocking receive in `data_handle` from the node `source` using the message tag `mpi_tag` within the communicator `comm`. After the call, the pointer to the request `req` can be used to test or to wait for the completion of the communication.

17.23.3.10 int starpu_mpi_isend_detached (starpu_data_handle_t *data_handle,* int *dest,* int *mpi_tag,* MPI_Comm *comm,* void(*)(void *) *callback,* void * *arg*)

Posts a standard-mode, non blocking send of `data_handle` to the node `dest` using the message tag `mpi_tag` within the communicator `comm`. On completion, the `callback` function is called with the argument `arg`. Similarly to the pthread detached functionality, when a detached communication completes, its resources are automatically released back to the system, there is no need to test or to wait for the completion of the request.

17.23.3.11 int starpu_mpi_irecv_detached (starpu_data_handle_t *data_handle,* int *source,* int *mpi_tag,* MPI_Comm *comm,* void(*)(void *) *callback,* void * *arg*)

Posts a nonblocking receive in `data_handle` from the node `source` using the message tag `mpi_tag` within the communicator `comm`. On completion, the `callback` function is called with the argument `arg`. Similarly to the pthread detached functionality, when a detached communication completes, its resources are automatically released back to the system, there is no need to test or to wait for the completion of the request.

17.23.3.12 int starpu_mpi_wait (starpu_mpi_req * *req,* MPI_Status * *status*)

Returns when the operation identified by request `req` is complete.

17.23.3.13 int starpu_mpi_test (starpu_mpi_req * *req,* int * *flag,* MPI_Status * *status*)

If the operation identified by `req` is complete, set `flag` to 1. The `status` object is set to contain information on the completed operation.

17.23.3.14 int starpu_mpi_barrier (MPI_Comm *comm*)

Blocks the caller until all group members of the communicator `comm` have called it.

17.23.3.15 int starpu_mpi_isend_detached_unlock_tag (starpu_data_handle_t *data_handle,* int *dest,* int *mpi_tag,* MPI_Comm *comm,* starpu_tag_t *tag*)

Posts a standard-mode, non blocking send of `data_handle` to the node `dest` using the message tag `mpi_tag` within the communicator `comm`. On completion, `tag` is unlocked.

17.23.3.16 int starpu_mpi_irecv_detached_unlock_tag (starpu_data_handle_t *data_handle*, int *source*, int *mpi_tag*, MPI_Comm *comm*, starpu_tag_t *tag*)

Posts a nonblocking receive in `data_handle` from the node `source` using the message tag `mpi_tag` within the communicator `comm`. On completion, `tag` is unlocked.

17.23.3.17 int starpu_mpi_isend_array_detached_unlock_tag (unsigned *array_size*, starpu_data_handle_t * *data_handle*, int * *dest*, int * *mpi_tag*, MPI_Comm * *comm*, starpu_tag_t *tag*)

Posts `array_size` standard-mode, non blocking send. Each post sends the n-th data of the array `data_handle` to the n-th node of the array `dest` using the n-th message tag of the array `mpi_tag` within the n-th communicator of the array `comm`. On completion of the all the requests, `tag` is unlocked.

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

Posts `array_size` nonblocking receive. Each post receives in the n-th data of the array `data_handle` from the n-th node of the array `source` using the n-th message tag of the array `mpi_tag` within the n-th communicator of the array `comm`. On completion of the all the requests, `tag` is unlocked.

17.23.3.19 void starpu_mpi_cache_flush (MPI_Comm *comm*, starpu_data_handle_t *data_handle*)

Clear the send and receive communication cache for the data `data_handle` and invalidate the value. The function has to be called synchronously by all the MPI nodes. The function does nothing if the cache mechanism is disabled (see STARPU_MPI_CACHE).

17.23.3.20 void starpu_mpi_cache_flush_all_data (MPI_Comm *comm*)

Clear the send and receive communication cache for all data and invalidate their values. The function has to be called synchronously by all the MPI nodes. The function does nothing if the cache mechanism is disabled (see STARPU_MPI_CACHE).

17.23.3.21 int starpu_data_set_tag (starpu_data_handle_t *handle*, int *tag*)

Tell StarPU-MPI which MPI tag to use when exchanging the data.

17.23.3.22 int starpu_data_get_tag (starpu_data_handle_t *handle*)

Returns the MPI tag to be used when exchanging the data.

17.23.3.23 int starpu_mpi_data_register (starpu_data_handle_t *handle*, int *tag*, int *rank*)

Calling this function should be prefered to calling both starpu_data_set_rank() and starpu_data_set_tag() as it also allows to automatically clear the MPI communication cache when unregistering the data.

17.23.3.24 int starpu_data_set_rank (starpu_data_handle_t *handle*, int *rank*)

Tell StarPU-MPI which MPI node "owns" a given data, that is, the node which will always keep an up-to-date value, and will by default execute tasks which write to it.

17.23.3.25 int starpu_data_get_rank (starpu_data_handle_t *handle*)

Returns the last value set by starpu_data_set_rank().

17.23.3.26 int starpu_mpi_insert_task (MPI_Comm *comm*, struct starpu_codelet * *codelet*, ...)

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

The arguments following the codelets are the same types as for the function starpu_insert_task(). The extra argument STARPU_EXECUTE_ON_NODE followed by an integer allows to specify the MPI node to execute the codelet. It is also possible to specify that the node owning a specific data will execute the codelet, by using STARPU_EXECUTE_ON_DATA followed by a data handle.

The internal algorithm is as follows:

1. Find out which MPI node is going to execute the codelet.
   - If there is only one node owning data in STARPU_W mode, it will be selected;
   - If there is several nodes owning data in STARPU_W node, the one selected will be the one having the least data in R mode so as to minimize the amount of data to be transfered;
   - The argument STARPU_EXECUTE_ON_NODE followed by an integer can be used to specify the node;
   - The argument STARPU_EXECUTE_ON_DATA followed by a data handle can be used to specify that the node owing the given data will execute the codelet.
2. Send and receive data as requested. Nodes owning data which need to be read by the task are sending them to the MPI node which will execute it. The latter receives them.
3. Execute the codelet. This is done by the MPI node selected in the 1st step of the algorithm.
4. If several MPI nodes own data to be written to, send written data back to their owners.

The algorithm also includes a communication cache mechanism that allows not to send data twice to the same MPI node, unless the data has been modified. The cache can be disabled (see STARPU_MPI_CACHE).

17.23.3.27 void starpu_mpi_get_data_on_node (MPI_Comm *comm*, starpu_data_handle_t *data_handle*, int *node*)

Transfer data `data_handle` to MPI node `node`, sending it from its owner if needed. At least the target node and the owner have to call the function.

17.23.3.28 void starpu_mpi_get_data_on_node_detached (MPI_Comm *comm*, starpu_data_handle_t *data_handle*, int *node*, void(*)(void *) *callback*, void * *arg*)

Transfer data `data_handle` to MPI node `node`, sending it from its owner if needed. At least the target node and the owner have to call the function. On reception, the `callback` function is called with the argument `arg`.

17.23.3.29 void starpu_mpi_redux_data (MPI_Comm *comm*, starpu_data_handle_t *data_handle*)

Perform a reduction on the given data. All nodes send the data to its owner node which will perform a reduction.

17.23.3.30 int starpu_mpi_scatter_detached (starpu_data_handle_t * *data_handles*, int *count*, int *root*, MPI_Comm *comm*, void(*)(void *) *scallback*, void * *sarg*, void(*)(void *) *rcallback*, void * *rarg*)

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

function is called with the argument `sarg` on the process `root`, the `rcallback` function is called with the argument `rarg` on any other process.

17.23.3.31 int starpu_mpi_gather_detached (starpu_data_handle_t * *data_handles*, int *count*, int *root*, MPI_Comm *comm*, void(*)(void *) *scallback*, void * *sarg*, void(*)(void *) *rcallback*, void * *rarg*)

Gather data from the different processes of the communicator onto the process `root`. Each process owning data handle in the array `data_handles` will send them to the process `root`. The process `root` must have valid data handles to receive the data. On completion of the collective communication, the `rcallback` function is called with the argument `rarg` on the process root, the `scallback` function is called with the argument `sarg` on any other process.

17.24 Task Bundles

Typedefs

- typedef struct _starpu_task_bundle * starpu_task_bundle_t

Functions

- void starpu_task_bundle_create (starpu_task_bundle_t *bundle)
- int starpu_task_bundle_insert (starpu_task_bundle_t bundle, struct starpu_task *task)
- int starpu_task_bundle_remove (starpu_task_bundle_t bundle, struct starpu_task *task)
- void starpu_task_bundle_close (starpu_task_bundle_t bundle)
- double starpu_task_bundle_expected_length (starpu_task_bundle_t bundle, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_bundle_expected_power (starpu_task_bundle_t bundle, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_bundle_expected_data_transfer_time (starpu_task_bundle_t bundle, unsigned memory_node)

17.24.1 Detailed Description

17.24.2 Typedef Documentation

17.24.2.1 starpu_task_bundle_t

Opaque structure describing a list of tasks that should be scheduled on the same worker whenever it's possible. It must be considered as a hint given to the scheduler as there is no guarantee that they will be executed on the same worker.

17.24.3 Function Documentation

17.24.3.1 void starpu_task_bundle_create (starpu_task_bundle_t * *bundle*)

Factory function creating and initializing `bundle`, when the call returns, memory needed is allocated and `bundle` is ready to use.

17.24.3.2 int starpu_task_bundle_insert (starpu_task_bundle_t *bundle*, struct starpu_task * *task*)

Insert `task` in `bundle`. Until `task` is removed from `bundle` its expected length and data transfer time will be considered along those of the other tasks of bundle. This function must not be called if `bundle` is already closed and/or `task` is already submitted. On success, it returns 0. There are two cases of error : if `bundle` is already closed it returns `-EPERM`, if `task` was already submitted it returns `-EINVAL`.

17.24.3.3 int starpu_task_bundle_remove (starpu_task_bundle_t *bundle*, struct starpu_task * *task*)

Remove `task` from `bundle`. Of course `task` must have been previously inserted in `bundle`. This function must not be called if `bundle` is already closed and/or `task` is already submitted. Doing so would result in undefined behaviour. On success, it returns 0. If `bundle` is already closed it returns `-ENOENT`.

17.24.3.4 void starpu_task_bundle_close (starpu_task_bundle_t *bundle*)

Inform the runtime that the user will not modify `bundle` anymore, it means no more inserting or removing task. Thus the runtime can destroy it when possible.

17.24.3.5 double starpu_task_bundle_expected_length (starpu_task_bundle_t *bundle*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Return the expected duration of `bundle` in micro-seconds.

17.24.3.6 double starpu_task_bundle_expected_power (starpu_task_bundle_t *bundle*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Return the expected power consumption of `bundle` in J.

17.24.3.7 double starpu_task_bundle_expected_data_transfer_time (starpu_task_bundle_t *bundle*, unsigned *memory_node*)

Return the time (in micro-seconds) expected to transfer all data used within `bundle`.

17.25 Task Lists

Data Structures

- struct starpu_task_list

Functions

- static STARPU_INLINE void starpu_task_list_init (struct starpu_task_list *list)
- static STARPU_INLINE void starpu_task_list_push_front (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE void starpu_task_list_push_back (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE struct starpu_task * starpu_task_list_front (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_back (struct starpu_task_list *list)
- static STARPU_INLINE int starpu_task_list_empty (struct starpu_task_list *list)

- static STARPU_INLINE void starpu_task_list_erase (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE struct starpu_task * starpu_task_list_pop_front (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_pop_back (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_begin (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_next (struct starpu_task *task)

17.25.1 Detailed Description

17.25.2 Data Structure Documentation

17.25.2.1 struct starpu_task_list

Stores a double-chained list of tasks

Data Fields

struct starpu_task *	head	head of the list
struct starpu_task *	tail	tail of the list

17.25.3 Function Documentation

17.25.3.1 void starpu_task_list_init (struct starpu_task_list * *list*) [static]

Initialize a list structure

17.25.3.2 void starpu_task_list_push_front (struct starpu_task_list * *list*, struct starpu_task * *task*) [static]

Push `task` at the front of `list`

17.25.3.3 void starpu_task_list_push_back (struct starpu_task_list * *list*, struct starpu_task * *task*) [static]

Push `task` at the back of `list`

17.25.3.4 struct starpu_task * starpu_task_list_front (struct starpu_task_list * *list*) [static]

Get the front of `list` (without removing it)

17.25.3.5 struct starpu_task * starpu_task_list_back (struct starpu_task_list * *list*) [static]

Get the back of `list` (without removing it)

17.25.3.6 int starpu_task_list_empty (struct starpu_task_list * *list*) [static]

Test if `list` is empty

17.25.3.7 void starpu_task_list_erase (struct starpu_task_list * *list*, struct starpu_task * *task*) [static]

Remove `task` from `list`

17.25.3.8 struct starpu_task * starpu_task_list_pop_front (struct starpu_task_list * *list*) [static]

Remove the element at the front of `list`

17.25.3.9 struct starpu_task * starpu_task_list_pop_back (struct starpu_task_list * *list*) [static]

Remove the element at the back of `list`

17.25.3.10 struct starpu_task * starpu_task_list_begin (struct starpu_task_list * *list*) [static]

Get the first task of `list`.

17.25.3.11 struct starpu_task * starpu_task_list_next (struct starpu_task * *task*) [static]

Get the next task of `list`. This is not erase-safe.

17.26 Parallel Tasks

Functions

- int starpu_combined_worker_get_size (void)
- int starpu_combined_worker_get_rank (void)
- unsigned starpu_combined_worker_get_count (void)
- int starpu_combined_worker_get_id (void)
- int starpu_combined_worker_assign_workerid (int nworkers, int workerid_array[])
- int starpu_combined_worker_get_description (int workerid, int *worker_size, int **combined_workerid)
- int starpu_combined_worker_can_execute_task (unsigned workerid, struct starpu_task *task, unsigned nimpl)
- void starpu_parallel_task_barrier_init (struct starpu_task *task, int workerid)
- void starpu_parallel_task_barrier_init_n (struct starpu_task *task, int worker_size)

17.26.1 Detailed Description

17.26.2 Function Documentation

17.26.2.1 int starpu_combined_worker_get_size (void)

Return the size of the current combined worker, i.e. the total number of cpus running the same task in the case of STARPU_SPMD parallel tasks, or the total number of threads that the task is allowed to start in the case of STARPU_FORKJOIN parallel tasks.

17.26.2.2 int starpu_combined_worker_get_rank (void)

Return the rank of the current thread within the combined worker. Can only be used in STARPU_FORKJOIN parallel tasks, to know which part of the task to work on.

17.26.2.3 unsigned starpu_combined_worker_get_count (void)

Return the number of different combined workers.

17.26.2.4 int starpu_combined_worker_get_id (void)

Return the identifier of the current combined worker.

17.26.2.5 int starpu_combined_worker_assign_workerid (int *nworkers*, int *workerid_array[]*)

Register a new combined worker and get its identifier

17.26.2.6 int starpu_combined_worker_get_description (int *workerid*, int * *worker_size*, int ** *combined_workerid*)

Get the description of a combined worker

17.26.2.7 int starpu_combined_worker_can_execute_task (unsigned *workerid*, struct **starpu_task** * *task*, unsigned *nimpl*)

Variant of starpu_worker_can_execute_task() compatible with combined workers

17.26.2.8 void starpu_parallel_task_barrier_init (struct **starpu_task** * *task*, int *workerid*)

Initialise the barrier for the parallel task, and dispatch the task between the different workers of the given combined worker.

17.26.2.9 void starpu_parallel_task_barrier_init_n (struct **starpu_task** * *task*, int *worker_size*)

Initialise the barrier for the parallel task, to be pushed to *worker_size* workers (without having to explicit a given combined worker).

17.27 Running Drivers

Functions

- int starpu_driver_run (struct starpu_driver *d)
- int starpu_driver_init (struct starpu_driver *d)
- int starpu_driver_run_once (struct starpu_driver *d)
- int starpu_driver_deinit (struct starpu_driver *d)
- void starpu_drivers_request_termination (void)

17.27.1 Detailed Description

17.27.2 Function Documentation

17.27.2.1 int starpu_driver_run (struct **starpu_driver** * *d*)

Initialize the given driver, run it until it receives a request to terminate, deinitialize it and return 0 on success. It returns `-EINVAL` if `d->type` is not a valid StarPU device type (STARPU_CPU_WORKER, STARPU_CUDA_WORKER or STARPU_OPENCL_WORKER). This is the same as using the following functions: calling starpu_driver_init(), then calling starpu_driver_run_once() in a loop, and eventually starpu_driver_deinit().

17.27.2.2 int starpu_driver_init (struct starpu_driver * d)

Initialize the given driver. Returns 0 on success, -EINVAL if d->type is not a valid starpu_worker_archtype.

17.27.2.3 int starpu_driver_run_once (struct starpu_driver * d)

Run the driver once, then returns 0 on success, -EINVAL if d->type is not a valid starpu_worker_archtype.

17.27.2.4 int starpu_driver_deinit (struct starpu_driver * d)

Deinitialize the given driver. Returns 0 on success, -EINVAL if d->type is not a valid starpu_worker_archtype.

17.27.2.5 void starpu_drivers_request_termination (void)

Notify all running drivers they should terminate.

17.28 Expert Mode

Functions

- void starpu_wake_all_blocked_workers (void)
- int starpu_progression_hook_register (unsigned(*func)(void *arg), void *arg)
- void starpu_progression_hook_deregister (int hook_id)

17.28.1 Detailed Description

17.28.2 Function Documentation

17.28.2.1 void starpu_wake_all_blocked_workers (void)

Wake all the workers, so they can inspect data requests and task submissions again.

17.28.2.2 int starpu_progression_hook_register (unsigned(*)(void *arg) func, void * arg)

Register a progression hook, to be called when workers are idle.

17.28.2.3 void starpu_progression_hook_deregister (int hook_id)

Unregister a given progression hook.

17.29 StarPU-Top Interface

Data Structures

- struct starpu_top_data
- struct starpu_top_param

Enumerations

- enum starpu_top_data_type { STARPU_TOP_DATA_BOOLEAN, STARPU_TOP_DATA_INTEGER, STARPU_TOP_DATA_FLOAT }
- enum starpu_top_param_type { STARPU_TOP_PARAM_BOOLEAN, STARPU_TOP_PARAM_INTEGER, STARPU_TOP_PARAM_FLOAT, STARPU_TOP_PARAM_ENUM }
- enum starpu_top_message_type { TOP_TYPE_GO, TOP_TYPE_SET, TOP_TYPE_CONTINUE, TOP_TYPE_ENABLE, TOP_TYPE_DISABLE, TOP_TYPE_DEBUG, TOP_TYPE_UNKNOW }

Functions to call before the initialisation

- struct starpu_top_data * starpu_top_add_data_boolean (const char *data_name, int active)
- struct starpu_top_data * starpu_top_add_data_integer (const char *data_name, int minimum_value, int maximum_value, int active)
- struct starpu_top_data * starpu_top_add_data_float (const char *data_name, double minimum_value, double maximum_value, int active)
- struct starpu_top_param * starpu_top_register_parameter_boolean (const char *param_name, int *parameter_field, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_float (const char *param_name, double *parameter_field, double minimum_value, double maximum_value, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_integer (const char *param_name, int *parameter_field, int minimum_value, int maximum_value, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_enum (const char *param_name, int *parameter_field, char **values, int nb_values, void(*callback)(struct starpu_top_param *))

Initialisation

- void starpu_top_init_and_wait (const char *server_name)

To call after initialisation

- void starpu_top_update_parameter (const struct starpu_top_param *param)
- void starpu_top_update_data_boolean (const struct starpu_top_data *data, int value)
- void starpu_top_update_data_integer (const struct starpu_top_data *data, int value)
- void starpu_top_update_data_float (const struct starpu_top_data *data, double value)
- void starpu_top_task_prevision (struct starpu_task *task, int devid, unsigned long long start, unsigned long long end)
- void starpu_top_debug_log (const char *message)
- void starpu_top_debug_lock (const char *message)

17.29.1 Detailed Description

17.29.2 Data Structure Documentation

17.29.2.1 struct starpu_top_data

todo

Data Fields

unsigned int	id	todo
const char *	name	todo
int	int_min_value	todo
int	int_max_value	todo
double	double_min_value	todo
double	double_max_value	todo
int	active	todo
enum starpu_top_data_type	type	todo
struct starpu_top_data *	next	todo

17.29.2.2 struct starpu_top_param

todo

Data Fields

- unsigned int id
- const char * name
- enum starpu_top_param_type type
- void * value
- char ** enum_values
- int nb_values
- void(* callback)(struct starpu_top_param *)
- int int_min_value
- int int_max_value
- double double_min_value
- double double_max_value
- struct starpu_top_param * next

17.29.2.2.1 Field Documentation

17.29.2.2.1.1 starpu_top_param::id

todo

17.29.2.2.1.2 starpu_top_param::name

todo

17.29.2.2.1.3 starpu_top_param::type

todo

17.29.2.2.1.4 starpu_top_param::value

todo

17.29.2.2.1.5 starpu_top_param::enum_values

only for enum type can be NULL

17.29.2.2.1.6 starpu_top_param::nb_values

todo

17.29.2.2.1.7 starpu_top_param::callback

todo

17.29.2.2.1.8 starpu_top_param::int_min_value

only for integer type

17.29.2.2.1.9 starpu_top_param::int_max_value

todo

17.29.2.2.1.10 starpu_top_param::double_min_value

only for double type

17.29.2.2.1.11 starpu_top_param::double_max_value

todo

17.29.2.2.1.12 starpu_top_param::next

todo

17.29.3 Enumeration Type Documentation

17.29.3.1 enum starpu_top_data_type

StarPU-Top Data type

Enumerator

STARPU_TOP_DATA_BOOLEAN todo
STARPU_TOP_DATA_INTEGER todo
STARPU_TOP_DATA_FLOAT todo

17.29.3.2 enum starpu_top_param_type

StarPU-Top Parameter type

Enumerator

STARPU_TOP_PARAM_BOOLEAN todo
STARPU_TOP_PARAM_INTEGER todo
STARPU_TOP_PARAM_FLOAT todo
STARPU_TOP_PARAM_ENUM todo

17.29.3.3 enum starpu_top_message_type

StarPU-Top Message type

Enumerator

TOP_TYPE_GO todo

TOP_TYPE_SET todo

TOP_TYPE_CONTINUE todo

TOP_TYPE_ENABLE todo

TOP_TYPE_DISABLE todo

TOP_TYPE_DEBUG todo

TOP_TYPE_UNKNOW todo

17.29.4 Function Documentation

17.29.4.1 struct starpu_top_data * starpu_top_add_data_boolean (const char * *data_name,* int *active*)

This fonction register a data named data_name of type boolean. If `active=0`, the value will NOT be displayed to user by default. Any other value will make the value displayed by default.

17.29.4.2 struct starpu_top_data * starpu_top_add_data_integer (const char * *data_name,* int *minimum_value,* int *maximum_value,* int *active*)

This fonction register a data named `data_name` of type integer. The minimum and maximum value will be usefull to define the scale in UI. If `active=0`, the value will NOT be displayed to user by default. Any other value will make the value displayed by default.

17.29.4.3 struct starpu_top_data * starpu_top_add_data_float (const char * *data_name,* double *minimum_value,* double *maximum_value,* int *active*)

This fonction register a data named data_name of type float. The minimum and maximum value will be usefull to define the scale in UI. If `active=0`, the value will NOT be displayed to user by default. Any other value will make the value displayed by default.

17.29.4.4 struct starpu_top_param * starpu_top_register_parameter_boolean (const char * *param_name,* int * *parameter_field,* void(*)(struct starpu_top_param *) *callback*)

This fonction register a parameter named `parameter_name`, of type boolean. The `callback` fonction will be called when the parameter is modified by UI, and can be null.

17.29.4.5 struct starpu_top_param * starpu_top_register_parameter_float (const char * *param_name,* double * *parameter_field,* double *minimum_value,* double *maximum_value,* void(*)(struct starpu_top_param *) *callback*)

his fonction register a parameter named `param_name`, of type integer. Minimum and maximum value will be used to prevent user seting incorrect value. The `callback` fonction will be called when the parameter is modified by UI, and can be null.

17.29.4.6 struct starpu_top_param * starpu_top_register_parameter_integer (const char * *param_name,* int * *parameter_field,* int *minimum_value,* int *maximum_value,* void(*)(struct starpu_top_param *) *callback*)

This fonction register a parameter named `param_name`, of type float. Minimum and maximum value will be used to prevent user seting incorrect value. The `callback` fonction will be called when the parameter is modified by UI, and can be null.

17.29.4.7 struct starpu_top_param * starpu_top_register_parameter_enum (const char * *param_name,* int * *parameter_field,* char ** *values,* int *nb_values,* void(*)(struct starpu_top_param *) *callback*)

This fonction register a parameter named `param_name`, of type enum. Minimum and maximum value will be used to prevent user seting incorrect value. The `callback` fonction will be called when the parameter is modified by UI, and can be null.

17.29.4.8 void starpu_top_init_and_wait (const char * *server_name*)

This function must be called when all parameters and data have been registered AND initialised (for parameters). This function will wait for a TOP to connect, send initialisation sentences, and wait for the GO message.

17.29.4.9 void starpu_top_update_parameter (const struct starpu_top_param * *param*)

This function should be called after every modification of a parameter from something other than starpu_top. This fonction notice UI that the configuration changed.

17.29.4.10 void starpu_top_update_data_boolean (const struct starpu_top_data * *data,* int *value*)

This function updates the value of the starpu_top_data on UI.

17.29.4.11 void starpu_top_update_data_integer (const struct starpu_top_data * *data,* int *value*)

This function updates the value of the starpu_top_data on UI.

17.29.4.12 void starpu_top_update_data_float (const struct starpu_top_data * *data,* double *value*)

This function updates the value of the starpu_top_data on UI.

17.29.4.13 void starpu_top_task_prevision (struct starpu_task * *task,* int *devid,* unsigned long long *start,* unsigned long long *end*)

This function notifies UI than the task have been planed to run from start to end, on computation-core.

17.29.4.14 void starpu_top_debug_log (const char * *message*)

This function is useful in debug mode. The starpu developper doesn't need to check if the debug mode is active. This is checked by starpu_top itsefl. It just send a message to display by UI.

17.29.4.15 void starpu_top_debug_lock (const char * *message*)

This function is useful in debug mode. The starpu developper doesn't need to check if the debug mode is active. This is checked by starpu_top itsefl. It send a message and wait for a continue message from UI to return. The lock (wich create a stop-point) should be called only by the main thread. Calling it from more than one thread is not supported.

17.30 Scheduling Contexts

StarPU permits on one hand grouping workers in combined workers in order to execute a parallel task and on the other hand grouping tasks in bundles that will be executed by a single specified worker. In contrast when we group

workers in scheduling contexts we submit starpu tasks to them and we schedule them with the policy assigned to the context. Scheduling contexts can be created, deleted and modified dynamically.

Data Structures

- struct starpu_sched_ctx_performance_counters

Scheduling Contexts Basic API

- #define STARPU_SCHED_CTX_POLICY_NAME
- #define STARPU_SCHED_CTX_POLICY_STRUCT
- #define STARPU_SCHED_CTX_POLICY_MIN_PRIO
- #define STARPU_SCHED_CTX_POLICY_MAX_PRIO
- unsigned starpu_sched_ctx_create (int *workerids_ctx, int nworkers_ctx, const char *sched_ctx_name,...)
- unsigned starpu_sched_ctx_create_inside_interval (const char *policy_name, const char *sched_name, int min_ncpus, int max_ncpus, int min_ngpus, int max_ngpus, unsigned allow_overlap)
- void starpu_sched_ctx_register_close_callback (unsigned sched_ctx_id, void(*close_callback)(unsigned sched_ctx_id, void *args), void *args)
- void starpu_sched_ctx_add_workers (int *workerids_ctx, int nworkers_ctx, unsigned sched_ctx_id)
- void starpu_sched_ctx_remove_workers (int *workerids_ctx, int nworkers_ctx, unsigned sched_ctx_id)
- void starpu_sched_ctx_delete (unsigned sched_ctx_id)
- void starpu_sched_ctx_set_inheritor (unsigned sched_ctx_id, unsigned inheritor)
- void starpu_sched_ctx_set_context (unsigned *sched_ctx_id)
- unsigned starpu_sched_ctx_get_context (void)
- void starpu_sched_ctx_stop_task_submission (void)
- void starpu_sched_ctx_finished_submit (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_get_workers_list (unsigned sched_ctx_id, int **workerids)
- unsigned starpu_sched_ctx_get_nworkers (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_get_nshared_workers (unsigned sched_ctx_id, unsigned sched_ctx_id2)
- unsigned starpu_sched_ctx_contains_worker (int workerid, unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_worker_get_id (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_overlapping_ctxs_on_worker (int workerid)

Scheduling Context Priorities

- #define STARPU_MIN_PRIO
- #define STARPU_MAX_PRIO
- #define STARPU_DEFAULT_PRIO
- int starpu_sched_ctx_set_min_priority (unsigned sched_ctx_id, int min_prio)
- int starpu_sched_ctx_set_max_priority (unsigned sched_ctx_id, int max_prio)
- int starpu_sched_ctx_get_min_priority (unsigned sched_ctx_id)
- int starpu_sched_ctx_get_max_priority (unsigned sched_ctx_id)

Scheduling Context Worker Collection

- struct starpu_worker_collection * starpu_sched_ctx_create_worker_collection (unsigned sched_ctx_id, enum starpu_worker_collection_type type)
- void starpu_sched_ctx_delete_worker_collection (unsigned sched_ctx_id)
- struct starpu_worker_collection * starpu_sched_ctx_get_worker_collection (unsigned sched_ctx_id)

Scheduling Context Link with Hypervisor

- void starpu_sched_ctx_set_perf_counters (unsigned sched_ctx_id, void *perf_counters)
- void starpu_sched_ctx_call_pushed_task_cb (int workerid, unsigned sched_ctx_id)
- void starpu_sched_ctx_notify_hypervisor_exists (void)
- unsigned starpu_sched_ctx_check_if_hypervisor_exists (void)
- void starpu_sched_ctx_set_policy_data (unsigned sched_ctx_id, void *policy_data)
- void * starpu_sched_ctx_get_policy_data (unsigned sched_ctx_id)

17.30.1 Detailed Description

StarPU permits on one hand grouping workers in combined workers in order to execute a parallel task and on the other hand grouping tasks in bundles that will be executed by a single specified worker. In contrast when we group workers in scheduling contexts we submit starpu tasks to them and we schedule them with the policy assigned to the context. Scheduling contexts can be created, deleted and modified dynamically.

17.30.2 Data Structure Documentation

17.30.2.1 struct starpu_sched_ctx_performance_counters

Performance counters used by the starpu to indicate the hypervisor how the application and the resources are executing.

Data Fields

- void(* notify_idle_cycle)(unsigned sched_ctx_id, int worker, double idle_time)
- void(* notify_poped_task)(unsigned sched_ctx_id, int worker)
- void(* notify_pushed_task)(unsigned sched_ctx_id, int worker)
- void(* **notify_post_exec_task**)(struct starpu_task *task, size_t data_size, uint32_t footprint, int hypervisor↩ _tag, double flops)
- void(* notify_submitted_job)(struct starpu_task *task, uint32_t footprint, size_t data_size)
- void(* **notify_empty_ctx**)(unsigned sched_ctx_id, struct starpu_task *task)
- void(* notify_delete_context)(unsigned sched_ctx)

17.30.2.1.1 Field Documentation

17.30.2.1.1.1 starpu_sched_ctx_performance_counters::notify_idle_cycle

Informs the hypervisor for how long a worker has been idle in the specified context

17.30.2.1.1.2 starpu_sched_ctx_performance_counters::notify_poped_task

Informs the hypervisor that a task executing a specified number of instructions has been poped from the worker

17.30.2.1.1.3 starpu_sched_ctx_performance_counters::notify_pushed_task

Notifies the hypervisor that a task has been scheduled on the queue of the worker corresponding to the specified context

17.30.2.1.1.4 starpu_sched_ctx_performance_counters::notify_submitted_job

Notifies the hypervisor that a task has just been submitted

17.30.2.1.1.5 starpu_sched_ctx_performance_counters::notify_delete_context

Notifies the hypervisor that the context was deleted

17.30.3 Macro Definition Documentation

17.30.3.1 #define STARPU_SCHED_CTX_POLICY_NAME

This macro is used when calling starpu_sched_ctx_create() to specify a name for a scheduling policy

17.30.3.2 #define STARPU_SCHED_CTX_POLICY_STRUCT

This macro is used when calling starpu_sched_ctx_create() to specify a pointer to a scheduling policy

17.30.3.3 #define STARPU_SCHED_CTX_POLICY_MIN_PRIO

This macro is used when calling starpu_sched_ctx_create() to specify a minimum scheduler priority value.

17.30.3.4 #define STARPU_SCHED_CTX_POLICY_MAX_PRIO

This macro is used when calling starpu_sched_ctx_create() to specify a maximum scheduler priority value.

17.30.3.5 #define STARPU_MIN_PRIO

Provided for legacy reasons.

17.30.3.6 #define STARPU_MAX_PRIO

Provided for legacy reasons.

17.30.3.7 #define STARPU_DEFAULT_PRIO

By convention, the default priority level should be 0 so that we can statically allocate tasks with a default priority.

17.30.4 Function Documentation

17.30.4.1 unsigned starpu_sched_ctx_create (int * *workerids_ctx*, int *nworkers_ctx*, const char * *sched_ctx_name*, ...)

This function creates a scheduling context with the given parameters (see below) and assigns the workers in `workerids_ctx` to execute the tasks submitted to it. The return value represents the identifier of the context that has just been created. It will be further used to indicate the context the tasks will be submitted to. The return value should be at most STARPU_NMAX_SCHED_CTXS.

The arguments following the name of the scheduling context can be of the following types:

- STARPU_SCHED_CTX_POLICY_NAME, followed by the name of a predefined scheduling policy
- STARPU_SCHED_CTX_POLICY_STRUCT, followed by a pointer to a custom scheduling policy (struct starpu_sched_policy *)
- STARPU_SCHED_CTX_POLICY_MIN_PRIO, followed by a integer representing the minimum priority value to be defined for the scheduling policy.
- STARPU_SCHED_CTX_POLICY_MAX_PRIO, followed by a integer representing the maximum priority value to be defined for the scheduling policy.

17.30.4.2 unsigned starpu_sched_ctx_create_inside_interval (const char * *policy_name,* const char * *sched_ctx_name,* int *min_ncpus,* int *max_ncpus,* int *min_ngpus,* int *max_ngpus,* unsigned *allow_overlap*)

Create a context indicating an approximate interval of resources

17.30.4.3 void starpu_sched_ctx_register_close_callback (unsigned *sched_ctx_id,* void(*)(unsigned sched_ctx_id, void *args) *close_callback,* void * *args*)

Execute the callback whenever the last task of the context finished executing, it is called with the pramaters: sched↩ _ctx and any other paramter needed by the application (packed in a void*)

17.30.4.4 void starpu_sched_ctx_add_workers (int * *workerids_ctx,* int *nworkers_ctx,* unsigned *sched_ctx_id*)

This function adds dynamically the workers in `workerids_ctx` to the context `sched_ctx_id`. The last argument cannot be greater than STARPU_NMAX_SCHED_CTXS.

17.30.4.5 void starpu_sched_ctx_remove_workers (int * *workerids_ctx,* int *nworkers_ctx,* unsigned *sched_ctx_id*)

This function removes the workers in `workerids_ctx` from the context `sched_ctx_id`. The last argument cannot be greater than STARPU_NMAX_SCHED_CTXS.

17.30.4.6 void starpu_sched_ctx_delete (unsigned *sched_ctx_id*)

Delete scheduling context `sched_ctx_id` and transfer remaining workers to the inheritor scheduling context.

17.30.4.7 void starpu_sched_ctx_set_inheritor (unsigned *sched_ctx_id,* unsigned *inheritor*)

Indicate which context whill inherit the resources of this context when he will be deleted.

17.30.4.8 void starpu_sched_ctx_set_context (unsigned * *sched_ctx_id*)

Set the scheduling context the subsequent tasks will be submitted to

17.30.4.9 unsigned starpu_sched_ctx_get_context (void)

Return the scheduling context the tasks are currently submitted to, or ::STARPU_NMAX_SCHED_CTXS if no default context has been defined by calling the function starpu_sched_ctx_set_context().

17.30.4.10 void starpu_sched_ctx_stop_task_submission (void)

Stop submitting tasks from the empty context list until the next time the context has time to check the empty context list

17.30.4.11 void starpu_sched_ctx_finished_submit (unsigned *sched_ctx_id*)

Indicate starpu that the application finished submitting to this context in order to move the workers to the inheritor as soon as possible.

17.30.4.12 unsigned starpu_sched_ctx_get_workers_list (unsigned *sched_ctx_id,* int ** *workerids*)

Returns the list of workers in the array `workerids`, the returned value is the number of workers. The user should free the `workerids` table after finishing using it (it is allocated inside the function with the proper size)

17.30.4.13 unsigned starpu_sched_ctx_get_nworkers (unsigned *sched_ctx_id*)

Return the number of workers managed by the specified contexts (Usually needed to verify if it manages any workers or if it should be blocked)

17.30.4.14 unsigned starpu_sched_ctx_get_nshared_workers (unsigned *sched_ctx_id,* unsigned *sched_ctx_id2*)

Return the number of workers shared by two contexts.

17.30.4.15 unsigned starpu_sched_ctx_contains_worker (int *workerid,* unsigned *sched_ctx_id*)

Return 1 if the worker belongs to the context and 0 otherwise

17.30.4.16 unsigned starpu_sched_ctx_worker_get_id (unsigned *sched_ctx_id*)

Return the workerid if the worker belongs to the context and -1 otherwise. If the thread calling this function is not a worker the function returns -1 as it calls the function starpu_worker_get_id()

17.30.4.17 unsigned starpu_sched_ctx_overlapping_ctxs_on_worker (int *workerid*)

Check if a worker is shared between several contexts

17.30.4.18 int starpu_sched_ctx_set_min_priority (unsigned *sched_ctx_id,* int *min_prio*)

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

17.30.4.19 int starpu_sched_ctx_set_max_priority (unsigned *sched_ctx_id,* int *max_prio*)

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

17.30.4.20 int starpu_sched_ctx_get_min_priority (unsigned *sched_ctx_id*)

Returns the current minimum priority level supported by the scheduling policy of the given scheduler context.

17.30.4.21 int starpu_sched_ctx_get_max_priority (unsigned *sched_ctx_id*)

Returns the current maximum priority level supported by the scheduling policy of the given scheduler context.

17.30.4.22 struct starpu_worker_collection * starpu_sched_ctx_create_worker_collection (unsigned *sched_ctx_id,* enum starpu_worker_collection_type *type*)

Create a worker collection of the type indicated by the last parameter for the context specified through the first parameter.

17.30.4.23 void starpu_sched_ctx_delete_worker_collection (unsigned *sched_ctx_id*)

Delete the worker collection of the specified scheduling context

17.30.4.24 struct starpu_worker_collection * starpu_sched_ctx_get_worker_collection (unsigned *sched_ctx_id*)

Return the worker collection managed by the indicated context

17.30.4.25 void starpu_sched_ctx_set_perf_counters (unsigned *sched_ctx_id,* void * *perf_counters*)

Indicates to starpu the pointer to the performance counter

17.30.4.26 void starpu_sched_ctx_call_pushed_task_cb (int *workerid,* unsigned *sched_ctx_id*)

Callback that lets the scheduling policy tell the hypervisor that a task was pushed on a worker

17.30.4.27 void starpu_sched_ctx_notify_hypervisor_exists (void)

Allow the hypervisor to let starpu know he's initialised

17.30.4.28 unsigned starpu_sched_ctx_check_if_hypervisor_exists (void)

Ask starpu if he is informed if the hypervisor is initialised

17.30.4.29 void starpu_sched_ctx_set_policy_data (unsigned *sched_ctx_id,* void * *policy_data*)

Allocate the scheduling policy data (private information of the scheduler like queues, variables, additional condition variables) the context

17.30.4.30 void * starpu_sched_ctx_get_policy_data (unsigned *sched_ctx_id*)

Return the scheduling policy data (private information of the scheduler) of the contexts previously assigned to.

17.31 Scheduling Policy

TODO. While StarPU comes with a variety of scheduling policies (see Task Scheduling Policy), it may sometimes be desirable to implement custom policies to address specific problems. The API described below allows users to write their own scheduling policy.

Data Structures

- struct starpu_sched_policy

Functions

- struct starpu_sched_policy ** starpu_sched_get_predefined_policies ()
- void starpu_worker_get_sched_condition (int workerid, starpu_pthread_mutex_t **sched_mutex, starpu_↩ pthread_cond_t **sched_cond)
- int starpu_sched_set_min_priority (int min_prio)
- int starpu_sched_set_max_priority (int max_prio)
- int starpu_sched_get_min_priority (void)
- int starpu_sched_get_max_priority (void)
- int starpu_push_local_task (int workerid, struct starpu_task *task, int back)
- int starpu_push_task_end (struct starpu_task *task)
- int starpu_worker_can_execute_task (unsigned workerid, struct starpu_task *task, unsigned nimpl)
- double starpu_timing_now (void)
- uint32_t starpu_task_footprint (struct starpu_perfmodel *model, struct starpu_task *task, enum starpu_↩ perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_expected_length (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_worker_get_relative_speedup (enum starpu_perfmodel_archtype perf_archtype)
- double starpu_task_expected_data_transfer_time (unsigned memory_node, struct starpu_task *task)
- double starpu_data_expected_transfer_time (starpu_data_handle_t handle, unsigned memory_node, enum starpu_data_access_mode mode)
- double starpu_task_expected_power (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_expected_conversion_time (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- int starpu_get_prefetch_flag (void)
- int starpu_prefetch_task_input_on_node (struct starpu_task *task, unsigned node)
- void starpu_sched_ctx_worker_shares_tasks_lists (int workerid, int sched_ctx_id)

17.31.1 Detailed Description

TODO. While StarPU comes with a variety of scheduling policies (see Task Scheduling Policy), it may sometimes be desirable to implement custom policies to address specific problems. The API described below allows users to write their own scheduling policy.

17.31.2 Data Structure Documentation

17.31.2.1 struct starpu_sched_policy

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

Data Fields

- void(* init_sched)(unsigned sched_ctx_id)
- void(* deinit_sched)(unsigned sched_ctx_id)
- int(* push_task)(struct starpu_task *)
- void(* push_task_notify)(struct starpu_task *, int workerid, int perf_workerid, unsigned sched_ctx_id)
- struct starpu_task *(* pop_task)(unsigned sched_ctx_id)
- struct starpu_task *(* pop_every_task)(unsigned sched_ctx_id)
- void(* pre_exec_hook)(struct starpu_task *)
- void(* post_exec_hook)(struct starpu_task *)
- void(* add_workers)(unsigned sched_ctx_id, int *workerids, unsigned nworkers)
- void(* remove_workers)(unsigned sched_ctx_id, int *workerids, unsigned nworkers)
- const char * policy_name
- const char * policy_description

17.31.2.1.1 Field Documentation

17.31.2.1.1.1 starpu_sched_policy::init_sched

Initialize the scheduling policy.

17.31.2.1.1.2 starpu_sched_policy::deinit_sched

Cleanup the scheduling policy.

17.31.2.1.1.3 starpu_sched_policy::push_task

Insert a task into the scheduler.

17.31.2.1.1.4 starpu_sched_policy::push_task_notify

Notify the scheduler that a task was pushed on a given worker. This method is called when a task that was explicitly assigned to a worker becomes ready and is about to be executed by the worker. This method therefore permits to keep the state of the scheduler coherent even when StarPU bypasses the scheduling strategy.

17.31.2.1.1.5 starpu_sched_policy::pop_task

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

17.31.2.1.1.6 starpu_sched_policy::pop_every_task

Remove all available tasks from the scheduler (tasks are chained by the means of the field starpu_task::prev and starpu_task::next). The mutex associated to the worker is already taken when this method is called. This is currently not used.

17.31.2.1.1.7 starpu_sched_policy::pre_exec_hook

Optional field. This method is called every time a task is starting.

17.31.2.1.1.8 starpu_sched_policy::post_exec_hook

Optional field. This method is called every time a task has been executed.

17.31.2.1.1.9 starpu_sched_policy::add_workers

Initialize scheduling structures corresponding to each worker used by the policy.

17.31.2.1.1.10 starpu_sched_policy::remove_workers

Deinitialize scheduling structures corresponding to each worker used by the policy.

17.31.2.1.1.11 starpu_sched_policy::policy_name

Optional field. Name of the policy.

17.31.2.1.1.12 starpu_sched_policy::policy_description

Optional field. Human readable description of the policy.

17.31.3 Function Documentation

17.31.3.1 struct starpu_sched_policy ** starpu_sched_get_predefined_policies ( )

Return an NULL-terminated array of all the predefined scheduling policies.

17.31.3.2 void starpu_worker_get_sched_condition (int *workerid*, starpu_pthread_mutex_t ** *sched_mutex*, starpu_pthread_cond_t ** *sched_cond*)

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

17.31.3.3 int starpu_sched_set_min_priority (int *min_prio*)

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

17.31.3.4 int starpu_sched_set_max_priority (int *max_prio*)

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

17.31.3.5 int starpu_sched_get_min_priority (void)

TODO: check if this is correct Returns the current minimum priority level supported by the scheduling policy

17.31.3.6 int starpu_sched_get_max_priority (void)

TODO: check if this is correct Returns the current maximum priority level supported by the scheduling policy

17.31.3.7 int starpu_push_local_task (int *workerid*, struct starpu_task * *task*, int *back*)

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

17.31.3.8 int starpu_push_task_end (struct starpu_task * *task*)

This function must be called by a scheduler to notify that the given task has just been pushed.

17.31.3.9 int starpu_worker_can_execute_task (unsigned *workerid*, struct starpu_task * *task*, unsigned *nimpl*)

Check if the worker specified by workerid can execute the codelet. Schedulers need to call it before assigning a task to a worker, otherwise the task may fail to execute.

17.31.3.10 double starpu_timing_now (void)

Return the current date in micro-seconds.

17.31.3.11 uint32_t starpu_task_footprint (struct starpu_perfmodel * *model*, struct starpu_task * *task*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Returns the footprint for a given task

17.31.3.12 double starpu_task_expected_length (struct starpu_task * *task*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Returns expected task duration in micro-seconds.

17.31.3.13 double starpu_worker_get_relative_speedup (enum starpu_perfmodel_archtype *perf_archtype*)

Returns an estimated speedup factor relative to CPU speed

17.31.3.14 double starpu_task_expected_data_transfer_time (unsigned *memory_node*, struct starpu_task * *task*)

Returns expected data transfer time in micro-seconds.

17.31.3.15 double starpu_data_expected_transfer_time (starpu_data_handle_t *handle*, unsigned *memory_node*, enum starpu_data_access_mode *mode*)

Predict the transfer time (in micro-seconds) to move `handle` to a memory node

17.31.3.16 double starpu_task_expected_power (struct starpu_task * *task*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Returns expected power consumption in J

17.31.3.17 double starpu_task_expected_conversion_time (struct starpu_task * *task*, enum starpu_perfmodel_archtype *arch*, unsigned *nimpl*)

Returns expected conversion time in ms (multiformat interface only)

17.31.3.18 int starpu_get_prefetch_flag (void)

Whether STARPU_PREFETCH was set

17.31.3.19 int starpu_prefetch_task_input_on_node (struct starpu_task * *task*, unsigned *node*)

Prefetch data for a given task on a given node

17.31.3.20 void starpu_sched_ctx_worker_shares_tasks_lists (int *workerid*, int *sched_ctx_id*)

The scheduling policies indicates if the worker may pop tasks from the list of other workers or if there is a central list with task for all the workers

17.32 Scheduling Context Hypervisor - Regular usage

Macros

- #define SC_HYPERVISOR_MAX_IDLE
- #define SC_HYPERVISOR_PRIORITY
- #define SC_HYPERVISOR_MIN_WORKERS
- #define SC_HYPERVISOR_MAX_WORKERS
- #define SC_HYPERVISOR_GRANULARITY
- #define SC_HYPERVISOR_FIXED_WORKERS
- #define SC_HYPERVISOR_MIN_TASKS
- #define SC_HYPERVISOR_NEW_WORKERS_MAX_IDLE
- #define SC_HYPERVISOR_TIME_TO_APPLY
- #define SC_HYPERVISOR_ISPEED_W_SAMPLE
- #define SC_HYPERVISOR_ISPEED_CTX_SAMPLE
- #define SC_HYPERVISOR_NULL

Functions

- void * sc_hypervisor_init (struct sc_hypervisor_policy *policy)
- void sc_hypervisor_shutdown (void)
- void sc_hypervisor_register_ctx (unsigned sched_ctx, double total_flops)
- void sc_hypervisor_unregister_ctx (unsigned sched_ctx)
- void sc_hypervisor_stop_resize (unsigned sched_ctx)
- void sc_hypervisor_start_resize (unsigned sched_ctx)
- const char * sc_hypervisor_get_policy ()
- void sc_hypervisor_add_workers_to_sched_ctx (int *workers_to_add, unsigned nworkers_to_add, unsigned sched_ctx)
- void sc_hypervisor_remove_workers_from_sched_ctx (int *workers_to_remove, unsigned nworkers_to_remove, unsigned sched_ctx, unsigned now)
- void sc_hypervisor_move_workers (unsigned sender_sched_ctx, unsigned receiver_sched_ctx, int *workers_to_move, unsigned nworkers_to_move, unsigned now)
- void sc_hypervisor_size_ctxs (unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void sc_hypervisor_set_type_of_task (struct starpu_codelet *cl, unsigned sched_ctx, uint32_t footprint, size_t data_size)
- void sc_hypervisor_update_diff_total_flops (unsigned sched_ctx, double diff_total_flops)
- void sc_hypervisor_update_diff_elapsed_flops (unsigned sched_ctx, double diff_task_flops)
- void sc_hypervisor_ctl (unsigned sched_ctx,...)

17.32.1 Detailed Description

17.32.2 Macro Definition Documentation

17.32.2.1 #define SC_HYPERVISOR_MAX_IDLE

This macro is used when calling sc_hypervisor_ctl() and must be followed by 3 arguments: an array of int for the workerids to apply the condition, an int to indicate the size of the array, and a double value indicating the maximum idle time allowed for a worker before the resizing process should be triggered

17.32.2.2 #define SC_HYPERVISOR_PRIORITY

This macro is used when calling sc_hypervisor_ctl() and must be followed by 3 arguments: an array of int for the workerids to apply the condition, an int to indicate the size of the array, and an int value indicating the priority of the workers previously mentioned. The workers with the smallest priority are moved the first.

17.32.2.3 #define SC_HYPERVISOR_MIN_WORKERS

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument(int) indicating the minimum number of workers a context should have, underneath this limit the context cannot execute.

17.32.2.4 #define SC_HYPERVISOR_MAX_WORKERS

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument(int) indicating the maximum number of workers a context should have, above this limit the context would not be able to scale

17.32.2.5 #define SC_HYPERVISOR_GRANULARITY

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument(int) indicating the granularity of the resizing process (the number of workers should be moved from the context once it is resized) This parameter is ignore for the Gflops rate based strategy (see Resizing Strategies), the number of workers that have to be moved is calculated by the strategy.

17.32.2.6 #define SC_HYPERVISOR_FIXED_WORKERS

This macro is used when calling sc_hypervisor_ctl() and must be followed by 2 arguments: an array of int for the workerids to apply the condition and an int to indicate the size of the array. These workers are not allowed to be moved from the context.

17.32.2.7 #define SC_HYPERVISOR_MIN_TASKS

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument (int) that indicated the minimum number of tasks that have to be executed before the context could be resized. This parameter is ignored for the Application Driven strategy (see Resizing Strategies) where the user indicates exactly when the resize should be done.

17.32.2.8 #define SC_HYPERVISOR_NEW_WORKERS_MAX_IDLE

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument, a double value indicating the maximum idle time allowed for workers that have just been moved from other contexts in the current context.

17.32.2.9 #define SC_HYPERVISOR_TIME_TO_APPLY

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument (int) indicating the tag an executed task should have such that this configuration should be taken into account.

17.32.2.10 #define SC_HYPERVISOR_ISPEED_W_SAMPLE

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument, a double, that indicates the number of flops needed to be executed before computing the speed of a worker

17.32.2.11 #define SC_HYPERVISOR_ISPEED_CTX_SAMPLE

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 argument, a double, that indicates the number of flops needed to be executed before computing the speed of a context

17.32.2.12 #define SC_HYPERVISOR_NULL

This macro is used when calling sc_hypervisor_ctl() and must be followed by 1 arguments

17.32.3 Function Documentation

17.32.3.1 void * sc_hypervisor_init (struct sc_hypervisor_policy * *policy*)

There is a single hypervisor that is in charge of resizing contexts and the resizing strategy is chosen at the initialization of the hypervisor. A single resize can be done at a time.

The Scheduling Context Hypervisor Plugin provides a series of performance counters to StarPU. By incrementing them, StarPU can help the hypervisor in the resizing decision making process.

This function initializes the hypervisor to use the strategy provided as parameter and creates the performance counters (see starpu_sched_ctx_performance_counters). These performance counters represent actually some callbacks that will be used by the contexts to notify the information needed by the hypervisor.

Note: The Hypervisor is actually a worker that takes this role once certain conditions trigger the resizing process (there is no additional thread assigned to the hypervisor).

17.32.3.2 void sc_hypervisor_shutdown (void)

The hypervisor and all information concerning it is cleaned. There is no synchronization between this function and starpu_shutdown(). Thus, this should be called after starpu_shutdown(), because the performance counters will still need allocated callback functions.

17.32.3.3 void sc_hypervisor_register_ctx (unsigned *sched_ctx,* double *total_flops*)

Scheduling Contexts that have to be resized by the hypervisor must be first registered to the hypervisor. This function registers the context to the hypervisor, and indicate the number of flops the context will execute (used for Gflops rate based strategy or any other custom strategy needing it, for the others we can pass 0.0)

17.32.3.4 void sc_hypervisor_unregister_ctx (unsigned *sched_ctx*)

Whenever we want to exclude contexts from the resizing process we have to unregister them from the hypervisor.

17.32.3.5 void sc_hypervisor_stop_resize (unsigned *sched_ctx*)

The user can totally forbid the resizing of a certain context or can then change his mind and allow it (in this case the resizing is managed by the hypervisor, that can forbid it or allow it)

17.32.3.6 void sc_hypervisor_start_resize (unsigned *sched_ctx*)

Allow resizing of a context. The user can then provide information to the hypervisor concerning the conditions of resizing.

17.32.3.7 char * sc_hypervisor_get_policy ( )

Returns the name of the resizing policy the hypervisor uses

17.32.3.8 void sc_hypervisor_add_workers_to_sched_ctx (int * *workers_to_add,* unsigned *nworkers_to_add,* unsigned *sched_ctx*)

Ask the hypervisor to add workers to a sched_ctx

17.32.3.9 void sc_hypervisor_remove_workers_from_sched_ctx (int * *workers_to_remove,* unsigned *nworkers_to_remove,* unsigned *sched_ctx,* unsigned *now*)

Ask the hypervisor to remove workers from a sched_ctx

17.32.3.10 void sc_hypervisor_move_workers (unsigned *sender_sched_ctx,* unsigned *receiver_sched_ctx,* int * *workers_to_move,* unsigned *nworkers_to_move,* unsigned *now*)

Moves workers from one context to another

17.32.3.11 void sc_hypervisor_size_ctxs (unsigned * *sched_ctxs,* int *nsched_ctxs,* int * *workers,* int *nworkers*)

Ask the hypervisor to chose a distribution of workers in the required contexts

17.32.3.12 void sc_hypervisor_set_type_of_task (struct **starpu_codelet** * *cl,* unsigned *sched_ctx,* uint32_t *footprint,* size_t *data_size*)

Indicate the types of tasks a context will execute in order to better decide the sizing of ctxs

17.32.3.13 void sc_hypervisor_update_diff_total_flops (unsigned *sched_ctx,* double *diff_total_flops*)

Change dynamically the total number of flops of a context, move the deadline of the finishing time of the context

17.32.3.14 void sc_hypervisor_update_diff_elapsed_flops (unsigned *sched_ctx,* double *diff_task_flops*)

Change dynamically the number of the elapsed flops in a context, modify the past in order to better compute the speed

17.32.3.15 void sc_hypervisor_ctl (unsigned *sched_ctx,* ...)

Inputs conditions to the context sched_ctx with the following arguments. The argument list must be zero-terminated.

17.33 Scheduling Context Hypervisor - Building a new resizing policy

Data Structures

- struct sc_hypervisor_policy
- struct sc_hypervisor_policy_config
- struct sc_hypervisor_wrapper
- struct sc_hypervisor_resize_ack
- struct sc_hypervisor_policy_task_pool

Functions

- void sc_hypervisor_post_resize_request (unsigned sched_ctx, int task_tag)
- unsigned sc_hypervisor_get_size_req (unsigned **sched_ctxs, int *nsched_ctxs, int **workers, int *nworkers)
- void sc_hypervisor_save_size_req (unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void sc_hypervisor_free_size_req (void)
- unsigned sc_hypervisor_can_resize (unsigned sched_ctx)
- struct sc_hypervisor_policy_config * sc_hypervisor_get_config (unsigned sched_ctx)
- void sc_hypervisor_set_config (unsigned sched_ctx, void *config)
- unsigned * sc_hypervisor_get_sched_ctxs ()
- int sc_hypervisor_get_nsched_ctxs ()
- struct sc_hypervisor_wrapper * sc_hypervisor_get_wrapper (unsigned sched_ctx)
- double sc_hypervisor_get_elapsed_flops_per_sched_ctx (struct sc_hypervisor_wrapper *sc_w)

17.33.1 Detailed Description

17.33.2 Data Structure Documentation

17.33.2.1 struct sc_hypervisor_policy

This structure contains all the methods that implement a hypervisor resizing policy.

Data Fields

- const char * name
- unsigned custom
- void(* size_ctxs)(unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void(* resize_ctxs)(unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void(* handle_idle_cycle)(unsigned sched_ctx, int worker)
- void(* handle_pushed_task)(unsigned sched_ctx, int worker)
- void(* handle_poped_task)(unsigned sched_ctx, int worker, struct starpu_task *task, uint32_t footprint)
- void(* handle_idle_end)(unsigned sched_ctx, int worker)
- void(* handle_post_exec_hook)(unsigned sched_ctx, int task_tag)
- void(* handle_submitted_job)(struct starpu_codelet *cl, unsigned sched_ctx, uint32_t footprint, size_t data_size)
- void(* end_ctx)(unsigned sched_ctx)

17.33.2.1.1 Field Documentation

17.33.2.1.1.1 sc_hypervisor_policy::name

Indicates the name of the policy, if there is not a custom policy, the policy corresponding to this name will be used by the hypervisor

17.33.2.1.1.2 sc_hypervisor_policy::custom

Indicates whether the policy is custom or not

17.33.2.1.1.3 sc_hypervisor_policy::size_ctxs

Distribute workers to contexts even at the beginning of the program

17.33.2.1.1.4 sc_hypervisor_policy::resize_ctxs

Require explicit resizing

17.33.2.1.1.5 sc_hypervisor_policy::handle_idle_cycle

It is called whenever the indicated worker executes another idle cycle in sched_ctx

17.33.2.1.1.6 sc_hypervisor_policy::handle_pushed_task

It is called whenever a task is pushed on the worker's queue corresponding to the context sched_ctx

17.33.2.1.1.7 sc_hypervisor_policy::handle_poped_task

It is called whenever a task is poped from the worker's queue corresponding to the context sched_ctx

The hypervisor takes a decision when another task was poped from this worker in this ctx

17.33.2.1.1.8 sc_hypervisor_policy::handle_idle_end

It is called whenever a task is executed on the indicated worker and context after a long period of idle time

17.33.2.1.1.9 sc_hypervisor_policy::handle_post_exec_hook

It is called whenever a tag task has just been executed. The table of resize requests is provided as well as the tag

17.33.2.1.1.10 sc_hypervisor_policy::handle_submitted_job

The hypervisor takes a decision when a job was submitted in this ctx

17.33.2.1.1.11 sc_hypervisor_policy::end_ctx

The hypervisor takes a decision when a certain ctx was deleted

17.33.2.2 struct sc_hypervisor_policy_config

This structure contains all configuration information of a context. It contains configuration information for each context, which can be used to construct new resize strategies.

Data Fields

int	min_nworkers	Indicates the minimum number of workers needed by the context
int	max_nworkers	Indicates the maximum number of workers needed by the context
int	granularity	Indicates the workers granularity of the context
int	priority[STARPU_NMAXWORKERS]	Indicates the priority of each worker in the context
double	max_idle[STARPU_NMAXWORKERS]	Indicates the maximum idle time accepted before a resize is triggered
double	min_working[STARPU_NMAXWORKERS]	Indicates that underneath this limit the priority of the worker is reduced
int	fixed_workers[STARPU_NMAXWORKERS]	Indicates which workers can be moved and which ones are fixed

double	new_workers_↩ max_idle	Indicates the maximum idle time accepted before a resize is triggered for the workers that just arrived in the new context
double	ispeed_w_↩ sample[STAR↩ PU_NMAXWO↩ RKERS]	Indicates the sample used to compute the instant speed per worker
double	ispeed_ctx_↩ sample	Indicates the sample used to compute the instant speed per ctxs
double	time_sample	todo

17.33.2.3 struct sc_hypervisor_wrapper

This structure is a wrapper of the contexts available in StarPU and contains all information about a context obtained by incrementing the performance counters.

Data Fields

unsigned	sched_ctx	The context wrapped
struct sc_hypervisor↩ _policy_config *	config	The corresponding resize configuration
double	start_time_w[S↩ TARPU_NMA↩ XWORKERS]	
double	current_idle_↩ time[STARPU↩ _NMAXWORK↩ ERS]	The idle time counter of each worker of the context
double	idle_time[STA↩ RPU_NMAXW↩ ORKERS]	The time the workers were idle from the last resize
double	idle_start_↩ time[STARPU↩ _NMAXWORK↩ ERS]	The moment when the workers started being idle
double	exec_time[STA↩ RPU_NMAXW↩ ORKERS]	
double	exec_start_↩ time[STARPU↩ _NMAXWORK↩ ERS]	
int	worker_to_be_↩ removed[STA↩ RPU_NMAXW↩ ORKERS]	The list of workers that will leave this contexts (lazy resizing process)
int	pushed_↩ tasks[STARPU↩ _NMAXWORK↩ ERS]	The number of pushed tasks of each worker of the context

int	poped_tasks[STARPU_NMAXWORKERS]	The number of poped tasks of each worker of the context
double	total_flops	The total number of flops to execute by the context
double	total_elapsed_flops[STARPU_NMAXWORKERS]	The number of flops executed by each workers of the context
double	elapsed_flops[STARPU_NMAXWORKERS]	The number of flops executed by each worker of the context from last resize
size_t	elapsed_data[STARPU_NMAXWORKERS]	The quantity of data (in bytes) used to execute tasks on each worker in this ctx
int	elapsed_tasks[STARPU_NMAXWORKERS]	The nr of tasks executed on each worker in this ctx
double	ref_speed[2]	The average speed of the workers (type of workers) when they belonged to this context 0 - cuda 1 - cpu
double	submitted_flops	The number of flops submitted to this ctx
double	remaining_flops	The number of flops that still have to be executed by the workers in the context
double	start_time	The time when he started executed
double	real_start_time	The first time a task was pushed to this context
double	hyp_react_start_time	
struct sc_hypervisor_resize_ack	resize_ack	The structure confirming the last resize finished and a new one can be done
starpu_pthread_mutex_t	mutex	The mutex needed to synchronize the acknowledgment of the workers into the receiver context
unsigned	total_flops_available	A boolean indicating if the hypervisor can use the flops corresponding to the entire execution of the context
unsigned	to_be_sized	
unsigned	compute_idle[STARPU_NMAXWORKERS]	

17.33.2.4 struct sc_hypervisor_resize_ack

This structures checks if the workers moved to another context are actually taken into account in that context.

Data Fields

int	receiver_sched_ctx	The context receiving the new workers
int *	moved_workers	The workers moved to the receiver context
int	nmoved_workers	The number of workers moved

int *	acked_workers	If the value corresponding to a worker is 1, this one is taken into account in the new context if 0 not yet

17.33.2.5 struct sc_hypervisor_policy_task_pool

task wrapper linked list

Data Fields

struct starpu_codelet *	cl	Which codelet has been executed
uint32_t	footprint	Task footprint key
unsigned	sched_ctx_id	Context the task belongs to
unsigned long	n	Number of tasks of this kind
size_t	data_size	The quantity of data(in bytes) needed by the task to execute
struct sc_hypervisor_policy_task_pool *	next	Other task kinds

17.33.3 Function Documentation

17.33.3.1 void sc_hypervisor_post_resize_request (unsigned *sched_ctx*, int *task_tag*)

Requires resizing the context `sched_ctx` whenever a task tagged with the id `task_tag` finished executing

17.33.3.2 unsigned sc_hypervisor_get_size_req (unsigned ** *sched_ctxs*, int * *nsched_ctxs*, int ** *workers*, int * *nworkers*)

Check if there are pending demands of resizing

17.33.3.3 void sc_hypervisor_save_size_req (unsigned * *sched_ctxs*, int *nsched_ctxs*, int * *workers*, int *nworkers*)

Save a demand of resizing

17.33.3.4 void sc_hypervisor_free_size_req (void)

Clear the list of pending demands of resizing

17.33.3.5 unsigned sc_hypervisor_can_resize (unsigned *sched_ctx*)

Check out if a context can be resized

17.33.3.6 struct sc_hypervisor_policy_config * sc_hypervisor_get_config (unsigned *sched_ctx*)

Returns the configuration structure of a context

17.33.3.7 void sc_hypervisor_set_config (unsigned *sched_ctx*, void * *config*)

Set a certain configuration to a contexts

17.33.3.8 unsigned * sc_hypervisor_get_sched_ctxs ( )

Gets the contexts managed by the hypervisor

17.33.3.9 int sc_hypervisor_get_nsched_ctxs ( )

Gets the number of contexts managed by the hypervisor

17.33.3.10 struct sc_hypervisor_wrapper * sc_hypervisor_get_wrapper (unsigned *sched_ctx*)

Returns the wrapper corresponding the context `sched_ctx`

17.33.3.11 double sc_hypervisor_get_elapsed_flops_per_sched_ctx (struct sc_hypervisor_wrapper * *sc_w*)

Returns the flops of a context elapsed from the last resize

Chapter 18

File Index

18.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 19

File Documentation

19.1 starpu.h File Reference

```
#include <stdlib.h>
#include <stdint.h>
#include <starpu_config.h>
#include <starpu_opencl.h>
#include <starpu_thread.h>
#include <starpu_thread_util.h>
#include <starpu_util.h>
#include <starpu_data.h>
#include <starpu_data_interfaces.h>
#include <starpu_data_filters.h>
#include <starpu_stdlib.h>
#include <starpu_perfmodel.h>
#include <starpu_worker.h>
#include <starpu_task.h>
#include <starpu_task_list.h>
#include <starpu_task_util.h>
#include <starpu_sched_ctx.h>
#include <starpu_expert.h>
#include <starpu_rand.h>
#include <starpu_cuda.h>
#include <starpu_cublas.h>
#include <starpu_bound.h>
#include <starpu_hash.h>
#include <starpu_profiling.h>
#include <starpu_top.h>
#include <starpu_fxt.h>
#include <starpu_driver.h>
#include "starpu_deprecated_api.h"
```

Data Structures

- struct starpu_conf

Macros

- #define **main**

Functions

- int starpu_conf_init (struct starpu_conf *conf)
- int starpu_init (struct starpu_conf *conf) STARPU_WARN_UNUSED_RESULT
- void starpu_pause ()
- void starpu_resume ()
- void starpu_shutdown (void)
- void starpu_topology_print (FILE *f)
- int starpu_asynchronous_copy_disabled (void)
- int starpu_asynchronous_cuda_copy_disabled (void)
- int starpu_asynchronous_opencl_copy_disabled (void)
- void **starpu_display_stats** ()
- void starpu_get_version (int *major, int *minor, int *release)

19.2 starpu_bound.h File Reference

```
#include <stdio.h>
```

Functions

- void starpu_bound_start (int deps, int prio)
- void starpu_bound_stop (void)
- void starpu_bound_print_dot (FILE *output)
- void starpu_bound_compute (double *res, double *integer_res, int integer)
- void starpu_bound_print_lp (FILE *output)
- void starpu_bound_print_mps (FILE *output)
- void starpu_bound_print (FILE *output, int integer)

19.3 starpu_config.h File Reference

```
#include <sys/types.h>
```

Macros

- #define STARPU_MAJOR_VERSION
- #define STARPU_MINOR_VERSION
- #define STARPU_RELEASE_VERSION
- #define **STARPU_USE_CPU**
- #define STARPU_USE_CUDA
- #define STARPU_USE_OPENCL
- #define **STARPU_SIMGRID**
- #define **STARPU_HAVE_ICC**
- #define **STARPU_ATLAS**
- #define **STARPU_GOTO**
- #define **STARPU_MKL**
- #define **STARPU_SYSTEM_BLAS**
- #define **STARPU_BUILD_DIR**
- #define STARPU_OPENCL_DATADIR
- #define **STARPU_HAVE_MAGMA**

- #define **STARPU_OPENGL_RENDER**
- #define **STARPU_USE_GTK**
- #define **STARPU_HAVE_X11**
- #define **STARPU_HAVE_POSIX_MEMALIGN**
- #define **STARPU_HAVE_MEMALIGN**
- #define **STARPU_HAVE_MALLOC_H**
- #define **STARPU_HAVE_SYNC_BOOL_COMPARE_AND_SWAP**
- #define **STARPU_HAVE_SYNC_FETCH_AND_ADD**
- #define **STARPU_HAVE_SYNC_FETCH_AND_OR**
- #define **STARPU_HAVE_SYNC_LOCK_TEST_AND_SET**
- #define **STARPU_HAVE_SYNC_SYNCHRONIZE**
- #define **STARPU_MODEL_DEBUG**
- #define **STARPU_NO_ASSERT**
- #define **STARPU_HAVE_FFTW**
- #define **STARPU_HAVE_FFTWF**
- #define **STARPU_HAVE_FFTWL**
- #define **STARPU_HAVE_CURAND**
- #define **STARPU_MAXNODES**
- #define STARPU_NMAXBUFS
- #define **STARPU_MAXCPUS**
- #define STARPU_MAXCUDADEVS
- #define STARPU_MAXOPENCLDEVS
- #define STARPU_NMAXWORKERS
- #define **STARPU_NMAX_SCHED_CTXS**
- #define **STARPU_MAXIMPLEMENTATIONS**
- #define **STARPU_USE_SC_HYPERVISOR**
- #define **STARPU_HAVE_GLPK_H**
- #define **STARPU_HAVE_LIBNUMA**
- #define **STARPU_HAVE_WINDOWS**
- #define **STARPU_HAVE_UNSETENV**
- #define **__starpu_func__**
- #define **__starpu_inline**
- #define **STARPU_QUICK_CHECK**
- #define **STARPU_USE_DRAND48**
- #define **STARPU_USE_ERAND48_R**
- #define **STARPU_HAVE_NEARBYINTF**
- #define **STARPU_HAVE_RINTF**
- #define **STARPU_USE_TOP**
- #define **STARPU_HAVE_HWLOC**
- #define **STARPU_HAVE_PTHREAD_BARRIER**

Initialisation

- #define STARPU_USE_MPI

Typedefs

- typedef ssize_t **starpu_ssize_t**

19.4 starpu_cublas.h File Reference

Functions

- void starpu_cublas_init (void)
- void starpu_cublas_shutdown (void)

19.5 starpu_cuda.h File Reference

```
#include <starpu_config.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>
#include <cublas.h>
```

Macros

- #define STARPU_CUBLAS_REPORT_ERROR(status)
- #define STARPU_CUDA_REPORT_ERROR(status)

Functions

- void starpu_cublas_report_error (const char *func, const char *file, int line, cublasStatus status)
- void starpu_cuda_report_error (const char *func, const char *file, int line, cudaError_t status)
- cudaStream_t starpu_cuda_get_local_stream (void)
- const struct cudaDeviceProp * starpu_cuda_get_device_properties (unsigned workerid)
- int starpu_cuda_copy_async_sync (void *src_ptr, unsigned src_node, void *dst_ptr, unsigned dst_node, size_t ssize, cudaStream_t stream, enum cudaMemcpyKind kind)
- void starpu_cuda_set_device (unsigned devid)

19.6 starpu_data.h File Reference

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

Data Structures

- struct starpu_data_descr

Macros

- #define starpu_data_malloc_pinned_if_possible
- #define starpu_data_free_pinned_if_possible

Typedefs

- typedef struct _starpu_data_state * starpu_data_handle_t

Enumerations

- enum starpu_data_access_mode {
  STARPU_NONE, STARPU_R, STARPU_W, STARPU_RW,
  STARPU_SCRATCH, STARPU_REDUX }
- enum starpu_node_kind { STARPU_UNUSED, STARPU_CPU_RAM, STARPU_CUDA_RAM, STARPU_↩
  OPENCL_RAM }

Functions

- int **starpu_data_acquire_on_node** (starpu_data_handle_t handle, int node, enum starpu_data_access_↩ mode mode)
- int **starpu_data_acquire_on_node_cb** (starpu_data_handle_t handle, int node, enum starpu_data_↩ access_mode mode, void(∗callback)(void ∗), void ∗arg)
- void **starpu_data_release_on_node** (starpu_data_handle_t handle, int node)
- void starpu_data_display_memory_stats ()
- unsigned starpu_worker_get_memory_node (unsigned workerid)
- unsigned **starpu_memory_nodes_get_count** (void)
- enum starpu_node_kind starpu_node_get_kind (unsigned node)
- void starpu_data_set_sequential_consistency_flag (starpu_data_handle_t handle, unsigned flag)
- unsigned starpu_data_get_default_sequential_consistency_flag (void)
- void starpu_data_set_default_sequential_consistency_flag (unsigned flag)
- unsigned **starpu_data_test_if_allocated_on_node** (starpu_data_handle_t handle, unsigned memory_↩ node)

Basic Data Management API

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

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

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

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

- void starpu_data_unregister (starpu_data_handle_t handle)
- void starpu_data_unregister_no_coherency (starpu_data_handle_t handle)
- void starpu_data_unregister_submit (starpu_data_handle_t handle)
- void starpu_data_invalidate (starpu_data_handle_t handle)
- void starpu_data_invalidate_submit (starpu_data_handle_t handle)
- void starpu_data_advise_as_important (starpu_data_handle_t handle, unsigned is_important)
- int starpu_data_request_allocation (starpu_data_handle_t handle, unsigned node)
- int starpu_data_prefetch_on_node (starpu_data_handle_t handle, unsigned node, unsigned async)
- void starpu_data_set_wt_mask (starpu_data_handle_t handle, uint32_t wt_mask)
- void starpu_data_query_status (starpu_data_handle_t handle, int memory_node, int ∗is_allocated, int ∗is_valid, int ∗is_requested)
- void starpu_data_set_reduction_methods (starpu_data_handle_t handle, struct starpu_codelet ∗redux_cl, struct starpu_codelet ∗init_cl)

MPI Insert Task

- int starpu_data_set_rank (starpu_data_handle_t handle, int rank)
- int starpu_data_get_rank (starpu_data_handle_t handle)
- int starpu_data_set_tag (starpu_data_handle_t handle, int tag)
- int starpu_data_get_tag (starpu_data_handle_t handle)

Access registered data from the application

- #define STARPU_DATA_ACQUIRE_CB(handle, mode, code)
- int starpu_data_acquire (starpu_data_handle_t handle, enum starpu_data_access_mode mode)
- int starpu_data_acquire_cb (starpu_data_handle_t handle, enum starpu_data_access_mode mode, void(∗callback)(void ∗), void ∗arg)
- void starpu_data_release (starpu_data_handle_t handle)

19.7 starpu_data_filters.h File Reference

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

Data Structures

- struct starpu_data_filter

Functions

Basic API

- void starpu_data_partition (starpu_data_handle_t initial_handle, struct starpu_data_filter ∗f)
- void starpu_data_unpartition (starpu_data_handle_t root_data, unsigned gathering_node)
- int starpu_data_get_nb_children (starpu_data_handle_t handle)
- starpu_data_handle_t starpu_data_get_child (starpu_data_handle_t handle, unsigned i)
- starpu_data_handle_t starpu_data_get_sub_data (starpu_data_handle_t root_data, unsigned depth,...)
- starpu_data_handle_t starpu_data_vget_sub_data (starpu_data_handle_t root_data, unsigned depth, va_list pa)
- void starpu_data_map_filters (starpu_data_handle_t root_data, unsigned nfilters,...)
- void starpu_data_vmap_filters (starpu_data_handle_t root_data, unsigned nfilters, va_list pa)

Predefined BCSR Filter Functions

This section gives a partial list of the predefined partitioning functions for BCSR data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_bcsr_filter_canonical_block (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)
- void starpu_csr_filter_vertical_block (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)

Predefined Matrix Filter Functions

This section gives a partial list of the predefined partitioning functions for matrix data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_matrix_filter_block (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_block_shadow (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_vertical_block (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)
- void starpu_matrix_filter_vertical_block_shadow (void ∗father_interface, void ∗child_interface, struct starpu_data_filter ∗f, unsigned id, unsigned nparts)

Predefined Vector Filter Functions

This section gives a partial list of the predefined partitioning functions for vector data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`.

- void starpu_vector_filter_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_list (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_vector_filter_divide_in_2 (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

Predefined Block Filter Functions

This section gives a partial list of the predefined partitioning functions for block data. Examples on how to use them are shown in Partitioning Data. The complete list can be found in the file `starpu_data_filters.h`. A usage example is available in examples/filters/shadow3d.c

- void starpu_block_filter_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_vertical_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_vertical_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_depth_block (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)
- void starpu_block_filter_depth_block_shadow (void *father_interface, void *child_interface, struct starpu_data_filter *f, unsigned id, unsigned nparts)

19.8 starpu_data_interfaces.h File Reference

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

Data Structures

- struct starpu_data_copy_methods
- struct starpu_data_interface_ops
- struct starpu_matrix_interface
- struct starpu_coo_interface
- struct starpu_block_interface
- struct starpu_vector_interface
- struct starpu_variable_interface
- struct starpu_csr_interface
- struct starpu_bcsr_interface
- struct starpu_multiformat_data_interface_ops
- struct starpu_multiformat_interface

Macros

- #define STARPU_MULTIFORMAT_GET_CPU_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_CUDA_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_OPENCL_PTR(interface)
- #define STARPU_MULTIFORMAT_GET_NX(interface)

Accessing COO Data Interfaces

- #define STARPU_COO_GET_COLUMNS(interface)
- #define STARPU_COO_GET_COLUMNS_DEV_HANDLE(interface)
- #define STARPU_COO_GET_ROWS(interface)
- #define STARPU_COO_GET_ROWS_DEV_HANDLE(interface)
- #define STARPU_COO_GET_VALUES(interface)
- #define STARPU_COO_GET_VALUES_DEV_HANDLE(interface)
- #define STARPU_COO_GET_OFFSET
- #define STARPU_COO_GET_NX(interface)
- #define STARPU_COO_GET_NY(interface)
- #define STARPU_COO_GET_NVALUES(interface)
- #define STARPU_COO_GET_ELEMSIZE(interface)

Enumerations

- enum starpu_data_interface_id {
  STARPU_UNKNOWN_INTERFACE_ID, STARPU_MATRIX_INTERFACE_ID, STARPU_BLOCK_INTER↩
  FACE_ID, STARPU_VECTOR_INTERFACE_ID,
  STARPU_CSR_INTERFACE_ID, STARPU_BCSR_INTERFACE_ID, STARPU_VARIABLE_INTERFACE_↩
  ID, STARPU_VOID_INTERFACE_ID,
  STARPU_MULTIFORMAT_INTERFACE_ID, STARPU_COO_INTERFACE_ID, STARPU_MAX_INTERFA↩
  CE_ID }

Functions

- void starpu_multiformat_data_register (starpu_data_handle_t *handle, unsigned home_node, void *ptr, uint32_t nobjects, struct starpu_multiformat_data_interface_ops *format_ops)

Defining Interface

Applications can provide their own interface as shown in Defining A New Data Interface.

- int starpu_interface_copy (uintptr_t src, size_t src_offset, unsigned src_node, uintptr_t dst, size_t dst_↩
  offset, unsigned dst_node, size_t size, void *async_data)
- uintptr_t starpu_malloc_on_node (unsigned dst_node, size_t size)
- void starpu_free_on_node (unsigned dst_node, uintptr_t addr, size_t size)
- int starpu_data_interface_get_next_id (void)

Basic Data Management API

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

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

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

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

- void starpu_data_register (starpu_data_handle_t *handleptr, unsigned home_node, void *data_interface, struct starpu_data_interface_ops *ops)
- void starpu_data_ptr_register (starpu_data_handle_t handle, unsigned node)
- void starpu_data_register_same (starpu_data_handle_t *handledst, starpu_data_handle_t handlesrc)
- starpu_data_handle_t starpu_data_lookup (const void *ptr)

Accessing Data Interfaces

*Each data interface is provided with a set of field access functions. The ones using a void * parameter aimed to be used in codelet implementations (see for example the code in Vector Scaling Using StarPU's API).*

- void * starpu_data_handle_to_pointer (starpu_data_handle_t handle, unsigned node)
- void * starpu_data_get_local_ptr (starpu_data_handle_t handle)
- enum starpu_data_interface_id starpu_data_get_interface_id (starpu_data_handle_t handle)
- int starpu_data_pack (starpu_data_handle_t handle, void **ptr, starpu_ssize_t *count)
- int starpu_data_unpack (starpu_data_handle_t handle, void *ptr, size_t count)
- size_t starpu_data_get_size (starpu_data_handle_t handle)

Registering Data

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

- void * starpu_data_get_interface_on_node (starpu_data_handle_t handle, unsigned memory_node)
- void starpu_matrix_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t ld, uint32_t nx, uint32_t ny, size_t elemsize)
- void starpu_matrix_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev↩ _handle, size_t offset, uint32_t ld)
- void starpu_coo_data_register (starpu_data_handle_t *handleptr, unsigned home_node, uint32_t nx, uint32_t ny, uint32_t n_values, uint32_t *columns, uint32_t *rows, uintptr_t values, size_t elemsize)
- void starpu_block_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t ldy, uint32_t ldz, uint32_t nx, uint32_t ny, uint32_t nz, size_t elemsize)
- void starpu_block_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev↩ _handle, size_t offset, uint32_t ldy, uint32_t ldz)
- void starpu_vector_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, uint32_t nx, size_t elemsize)
- void starpu_vector_ptr_register (starpu_data_handle_t handle, unsigned node, uintptr_t ptr, uintptr_t dev↩ _handle, size_t offset)
- void starpu_variable_data_register (starpu_data_handle_t *handle, unsigned home_node, uintptr_t ptr, size_t size)
- void starpu_void_data_register (starpu_data_handle_t *handle)
- void starpu_csr_data_register (starpu_data_handle_t *handle, unsigned home_node, uint32_t nnz, uint32_t nrow, uintptr_t nzval, uint32_t *colind, uint32_t *rowptr, uint32_t firstentry, size_t elemsize)
- void starpu_bcsr_data_register (starpu_data_handle_t *handle, unsigned home_node, uint32_t nnz, uint32_t nrow, uintptr_t nzval, uint32_t *colind, uint32_t *rowptr, uint32_t firstentry, uint32_t r, uint32_t c, size_t elemsize)

Variables

- struct starpu_data_interface_ops **starpu_interface_matrix_ops**

Accessing Variable Data Interfaces

- #define STARPU_VARIABLE_GET_PTR(interface)
- #define STARPU_VARIABLE_GET_ELEMSIZE(interface)
- #define STARPU_VARIABLE_GET_DEV_HANDLE(interface)
- #define STARPU_VARIABLE_GET_OFFSET
- size_t starpu_variable_get_elemsize (starpu_data_handle_t handle)
- uintptr_t starpu_variable_get_local_ptr (starpu_data_handle_t handle)

Accessing Vector Data Interfaces

- #define STARPU_VECTOR_GET_PTR(interface)
- #define STARPU_VECTOR_GET_DEV_HANDLE(interface)
- #define STARPU_VECTOR_GET_OFFSET(interface)
- #define STARPU_VECTOR_GET_NX(interface)
- #define STARPU_VECTOR_GET_ELEMSIZE(interface)
- uint32_t starpu_vector_get_nx (starpu_data_handle_t handle)
- size_t starpu_vector_get_elemsize (starpu_data_handle_t handle)
- uintptr_t starpu_vector_get_local_ptr (starpu_data_handle_t handle)

Accessing Matrix Data Interfaces

- #define STARPU_MATRIX_GET_PTR(interface)
- #define STARPU_MATRIX_GET_DEV_HANDLE(interface)
- #define STARPU_MATRIX_GET_OFFSET(interface)
- #define STARPU_MATRIX_GET_NX(interface)
- #define STARPU_MATRIX_GET_NY(interface)
- #define STARPU_MATRIX_GET_LD(interface)
- #define STARPU_MATRIX_GET_ELEMSIZE(interface)
- uint32_t starpu_matrix_get_nx (starpu_data_handle_t handle)
- uint32_t starpu_matrix_get_ny (starpu_data_handle_t handle)
- uint32_t starpu_matrix_get_local_ld (starpu_data_handle_t handle)
- uintptr_t starpu_matrix_get_local_ptr (starpu_data_handle_t handle)
- size_t starpu_matrix_get_elemsize (starpu_data_handle_t handle)

Accessing Block Data Interfaces

- #define STARPU_BLOCK_GET_PTR(interface)
- #define STARPU_BLOCK_GET_DEV_HANDLE(interface)
- #define STARPU_BLOCK_GET_OFFSET(interface)
- #define STARPU_BLOCK_GET_NX(interface)
- #define STARPU_BLOCK_GET_NY(interface)
- #define STARPU_BLOCK_GET_NZ(interface)
- #define STARPU_BLOCK_GET_LDY(interface)
- #define STARPU_BLOCK_GET_LDZ(interface)
- #define STARPU_BLOCK_GET_ELEMSIZE(interface)
- uint32_t starpu_block_get_nx (starpu_data_handle_t handle)
- uint32_t starpu_block_get_ny (starpu_data_handle_t handle)
- uint32_t starpu_block_get_nz (starpu_data_handle_t handle)
- uint32_t starpu_block_get_local_ldy (starpu_data_handle_t handle)
- uint32_t starpu_block_get_local_ldz (starpu_data_handle_t handle)
- uintptr_t starpu_block_get_local_ptr (starpu_data_handle_t handle)
- size_t starpu_block_get_elemsize (starpu_data_handle_t handle)

Accessing BCSR Data Interfaces

- #define STARPU_BCSR_GET_NNZ(interface)
- #define STARPU_BCSR_GET_NZVAL(interface)
- #define STARPU_BCSR_GET_NZVAL_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_COLIND(interface)
- #define STARPU_BCSR_GET_COLIND_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_ROWPTR(interface)
- #define STARPU_BCSR_GET_ROWPTR_DEV_HANDLE(interface)
- #define STARPU_BCSR_GET_OFFSET
- uint32_t starpu_bcsr_get_nnz (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_nrow (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_firstentry (starpu_data_handle_t handle)
- uintptr_t starpu_bcsr_get_local_nzval (starpu_data_handle_t handle)
- uint32_t * starpu_bcsr_get_local_colind (starpu_data_handle_t handle)
- uint32_t * starpu_bcsr_get_local_rowptr (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_r (starpu_data_handle_t handle)
- uint32_t starpu_bcsr_get_c (starpu_data_handle_t handle)
- size_t starpu_bcsr_get_elemsize (starpu_data_handle_t handle)

Accessing CSR Data Interfaces

- #define STARPU_CSR_GET_NNZ(interface)
- #define STARPU_CSR_GET_NROW(interface)
- #define STARPU_CSR_GET_NZVAL(interface)
- #define STARPU_CSR_GET_NZVAL_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_COLIND(interface)
- #define STARPU_CSR_GET_COLIND_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_ROWPTR(interface)
- #define STARPU_CSR_GET_ROWPTR_DEV_HANDLE(interface)
- #define STARPU_CSR_GET_OFFSET
- #define STARPU_CSR_GET_FIRSTENTRY(interface)
- #define STARPU_CSR_GET_ELEMSIZE(interface)
- uint32_t starpu_csr_get_nnz (starpu_data_handle_t handle)
- uint32_t starpu_csr_get_nrow (starpu_data_handle_t handle)
- uint32_t starpu_csr_get_firstentry (starpu_data_handle_t handle)
- uintptr_t starpu_csr_get_local_nzval (starpu_data_handle_t handle)
- uint32_t * starpu_csr_get_local_colind (starpu_data_handle_t handle)
- uint32_t * starpu_csr_get_local_rowptr (starpu_data_handle_t handle)
- size_t starpu_csr_get_elemsize (starpu_data_handle_t handle)

19.9 starpu_deprecated_api.h File Reference

19.10 starpu_driver.h File Reference

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

Data Structures

- struct starpu_driver
- union starpu_driver.id

Functions

- int starpu_driver_run (struct starpu_driver *d)
- void starpu_drivers_request_termination (void)
- int starpu_driver_init (struct starpu_driver *d)
- int starpu_driver_run_once (struct starpu_driver *d)
- int starpu_driver_deinit (struct starpu_driver *d)

19.11 starpu_expert.h File Reference

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

Functions

- void starpu_wake_all_blocked_workers (void)
- int starpu_progression_hook_register (unsigned(*func)(void *arg), void *arg)
- void starpu_progression_hook_deregister (int hook_id)

19.12 starpu_fxt.h File Reference

```
#include <starpu_perfmodel.h>
```

Data Structures

- struct starpu_fxt_codelet_event
- struct starpu_fxt_options

Macros

- #define **STARPU_FXT_MAX_FILES**

Functions

- void starpu_fxt_options_init (struct starpu_fxt_options *options)
- void starpu_fxt_generate_trace (struct starpu_fxt_options *options)
- void starpu_fxt_start_profiling (void)
- void starpu_fxt_stop_profiling (void)

19.13 starpu_hash.h File Reference

```
#include <stdint.h>
#include <stddef.h>
```

Functions

Defining Interface

Applications can provide their own interface as shown in Defining A New Data Interface.

- uint32_t starpu_hash_crc32c_be_n (const void *input, size_t n, uint32_t inputcrc)
- uint32_t starpu_hash_crc32c_be (uint32_t input, uint32_t inputcrc)
- uint32_t starpu_hash_crc32c_string (const char *str, uint32_t inputcrc)

19.14 starpu_opencl.h File Reference

```
#include <starpu_config.h>
#include <CL/cl.h>
#include <assert.h>
```

Data Structures

- struct starpu_opencl_program

Functions

Writing OpenCL kernels

- void starpu_opencl_get_context (int devid, cl_context *context)
- void starpu_opencl_get_device (int devid, cl_device_id *device)
- void starpu_opencl_get_queue (int devid, cl_command_queue *queue)
- void starpu_opencl_get_current_context (cl_context *context)
- void starpu_opencl_get_current_queue (cl_command_queue *queue)
- int starpu_opencl_set_kernel_args (cl_int *err, cl_kernel *kernel,...)

Compiling OpenCL kernels

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

- void starpu_opencl_load_program_source (const char *source_file_name, char *located_file_name, char *located_dir_name, char *opencl_program_source)
- int starpu_opencl_compile_opencl_from_file (const char *source_file_name, const char *build_options)
- int starpu_opencl_compile_opencl_from_string (const char *opencl_program_source, const char *file_name, const char *build_options)
- int starpu_opencl_load_binary_opencl (const char *kernel_id, struct starpu_opencl_program *opencl_programs)
- int starpu_opencl_load_opencl_from_file (const char *source_file_name, struct starpu_opencl_program *opencl_programs, const char *build_options)
- int starpu_opencl_load_opencl_from_string (const char *opencl_program_source, struct starpu_opencl_program *opencl_programs, const char *build_options)
- int starpu_opencl_unload_opencl (struct starpu_opencl_program *opencl_programs)

Loading OpenCL kernels

- int starpu_opencl_load_kernel (cl_kernel *kernel, cl_command_queue *queue, struct starpu_opencl_program *opencl_programs, const char *kernel_name, int devid)
- int starpu_opencl_release_kernel (cl_kernel kernel)

OpenCL statistics

- int starpu_opencl_collect_stats (cl_event event)

OpenCL utilities

- #define STARPU_OPENCL_DISPLAY_ERROR(status)
- #define STARPU_OPENCL_REPORT_ERROR(status)
- #define STARPU_OPENCL_REPORT_ERROR_WITH_MSG(msg, status)
- const char * starpu_opencl_error_string (cl_int status)
- void starpu_opencl_display_error (const char *func, const char *file, int line, const char *msg, cl_int status)
- static __starpu_inline void starpu_opencl_report_error (const char *func, const char *file, int line, const char *msg, cl_int status)
- cl_int starpu_opencl_allocate_memory (cl_mem *addr, size_t size, cl_mem_flags flags)
- cl_int starpu_opencl_copy_ram_to_opencl (void *ptr, unsigned src_node, cl_mem buffer, unsigned dst_node, size_t size, size_t offset, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_opencl_to_ram (cl_mem buffer, unsigned src_node, void *ptr, unsigned dst_node, size_t size, size_t offset, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_opencl_to_opencl (cl_mem src, unsigned src_node, size_t src_offset, cl_mem dst, unsigned dst_node, size_t dst_offset, size_t size, cl_event *event, int *ret)
- cl_int starpu_opencl_copy_async_sync (uintptr_t src, size_t src_offset, unsigned src_node, uintptr_t dst, size_t dst_offset, unsigned dst_node, size_t size, cl_event *event)

19.15 starpu_perfmodel.h File Reference

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

Data Structures

- struct starpu_perfmodel_history_entry
- struct starpu_perfmodel_history_list
- struct starpu_perfmodel_regression_model
- struct starpu_perfmodel_per_arch
- struct starpu_perfmodel

Macros

- #define **STARPU_NARCH_VARIATIONS**
- #define **starpu_per_arch_perfmodel**

Enumerations

- enum starpu_perfmodel_archtype { STARPU_CPU_DEFAULT, STARPU_CUDA_DEFAULT, STARPU_O↩ PENCL_DEFAULT }
- enum starpu_perfmodel_type {
  STARPU_PER_ARCH, STARPU_COMMON, STARPU_HISTORY_BASED, STARPU_REGRESSION_B↩ ASED,
  STARPU_NL_REGRESSION_BASED }

Functions

- enum starpu_perfmodel_archtype starpu_worker_get_perf_archtype (int workerid)
- int starpu_perfmodel_load_symbol (const char *symbol, struct starpu_perfmodel *model)
- int starpu_perfmodel_unload_model (struct starpu_perfmodel *model)
- void starpu_perfmodel_debugfilepath (struct starpu_perfmodel *model, enum starpu_perfmodel_archtype arch, char *path, size_t maxlen, unsigned nimpl)
- void starpu_perfmodel_get_arch_name (enum starpu_perfmodel_archtype arch, char *archname, size_t maxlen, unsigned nimpl)
- double **starpu_permodel_history_based_expected_perf** (struct starpu_perfmodel *model, enum starpu_perfmodel_archtype arch, uint32_t footprint)
- int starpu_perfmodel_list (FILE *output)
- void starpu_perfmodel_print (struct starpu_perfmodel *model, enum starpu_perfmodel_archtype arch, unsigned nimpl, char *parameter, uint32_t *footprint, FILE *output)
- int starpu_perfmodel_print_all (struct starpu_perfmodel *model, char *arch, char *parameter, uint32_t *footprint, FILE *output)
- void starpu_perfmodel_directory (FILE *output)
- void starpu_perfmodel_update_history (struct starpu_perfmodel *model, struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned cpuid, unsigned nimpl, double measured)
- void starpu_bus_print_bandwidth (FILE *f)
- void starpu_bus_print_affinity (FILE *f)
- double starpu_transfer_bandwidth (unsigned src_node, unsigned dst_node)
- double starpu_transfer_latency (unsigned src_node, unsigned dst_node)
- double starpu_transfer_predict (unsigned src_node, unsigned dst_node, size_t size)

19.16 starpu_profiling.h File Reference

```
#include <starpu.h>
#include <errno.h>
#include <time.h>
#include <starpu_util.h>
```

Data Structures

- struct starpu_profiling_task_info
- struct starpu_profiling_worker_info
- struct starpu_profiling_bus_info

Macros

- #define STARPU_PROFILING_DISABLE
- #define STARPU_PROFILING_ENABLE
- #define **starpu_timespec_cmp**(a, b, CMP)

Functions

- void starpu_profiling_init ()
- void starpu_profiling_set_id (int new_id)
- int starpu_profiling_status_set (int status)
- int starpu_profiling_status_get (void)
- int starpu_profiling_worker_get_info (int workerid, struct starpu_profiling_worker_info *worker_info)
- int starpu_bus_get_count (void)

- int starpu_bus_get_id (int src, int dst)
- int starpu_bus_get_src (int busid)
- int starpu_bus_get_dst (int busid)
- int starpu_bus_get_profiling_info (int busid, struct starpu_profiling_bus_info *bus_info)
- static __starpu_inline void **starpu_timespec_clear** (struct timespec *tsp)
- static __starpu_inline void **starpu_timespec_add** (struct timespec *a, struct timespec *b, struct timespec *result)
- static __starpu_inline void **starpu_timespec_accumulate** (struct timespec *result, struct timespec *a)
- static __starpu_inline void **starpu_timespec_sub** (const struct timespec *a, const struct timespec *b, struct timespec *result)
- double starpu_timing_timespec_delay_us (struct timespec *start, struct timespec *end)
- double starpu_timing_timespec_to_us (struct timespec *ts)
- void starpu_profiling_bus_helper_display_summary (void)
- void starpu_profiling_worker_helper_display_summary (void)

19.17 starpu_rand.h File Reference

```
#include <stdlib.h>
#include <starpu_config.h>
```

Macros

- #define **starpu_srand48**(seed)
- #define **starpu_drand48**()
- #define **starpu_erand48**(xsubi)
- #define **starpu_srand48_r**(seed, buffer)
- #define **starpu_erand48_r**(xsubi, buffer, result)

Typedefs

- typedef int **starpu_drand48_data**

19.18 starpu_sched_ctx.h File Reference

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

Functions

- unsigned **starpu_sched_ctx_contains_type_of_worker** (enum starpu_worker_archtype arch, unsigned sched_ctx_id)
- int starpu_sched_get_min_priority (void)
- int starpu_sched_get_max_priority (void)
- int starpu_sched_set_min_priority (int min_prio)
- int starpu_sched_set_max_priority (int max_prio)
- int **starpu_sched_ctx_min_priority_is_set** (unsigned sched_ctx_id)
- int **starpu_sched_ctx_max_priority_is_set** (unsigned sched_ctx_id)
- int **starpu_sched_ctx_get_nready_tasks** (unsigned sched_ctx_id)
- double **starpu_sched_ctx_get_nready_flops** (unsigned sched_ctx_id)

Scheduling Context Worker Collection

- struct starpu_worker_collection * starpu_sched_ctx_create_worker_collection (unsigned sched_ctx_id, enum starpu_worker_collection_type type)
- void starpu_sched_ctx_delete_worker_collection (unsigned sched_ctx_id)
- struct starpu_worker_collection * starpu_sched_ctx_get_worker_collection (unsigned sched_ctx_id)

Scheduling Context Link with Hypervisor

- void starpu_sched_ctx_set_policy_data (unsigned sched_ctx_id, void *policy_data)
- void * starpu_sched_ctx_get_policy_data (unsigned sched_ctx_id)
- void starpu_sched_ctx_call_pushed_task_cb (int workerid, unsigned sched_ctx_id)

Scheduling Contexts Basic API

- #define STARPU_SCHED_CTX_POLICY_NAME
- #define STARPU_SCHED_CTX_POLICY_STRUCT
- #define STARPU_SCHED_CTX_POLICY_MIN_PRIO
- #define STARPU_SCHED_CTX_POLICY_MAX_PRIO
- unsigned starpu_sched_ctx_create (int *workerids_ctx, int nworkers_ctx, const char *sched_ctx_name,...)
- unsigned starpu_sched_ctx_create_inside_interval (const char *policy_name, const char *sched_name, int min_ncpus, int max_ncpus, int min_ngpus, int max_ngpus, unsigned allow_overlap)
- void starpu_sched_ctx_register_close_callback (unsigned sched_ctx_id, void(*close_callback)(unsigned sched_ctx_id, void *args), void *args)
- void starpu_sched_ctx_add_workers (int *workerids_ctx, int nworkers_ctx, unsigned sched_ctx_id)
- void starpu_sched_ctx_remove_workers (int *workerids_ctx, int nworkers_ctx, unsigned sched_ctx_id)
- void starpu_sched_ctx_delete (unsigned sched_ctx_id)
- void starpu_sched_ctx_set_inheritor (unsigned sched_ctx_id, unsigned inheritor)
- void starpu_sched_ctx_set_context (unsigned *sched_ctx_id)
- unsigned starpu_sched_ctx_get_context (void)
- void starpu_sched_ctx_stop_task_submission (void)
- void starpu_sched_ctx_finished_submit (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_get_workers_list (unsigned sched_ctx_id, int **workerids)
- unsigned starpu_sched_ctx_get_nworkers (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_get_nshared_workers (unsigned sched_ctx_id, unsigned sched_ctx_id2)
- unsigned starpu_sched_ctx_contains_worker (int workerid, unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_worker_get_id (unsigned sched_ctx_id)
- unsigned starpu_sched_ctx_overlapping_ctxs_on_worker (int workerid)

Scheduling Context Priorities

- #define STARPU_MIN_PRIO
- #define STARPU_MAX_PRIO
- #define STARPU_DEFAULT_PRIO
- int starpu_sched_ctx_get_min_priority (unsigned sched_ctx_id)
- int starpu_sched_ctx_get_max_priority (unsigned sched_ctx_id)
- int starpu_sched_ctx_set_min_priority (unsigned sched_ctx_id, int min_prio)
- int starpu_sched_ctx_set_max_priority (unsigned sched_ctx_id, int max_prio)

19.19 starpu_sched_ctx_hypervisor.h File Reference

Data Structures

- struct starpu_sched_ctx_performance_counters

Functions

Scheduling Context Link with Hypervisor

- void starpu_sched_ctx_set_perf_counters (unsigned sched_ctx_id, void *perf_counters)
- void starpu_sched_ctx_notify_hypervisor_exists (void)
- unsigned starpu_sched_ctx_check_if_hypervisor_exists (void)

19.20 starpu_scheduler.h File Reference

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

Data Structures

- struct starpu_sched_policy

Functions

- struct starpu_sched_policy ** starpu_sched_get_predefined_policies ()
- void starpu_worker_get_sched_condition (int workerid, starpu_pthread_mutex_t **sched_mutex, starpu_pthread_cond_t **sched_cond)
- int **starpu_wakeup_worker** (int workerid, starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex)
- int starpu_worker_can_execute_task (unsigned workerid, struct starpu_task *task, unsigned nimpl)
- int starpu_push_local_task (int workerid, struct starpu_task *task, int back)
- int starpu_push_task_end (struct starpu_task *task)
- int starpu_combined_worker_assign_workerid (int nworkers, int workerid_array[])
- int starpu_combined_worker_get_description (int workerid, int *worker_size, int **combined_workerid)
- int starpu_combined_worker_can_execute_task (unsigned workerid, struct starpu_task *task, unsigned nimpl)
- int starpu_get_prefetch_flag (void)
- int starpu_prefetch_task_input_on_node (struct starpu_task *task, unsigned node)
- uint32_t starpu_task_footprint (struct starpu_perfmodel *model, struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_expected_length (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_worker_get_relative_speedup (enum starpu_perfmodel_archtype perf_archtype)
- double starpu_task_expected_data_transfer_time (unsigned memory_node, struct starpu_task *task)
- double starpu_data_expected_transfer_time (starpu_data_handle_t handle, unsigned memory_node, enum starpu_data_access_mode mode)
- double starpu_task_expected_power (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_expected_conversion_time (struct starpu_task *task, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_bundle_expected_length (starpu_task_bundle_t bundle, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- double starpu_task_bundle_expected_data_transfer_time (starpu_task_bundle_t bundle, unsigned memory_node)
- double starpu_task_bundle_expected_power (starpu_task_bundle_t bundle, enum starpu_perfmodel_archtype arch, unsigned nimpl)
- void starpu_sched_ctx_worker_shares_tasks_lists (int workerid, int sched_ctx_id)

19.21 starpu_stdlib.h File Reference

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

Macros

- #define STARPU_MALLOC_PINNED
- #define STARPU_MALLOC_COUNT

Functions

- void starpu_malloc_set_align (size_t align)
- int starpu_malloc (void **A, size_t dim)
- int starpu_free (void *A)
- int starpu_malloc_flags (void **A, size_t dim, int flags)
- int starpu_free_flags (void *A, size_t dim, int flags)
- starpu_ssize_t starpu_memory_get_total (unsigned node)
- starpu_ssize_t starpu_memory_get_available (unsigned node)

19.22 starpu_task.h File Reference

```
#include <starpu.h>
#include <starpu_data.h>
#include <starpu_util.h>
#include <starpu_task_bundle.h>
#include <errno.h>
#include <cuda.h>
```

Data Structures

- struct starpu_codelet
- struct starpu_task

Macros

- #define STARPU_CPU
- #define STARPU_CUDA
- #define STARPU_OPENCL
- #define STARPU_TASK_INVALID
- #define STARPU_MULTIPLE_CPU_IMPLEMENTATIONS
- #define STARPU_MULTIPLE_CUDA_IMPLEMENTATIONS
- #define STARPU_MULTIPLE_OPENCL_IMPLEMENTATIONS
- #define STARPU_TASK_INITIALIZER
- #define STARPU_TASK_GET_HANDLE(task, i)
- #define STARPU_TASK_SET_HANDLE(task, handle, i)
- #define STARPU_CODELET_GET_MODE(codelet, i)
- #define STARPU_CODELET_SET_MODE(codelet, mode, i)
- #define **STARPU_CODELET_GET_NODE**(codelet, i)
- #define **STARPU_CODELET_SET_NODE**(codelet, __node, i)

Typedefs

- typedef uint64_t starpu_tag_t
- typedef void(∗ starpu_cpu_func_t)(void ∗∗, void ∗)
- typedef void(∗ starpu_cuda_func_t)(void ∗∗, void ∗)
- typedef void(∗ starpu_opencl_func_t)(void ∗∗, void ∗)

Enumerations

- enum starpu_codelet_type { STARPU_SEQ, STARPU_SPMD, STARPU_FORKJOIN }
- enum starpu_task_status {
  STARPU_TASK_INVALID, **STARPU_TASK_INVALID**, STARPU_TASK_BLOCKED, STARPU_TASK_R↩
  EADY,
  STARPU_TASK_RUNNING, STARPU_TASK_FINISHED, STARPU_TASK_BLOCKED_ON_TAG, STARP↩
  U_TASK_BLOCKED_ON_TASK,
  STARPU_TASK_BLOCKED_ON_DATA }

Functions

- void starpu_tag_declare_deps (starpu_tag_t id, unsigned ndeps,...)
- void starpu_tag_declare_deps_array (starpu_tag_t id, unsigned ndeps, starpu_tag_t ∗array)
- void starpu_task_declare_deps_array (struct starpu_task ∗task, unsigned ndeps, struct starpu_task ∗task↩
  _array[])
- int starpu_tag_wait (starpu_tag_t id)
- int starpu_tag_wait_array (unsigned ntags, starpu_tag_t ∗id)
- void starpu_tag_notify_from_apps (starpu_tag_t id)
- void starpu_tag_restart (starpu_tag_t id)
- void starpu_tag_remove (starpu_tag_t id)
- void starpu_task_init (struct starpu_task ∗task)
- void starpu_task_clean (struct starpu_task ∗task)
- struct starpu_task ∗ starpu_task_create (void)
- void starpu_task_destroy (struct starpu_task ∗task)
- int starpu_task_submit (struct starpu_task ∗task) STARPU_WARN_UNUSED_RESULT
- int starpu_task_submit_to_ctx (struct starpu_task ∗task, unsigned sched_ctx_id)
- int starpu_task_wait (struct starpu_task ∗task) STARPU_WARN_UNUSED_RESULT
- int starpu_task_wait_for_all (void)
- int starpu_task_wait_for_all_in_ctx (unsigned sched_ctx_id)
- int starpu_task_wait_for_no_ready (void)
- int starpu_task_nready (void)
- int starpu_task_nsubmitted (void)
- void starpu_codelet_init (struct starpu_codelet ∗cl)
- void starpu_codelet_display_stats (struct starpu_codelet ∗cl)
- struct starpu_task ∗ starpu_task_get_current (void)
- void starpu_parallel_task_barrier_init (struct starpu_task ∗task, int workerid)
- void starpu_parallel_task_barrier_init_n (struct starpu_task ∗task, int worker_size)
- struct starpu_task ∗ starpu_task_dup (struct starpu_task ∗task)
- void starpu_task_set_implementation (struct starpu_task ∗task, unsigned impl)
- unsigned starpu_task_get_implementation (struct starpu_task ∗task)

19.23 starpu_task_bundle.h File Reference

Typedefs

- typedef struct
  _starpu_task_bundle ∗ starpu_task_bundle_t

Functions

- void starpu_task_bundle_create (starpu_task_bundle_t *bundle)
- int starpu_task_bundle_insert (starpu_task_bundle_t bundle, struct starpu_task *task)
- int starpu_task_bundle_remove (starpu_task_bundle_t bundle, struct starpu_task *task)
- void starpu_task_bundle_close (starpu_task_bundle_t bundle)

19.24 starpu_task_list.h File Reference

```
#include <starpu_task.h>
#include <starpu_util.h>
```

Data Structures

- struct starpu_task_list

Functions

- static STARPU_INLINE void starpu_task_list_init (struct starpu_task_list *list)
- static STARPU_INLINE void starpu_task_list_push_front (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE void starpu_task_list_push_back (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE struct starpu_task * starpu_task_list_front (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_back (struct starpu_task_list *list)
- static STARPU_INLINE int starpu_task_list_empty (struct starpu_task_list *list)
- static STARPU_INLINE void starpu_task_list_erase (struct starpu_task_list *list, struct starpu_task *task)
- static STARPU_INLINE struct starpu_task * starpu_task_list_pop_front (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_pop_back (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * starpu_task_list_begin (struct starpu_task_list *list)
- static STARPU_INLINE struct starpu_task * **starpu_task_list_end** (struct starpu_task_list *list STARPU_ATTRIBUTE_UNUSED)
- static STARPU_INLINE struct starpu_task * starpu_task_list_next (struct starpu_task *task)

19.25 starpu_task_util.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <starpu.h>
```

Macros

- #define STARPU_VALUE
- #define STARPU_CALLBACK
- #define STARPU_CALLBACK_WITH_ARG
- #define STARPU_CALLBACK_ARG
- #define STARPU_PRIORITY
- #define STARPU_DATA_ARRAY
- #define STARPU_TAG
- #define **STARPU_HYPERVISOR_TAG**
- #define STARPU_FLOPS
- #define STARPU_SCHED_CTX
- #define **STARPU_PROLOGUE_CALLBACK**
- #define **STARPU_PROLOGUE_CALLBACK_ARG**
- #define STARPU_EXECUTE_ON_WORKER

MPI Insert Task

- #define STARPU_EXECUTE_ON_NODE
- #define STARPU_EXECUTE_ON_DATA

Functions

- void starpu_create_sync_task (starpu_tag_t sync_tag, unsigned ndeps, starpu_tag_t *deps, void(*callback)(void *), void *callback_arg)
- struct starpu_task * starpu_task_build (struct starpu_codelet *cl,...)
- int starpu_insert_task (struct starpu_codelet *cl,...)
- void starpu_codelet_unpack_args (void *cl_arg,...)
- void starpu_codelet_pack_args (void **arg_buffer, size_t *arg_buffer_size,...)

19.26 starpu_thread.h File Reference

```
#include <starpu_config.h>
#include <xbt/synchro_core.h>
#include <msg/msg.h>
```

Data Structures

- struct starpu_pthread_barrier_t

Macros

- #define STARPU_PTHREAD_MUTEX_INITIALIZER
- #define STARPU_PTHREAD_COND_INITIALIZER
- #define **STARPU_PTHREAD_BARRIER_SERIAL_THREAD**

Typedefs

- typedef int **starpu_pthread_t**
- typedef int **starpu_pthread_attr_t**
- typedef xbt_mutex_t **starpu_pthread_mutex_t**
- typedef int **starpu_pthread_mutexattr_t**
- typedef int **starpu_pthread_key_t**
- typedef xbt_cond_t **starpu_pthread_cond_t**
- typedef int **starpu_pthread_condattr_t**
- typedef xbt_mutex_t **starpu_pthread_rwlock_t**
- typedef int **starpu_pthread_rwlockattr_t**
- typedef int **starpu_pthread_barrierattr_t**

Functions

- int **starpu_pthread_create_on** (char *name, starpu_pthread_t *thread, const starpu_pthread_attr_t *attr, void *(*start_routine)(void *), void *arg, int where)
- int starpu_pthread_create (starpu_pthread_t *thread, const starpu_pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
- int starpu_pthread_join (starpu_pthread_t thread, void **retval)
- int starpu_pthread_attr_init (starpu_pthread_attr_t *attr)
- int starpu_pthread_attr_destroy (starpu_pthread_attr_t *attr)
- int starpu_pthread_attr_setdetachstate (starpu_pthread_attr_t *attr, int detachstate)
- int starpu_pthread_mutex_init (starpu_pthread_mutex_t *mutex, const starpu_pthread_mutexattr_t *mutexattr)
- int starpu_pthread_mutex_destroy (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_lock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_unlock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_mutex_trylock (starpu_pthread_mutex_t *mutex)
- int starpu_pthread_key_create (starpu_pthread_key_t *key, void(*destr_function)(void *))
- int starpu_pthread_key_delete (starpu_pthread_key_t key)
- int starpu_pthread_setspecific (starpu_pthread_key_t key, const void *pointer)
- void * starpu_pthread_getspecific (starpu_pthread_key_t key)
- int starpu_pthread_cond_init (starpu_pthread_cond_t *cond, starpu_pthread_condattr_t *cond_attr)
- int starpu_pthread_cond_signal (starpu_pthread_cond_t *cond)
- int starpu_pthread_cond_broadcast (starpu_pthread_cond_t *cond)
- int starpu_pthread_cond_wait (starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex)
- int starpu_pthread_cond_timedwait (starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex, const struct timespec *abstime)
- int starpu_pthread_cond_destroy (starpu_pthread_cond_t *cond)
- int starpu_pthread_rwlock_init (starpu_pthread_rwlock_t *rwlock, const starpu_pthread_rwlockattr_t *attr)
- int starpu_pthread_rwlock_destroy (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_rdlock (starpu_pthread_rwlock_t *rwlock)
- int **starpu_pthread_rwlock_tryrdlock** (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_wrlock (starpu_pthread_rwlock_t *rwlock)
- int **starpu_pthread_rwlock_trywrlock** (starpu_pthread_rwlock_t *rwlock)
- int starpu_pthread_rwlock_unlock (starpu_pthread_rwlock_t *rwlock)
- int **starpu_pthread_barrier_init** (starpu_pthread_barrier_t *barrier, const starpu_pthread_barrierattr_t *attr, unsigned count)
- int **starpu_pthread_barrier_destroy** (starpu_pthread_barrier_t *barrier)
- int **starpu_pthread_barrier_wait** (starpu_pthread_barrier_t *barrier)

19.26.1 Data Structure Documentation

19.26.1.1 struct starpu_pthread_barrier_t

Data Fields

starpu_↩ pthread_mutex↩ _t	mutex	
starpu_↩ pthread_cond_t	cond	
unsigned	count	
unsigned	done	

19.27 starpu_thread_util.h File Reference

```
#include <starpu_util.h>
#include <errno.h>
```

Macros

- #define STARPU_PTHREAD_CREATE_ON(name, thread, attr, routine, arg, where)
- #define STARPU_PTHREAD_CREATE(thread, attr, routine, arg)
- #define STARPU_PTHREAD_MUTEX_INIT(mutex, attr)
- #define STARPU_PTHREAD_MUTEX_DESTROY(mutex)
- #define STARPU_PTHREAD_MUTEX_LOCK(mutex)
- #define **STARPU_PTHREAD_MUTEX_TRYLOCK**(mutex)
- #define STARPU_PTHREAD_MUTEX_UNLOCK(mutex)
- #define STARPU_PTHREAD_KEY_CREATE(key, destr)
- #define STARPU_PTHREAD_KEY_DELETE(key)
- #define STARPU_PTHREAD_SETSPECIFIC(key, ptr)
- #define STARPU_PTHREAD_GETSPECIFIC(key)
- #define STARPU_PTHREAD_RWLOCK_INIT(rwlock, attr)
- #define STARPU_PTHREAD_RWLOCK_RDLOCK(rwlock)
- #define **STARPU_PTHREAD_RWLOCK_TRYRDLOCK**(rwlock)
- #define STARPU_PTHREAD_RWLOCK_WRLOCK(rwlock)
- #define **STARPU_PTHREAD_RWLOCK_TRYWRLOCK**(rwlock)
- #define STARPU_PTHREAD_RWLOCK_UNLOCK(rwlock)
- #define STARPU_PTHREAD_RWLOCK_DESTROY(rwlock)
- #define STARPU_PTHREAD_COND_INIT(cond, attr)
- #define STARPU_PTHREAD_COND_DESTROY(cond)
- #define STARPU_PTHREAD_COND_SIGNAL(cond)
- #define STARPU_PTHREAD_COND_BROADCAST(cond)
- #define STARPU_PTHREAD_COND_WAIT(cond, mutex)
- #define STARPU_PTHREAD_BARRIER_INIT(barrier, attr, count)
- #define STARPU_PTHREAD_BARRIER_DESTROY(barrier)
- #define STARPU_PTHREAD_BARRIER_WAIT(barrier)

Functions

- static STARPU_INLINE int **_starpu_pthread_mutex_trylock** (starpu_pthread_mutex_t *mutex, char *file, int line)
- static STARPU_INLINE int **_starpu_pthread_rwlock_tryrdlock** (starpu_pthread_rwlock_t *rwlock, char *file, int line)
- static STARPU_INLINE int **_starpu_pthread_rwlock_trywrlock** (starpu_pthread_rwlock_t *rwlock, char *file, int line)

19.28 starpu_top.h File Reference

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

Data Structures

- struct starpu_top_data
- struct starpu_top_param

Enumerations

- enum starpu_top_data_type { STARPU_TOP_DATA_BOOLEAN, STARPU_TOP_DATA_INTEGER, STARPU_TOP_DATA_FLOAT }
- enum starpu_top_param_type { STARPU_TOP_PARAM_BOOLEAN, STARPU_TOP_PARAM_INTEGER, STARPU_TOP_PARAM_FLOAT, STARPU_TOP_PARAM_ENUM }
- enum starpu_top_message_type { TOP_TYPE_GO, TOP_TYPE_SET, TOP_TYPE_CONTINUE, TOP_TYPE_ENABLE, TOP_TYPE_DISABLE, TOP_TYPE_DEBUG, TOP_TYPE_UNKNOW }

Functions

Functions to call before the initialisation

- struct starpu_top_data * starpu_top_add_data_boolean (const char *data_name, int active)
- struct starpu_top_data * starpu_top_add_data_integer (const char *data_name, int minimum_value, int maximum_value, int active)
- struct starpu_top_data * starpu_top_add_data_float (const char *data_name, double minimum_value, double maximum_value, int active)
- struct starpu_top_param * starpu_top_register_parameter_boolean (const char *param_name, int *parameter_field, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_integer (const char *param_name, int *parameter_field, int minimum_value, int maximum_value, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_float (const char *param_name, double *parameter_field, double minimum_value, double maximum_value, void(*callback)(struct starpu_top_param *))
- struct starpu_top_param * starpu_top_register_parameter_enum (const char *param_name, int *parameter_field, char **values, int nb_values, void(*callback)(struct starpu_top_param *))

Initialisation

- void starpu_top_init_and_wait (const char *server_name)

To call after initialisation

- void starpu_top_update_parameter (const struct starpu_top_param *param)
- void starpu_top_update_data_boolean (const struct starpu_top_data *data, int value)
- void starpu_top_update_data_integer (const struct starpu_top_data *data, int value)
- void starpu_top_update_data_float (const struct starpu_top_data *data, double value)
- void starpu_top_task_prevision (struct starpu_task *task, int devid, unsigned long long start, unsigned long long end)
- void starpu_top_debug_log (const char *message)
- void starpu_top_debug_lock (const char *message)

19.29 starpu_util.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <starpu_config.h>
#include <starpu_task.h>
#include <sys/time.h>
```

Macros

- #define STARPU_GNUC_PREREQ(maj, min)
- #define STARPU_UNLIKELY(expr)
- #define STARPU_LIKELY(expr)
- #define STARPU_ATTRIBUTE_UNUSED
- #define STARPU_ATTRIBUTE_INTERNAL
- #define STARPU_ATTRIBUTE_MALLOC
- #define STARPU_ATTRIBUTE_WARN_UNUSED_RESULT
- #define STARPU_ATTRIBUTE_PURE
- #define STARPU_ATTRIBUTE_ALIGNED(size)
- #define **STARPU_INLINE**
- #define **endif**
- #define STARPU_WARN_UNUSED_RESULT
- #define STARPU_POISON_PTR
- #define STARPU_MIN(a, b)
- #define STARPU_MAX(a, b)
- #define STARPU_ASSERT(x)
- #define STARPU_ASSERT_MSG(x, msg,...)
- #define STARPU_ABORT()
- #define STARPU_ABORT_MSG(msg,...)
- #define STARPU_CHECK_RETURN_VALUE(err, message,...)
- #define STARPU_CHECK_RETURN_VALUE_IS(err, value, message,...)
- #define **STARPU_ATOMIC_SOMETHING**(name, expr)
- #define STARPU_RMB()
- #define STARPU_WMB()

Functions

- static __starpu_inline int starpu_get_env_number (const char *str)
- static __starpu_inline int **starpu_get_env_number_default** (const char *str, int defval)
- void **starpu_trace_user_event** (unsigned long code)
- void starpu_execute_on_each_worker (void(*func)(void *), void *arg, uint32_t where)
- void **starpu_execute_on_each_worker_ex** (void(*func)(void *), void *arg, uint32_t where, const char *name)
- void **starpu_execute_on_specific_workers** (void(*func)(void *), void *arg, unsigned num_workers, unsigned *workers, const char *name)
- int starpu_data_cpy (starpu_data_handle_t dst_handle, starpu_data_handle_t src_handle, int asynchronous, void(*callback_func)(void *), void *callback_arg)
- double starpu_timing_now (void)

19.30 starpu_worker.h File Reference

```
#include <stdlib.h>
#include <starpu_config.h>
```

Data Structures

- struct starpu_sched_ctx_iterator
- struct starpu_worker_collection

Enumerations

- enum starpu_worker_archtype { STARPU_ANY_WORKER, STARPU_CPU_WORKER, STARPU_CUDA_↩ WORKER, STARPU_OPENCL_WORKER }
- enum starpu_worker_collection_type { STARPU_WORKER_LIST }

Functions

- unsigned starpu_worker_get_count (void)
- unsigned starpu_combined_worker_get_count (void)
- unsigned **starpu_worker_is_combined_worker** (int id)
- unsigned starpu_cpu_worker_get_count (void)
- unsigned starpu_cuda_worker_get_count (void)
- unsigned starpu_opencl_worker_get_count (void)
- int starpu_worker_get_id (void)
- int starpu_combined_worker_get_id (void)
- int starpu_combined_worker_get_size (void)
- int starpu_combined_worker_get_rank (void)
- enum starpu_worker_archtype starpu_worker_get_type (int id)
- int starpu_worker_get_count_by_type (enum starpu_worker_archtype type)
- int starpu_worker_get_ids_by_type (enum starpu_worker_archtype type, int ∗workerids, int maxsize)
- int starpu_worker_get_by_type (enum starpu_worker_archtype type, int num)
- int starpu_worker_get_by_devid (enum starpu_worker_archtype type, int devid)
- void starpu_worker_get_name (int id, char ∗dst, size_t maxlen)
- int starpu_worker_get_devid (int id)
- int **starpu_worker_get_nsched_ctxs** (int workerid)

19.31 starpu_mpi.h File Reference

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

Typedefs

- typedef void ∗ **starpu_mpi_req**

Functions

Communication

- int starpu_mpi_isend (starpu_data_handle_t data_handle, starpu_mpi_req *req, int dest, int mpi_tag, MPI_Comm comm)
- int starpu_mpi_irecv (starpu_data_handle_t data_handle, starpu_mpi_req *req, int source, int mpi_tag, MPI_Comm comm)
- int starpu_mpi_send (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm)
- int starpu_mpi_recv (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, MPI_Status *status)
- int starpu_mpi_isend_detached (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm, void(*callback)(void *), void *arg)
- int starpu_mpi_irecv_detached (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, void(*callback)(void *), void *arg)
- int starpu_mpi_wait (starpu_mpi_req *req, MPI_Status *status)
- int starpu_mpi_test (starpu_mpi_req *req, int *flag, MPI_Status *status)
- int starpu_mpi_barrier (MPI_Comm comm)
- int starpu_mpi_isend_detached_unlock_tag (starpu_data_handle_t data_handle, int dest, int mpi_tag, MPI_Comm comm, starpu_tag_t tag)
- int starpu_mpi_irecv_detached_unlock_tag (starpu_data_handle_t data_handle, int source, int mpi_tag, MPI_Comm comm, starpu_tag_t tag)
- int starpu_mpi_isend_array_detached_unlock_tag (unsigned array_size, starpu_data_handle_t *data_handle, int *dest, int *mpi_tag, MPI_Comm *comm, starpu_tag_t tag)
- int starpu_mpi_irecv_array_detached_unlock_tag (unsigned array_size, starpu_data_handle_t *data_handle, int *source, int *mpi_tag, MPI_Comm *comm, starpu_tag_t tag)

Initialisation

- int starpu_mpi_init (int *argc, char ***argv, int initialize_mpi)
- int starpu_mpi_initialize (void)
- int starpu_mpi_initialize_extended (int *rank, int *world_size)
- int starpu_mpi_shutdown (void)
- void starpu_mpi_comm_amounts_retrieve (size_t *comm_amounts)

MPI Insert Task

- int starpu_mpi_insert_task (MPI_Comm comm, struct starpu_codelet *codelet,...)
- void starpu_mpi_get_data_on_node (MPI_Comm comm, starpu_data_handle_t data_handle, int node)
- void starpu_mpi_get_data_on_node_detached (MPI_Comm comm, starpu_data_handle_t data_handle, int node, void(*callback)(void *), void *arg)
- void starpu_mpi_data_register (starpu_data_handle_t data_handle, int tag, int rank)

Collective Operations

- void starpu_mpi_redux_data (MPI_Comm comm, starpu_data_handle_t data_handle)
- int starpu_mpi_scatter_detached (starpu_data_handle_t *data_handles, int count, int root, MPI_Comm comm, void(*scallback)(void *), void *sarg, void(*rcallback)(void *), void *rarg)
- int starpu_mpi_gather_detached (starpu_data_handle_t *data_handles, int count, int root, MPI_Comm comm, void(*scallback)(void *), void *sarg, void(*rcallback)(void *), void *rarg)

Communication Cache

- void starpu_mpi_cache_flush (MPI_Comm comm, starpu_data_handle_t data_handle)
- void starpu_mpi_cache_flush_all_data (MPI_Comm comm)

19.32 sc_hypervisor.h File Reference

```
#include <starpu.h>
#include <starpu_sched_ctx_hypervisor.h>
#include <sc_hypervisor_config.h>
#include <sc_hypervisor_monitoring.h>
#include <math.h>
```

Data Structures

- struct sc_hypervisor_policy

Functions

- void * sc_hypervisor_init (struct sc_hypervisor_policy *policy)
- void sc_hypervisor_shutdown (void)
- void sc_hypervisor_register_ctx (unsigned sched_ctx, double total_flops)
- void sc_hypervisor_unregister_ctx (unsigned sched_ctx)
- void sc_hypervisor_post_resize_request (unsigned sched_ctx, int task_tag)
- void **sc_hypervisor_resize_ctxs** (unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void sc_hypervisor_stop_resize (unsigned sched_ctx)
- void sc_hypervisor_start_resize (unsigned sched_ctx)
- const char * sc_hypervisor_get_policy ()
- void sc_hypervisor_add_workers_to_sched_ctx (int *workers_to_add, unsigned nworkers_to_add, unsigned sched_ctx)
- void sc_hypervisor_remove_workers_from_sched_ctx (int *workers_to_remove, unsigned nworkers_to_remove, unsigned sched_ctx, unsigned now)
- void sc_hypervisor_move_workers (unsigned sender_sched_ctx, unsigned receiver_sched_ctx, int *workers_to_move, unsigned nworkers_to_move, unsigned now)
- void sc_hypervisor_size_ctxs (unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- unsigned sc_hypervisor_get_size_req (unsigned **sched_ctxs, int *nsched_ctxs, int **workers, int *nworkers)
- void sc_hypervisor_save_size_req (unsigned *sched_ctxs, int nsched_ctxs, int *workers, int nworkers)
- void sc_hypervisor_free_size_req (void)
- unsigned sc_hypervisor_can_resize (unsigned sched_ctx)
- void sc_hypervisor_set_type_of_task (struct starpu_codelet *cl, unsigned sched_ctx, uint32_t footprint, size_t data_size)
- void sc_hypervisor_update_diff_total_flops (unsigned sched_ctx, double diff_total_flops)
- void sc_hypervisor_update_diff_elapsed_flops (unsigned sched_ctx, double diff_task_flops)
- void **sc_hypervisor_update_resize_interval** (unsigned *sched_ctxs, int nsched_ctxs)

Variables

- starpu_pthread_mutex_t **act_hypervisor_mutex**

19.33 sc_hypervisor_config.h File Reference

```
#include <sc_hypervisor.h>
```

Data Structures

- struct sc_hypervisor_policy_config

Macros

- #define SC_HYPERVISOR_MAX_IDLE
- #define **SC_HYPERVISOR_MIN_WORKING**
- #define SC_HYPERVISOR_PRIORITY
- #define SC_HYPERVISOR_MIN_WORKERS
- #define SC_HYPERVISOR_MAX_WORKERS
- #define SC_HYPERVISOR_GRANULARITY
- #define SC_HYPERVISOR_FIXED_WORKERS
- #define SC_HYPERVISOR_MIN_TASKS
- #define SC_HYPERVISOR_NEW_WORKERS_MAX_IDLE
- #define SC_HYPERVISOR_TIME_TO_APPLY
- #define SC_HYPERVISOR_NULL
- #define SC_HYPERVISOR_ISPEED_W_SAMPLE
- #define SC_HYPERVISOR_ISPEED_CTX_SAMPLE
- #define **SC_HYPERVISOR_TIME_SAMPLE**
- #define **MAX_IDLE_TIME**
- #define **MIN_WORKING_TIME**

Functions

- void sc_hypervisor_set_config (unsigned sched_ctx, void ∗config)
- struct sc_hypervisor_policy_config ∗ sc_hypervisor_get_config (unsigned sched_ctx)
- void sc_hypervisor_ctl (unsigned sched_ctx,...)

19.34 sc_hypervisor_lp.h File Reference

```
#include <sc_hypervisor.h>
#include <starpu_config.h>
```

Functions

- double **sc_hypervisor_lp_get_nworkers_per_ctx** (int nsched_ctxs, int ntypes_of_workers, double res[nsched_ctxs][ntypes_of_workers], int total_nw[ntypes_of_workers], struct types_of_workers ∗tw)
- double **sc_hypervisor_lp_get_tmax** (int nw, int ∗workers)
- void **sc_hypervisor_lp_round_double_to_int** (int ns, int nw, double res[ns][nw], int res_rounded[ns][nw])
- void **sc_hypervisor_lp_redistribute_resources_in_ctxs** (int ns, int nw, int res_rounded[ns][nw], double res[ns][nw], unsigned ∗sched_ctxs, struct types_of_workers ∗tw)
- void **sc_hypervisor_lp_distribute_resources_in_ctxs** (unsigned ∗sched_ctxs, int ns, int nw, int res_↩ rounded[ns][nw], double res[ns][nw], int ∗workers, int nworkers, struct types_of_workers ∗tw)
- void **sc_hypervisor_lp_place_resources_in_ctx** (int ns, int nw, double w_in_s[ns][nw], unsigned ∗sched↩ _ctxs, int ∗workers, unsigned do_size, struct types_of_workers ∗tw)
- void **sc_hypervisor_lp_share_remaining_resources** (int ns, unsigned ∗sched_ctxs, int nworkers, int ∗workers)
- double **sc_hypervisor_lp_find_tmax** (double t1, double t2)
- unsigned **sc_hypervisor_lp_execute_dichotomy** (int ns, int nw, double w_in_s[ns][nw], unsigned solve↩ _lp_integer, void ∗specific_data, double tmin, double tmax, double smallest_tmax, double(∗lp_estimated_↩ distrib_func)(int ns, int nw, double draft_w_in_s[ns][nw], unsigned is_integer, double tmax, void ∗specifc_↩ data))

19.35 sc_hypervisor_monitoring.h File Reference

```
#include <sc_hypervisor.h>
```

Data Structures

- struct sc_hypervisor_resize_ack
- struct sc_hypervisor_wrapper

Functions

- struct sc_hypervisor_wrapper * sc_hypervisor_get_wrapper (unsigned sched_ctx)
- unsigned * sc_hypervisor_get_sched_ctxs ()
- int sc_hypervisor_get_nsched_ctxs ()
- int **sc_hypervisor_get_nworkers_ctx** (unsigned sched_ctx, enum starpu_worker_archtype arch)
- double sc_hypervisor_get_elapsed_flops_per_sched_ctx (struct sc_hypervisor_wrapper *sc_w)
- double **sc_hypervisor_get_total_elapsed_flops_per_sched_ctx** (struct sc_hypervisor_wrapper *sc_w)
- double **sc_hypervisorsc_hypervisor_get_speed_per_worker_type** (struct sc_hypervisor_wrapper *sc_w, enum starpu_worker_archtype arch)
- double **sc_hypervisor_get_speed** (struct sc_hypervisor_wrapper *sc_w, enum starpu_worker_archtype arch)

19.36 sc_hypervisor_policy.h File Reference

```
#include <sc_hypervisor.h>
```

Data Structures

- struct types_of_workers
- struct sc_hypervisor_policy_task_pool

Macros

- #define **HYPERVISOR_REDIM_SAMPLE**
- #define **HYPERVISOR_START_REDIM_SAMPLE**
- #define **SC_NOTHING**
- #define **SC_IDLE**
- #define **SC_SPEED**

Functions

- void **sc_hypervisor_policy_add_task_to_pool** (struct starpu_codelet *cl, unsigned sched_ctx, uint32_t footprint, struct sc_hypervisor_policy_task_pool **task_pools, size_t data_size)
- void **sc_hypervisor_policy_remove_task_from_pool** (struct starpu_task *task, uint32_t footprint, struct sc_hypervisor_policy_task_pool **task_pools)
- struct sc_hypervisor_policy_task_pool * **sc_hypervisor_policy_clone_task_pool** (struct sc_hypervisor_policy_task_pool *tp)

- void **sc_hypervisor_get_tasks_times** (int nw, int nt, double times[nw][nt], int *workers, unsigned size_ctxs, struct sc_hypervisor_policy_task_pool *task_pools)
- unsigned **sc_hypervisor_find_lowest_prio_sched_ctx** (unsigned req_sched_ctx, int nworkers_to_move)
- int * **sc_hypervisor_get_idlest_workers** (unsigned sched_ctx, int *nworkers, enum starpu_worker_archtype arch)
- int * **sc_hypervisor_get_idlest_workers_in_list** (int *start, int *workers, int nall_workers, int *nworkers, enum starpu_worker_archtype arch)
- int **sc_hypervisor_get_movable_nworkers** (struct sc_hypervisor_policy_config *config, unsigned sched_ctx, enum starpu_worker_archtype arch)
- int **sc_hypervisor_compute_nworkers_to_move** (unsigned req_sched_ctx)
- unsigned **sc_hypervisor_policy_resize** (unsigned sender_sched_ctx, unsigned receiver_sched_ctx, unsigned force_resize, unsigned now)
- unsigned **sc_hypervisor_policy_resize_to_unknown_receiver** (unsigned sender_sched_ctx, unsigned now)
- double **sc_hypervisor_get_ctx_speed** (struct sc_hypervisor_wrapper *sc_w)
- double **sc_hypervisor_get_slowest_ctx_exec_time** (void)
- double **sc_hypervisor_get_fastest_ctx_exec_time** (void)
- double **sc_hypervisor_get_speed_per_worker** (struct sc_hypervisor_wrapper *sc_w, unsigned worker)
- double **sc_hypervisor_get_speed_per_worker_type** (struct sc_hypervisor_wrapper *sc_w, enum starpu_worker_archtype arch)
- double **sc_hypervisor_get_ref_speed_per_worker_type** (struct sc_hypervisor_wrapper *sc_w, enum starpu_worker_archtype arch)
- void **sc_hypervisor_group_workers_by_type** (struct types_of_workers *tw, int *total_nw)
- enum starpu_worker_archtype **sc_hypervisor_get_arch_for_index** (unsigned w, struct types_of_workers *tw)
- unsigned **sc_hypervisor_get_index_for_arch** (enum starpu_worker_archtype arch, struct types_of_workers *tw)
- unsigned **sc_hypervisor_criteria_fulfilled** (unsigned sched_ctx, int worker)
- unsigned **sc_hypervisor_check_idle** (unsigned sched_ctx, int worker)
- unsigned **sc_hypervisor_check_speed_gap_btw_ctxs** (void)
- unsigned **sc_hypervisor_get_resize_criteria** ()
- struct types_of_workers * **sc_hypervisor_get_types_of_workers** (int *workers, unsigned nworkers)

19.36.1 Data Structure Documentation

19.36.1.1 struct types_of_workers

Data Fields

unsigned	ncpus	
unsigned	ncuda	
unsigned	nw	

Chapter 20

Deprecated List

Global starpu_codelet::cpu_func

Optional field which has been made deprecated. One should use instead the field starpu_codelet::cpu_funcs.

Global starpu_codelet::cuda_func

Optional field which has been made deprecated. One should use instead the starpu_codelet::cuda_funcs field.

Global starpu_codelet::opencl_func

Optional field which has been made deprecated. One should use instead the starpu_codelet::opencl_funcs field.

Global starpu_data_free_pinned_if_possible

Equivalent to starpu_free(). This macro is provided to avoid breaking old codes.

Global starpu_data_malloc_pinned_if_possible

Equivalent to starpu_malloc(). This macro is provided to avoid breaking old codes.

Global starpu_mpi_initialize (void)

This function has been made deprecated. One should use instead the function starpu_mpi_init(). This function does not call MPI_Init(), it should be called beforehand.

Global starpu_mpi_initialize_extended (int *rank, int *world_size)

This function has been made deprecated. One should use instead the function starpu_mpi_init(). MPI will be initialized by starpumpi by calling `MPI_Init_Thread(argc, argv, MPI_THREAD_SERIALIZED, ...)`.

Global STARPU_MULTIPLE_CPU_IMPLEMENTATIONS

Setting the field starpu_codelet::cpu_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::cpu_funcs.

Global STARPU_MULTIPLE_CUDA_IMPLEMENTATIONS

Setting the field starpu_codelet::cuda_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::cuda_funcs.

Global STARPU_MULTIPLE_OPENCL_IMPLEMENTATIONS

Setting the field starpu_codelet::opencl_func with this macro indicates the codelet will have several implementations. The use of this macro is deprecated. One should always only define the field starpu_codelet::opencl_funcs.

Global starpu_perfmodel::cost_model)(struct starpu_data_descr *)

This field is deprecated. Use instead the field starpu_perfmodel::cost_function field.

Global starpu_perfmodel_per_arch::cost_model)(struct starpu_data_descr *t)

This field is deprecated. Use instead the field starpu_perfmodel_per_arch::cost_function.

Global starpu_task::buffers [STARPU_NMAXBUFS]

This field has been made deprecated. One should use instead the field starpu_task::handles to specify the data handles accessed by the task. The access modes are now defined in the field starpu_codelet::modes.

Index

Part III

Appendix

Chapter 21

Full Source Code for the 'Scaling a Vector' Example

21.1 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_opencl_func(void *buffers[], void *_args);

static struct starpu_codelet cl = {
    .where = STARPU_CPU | STARPU_CUDA | STARPU_OPENCL,
    /* CPU implementation of the codelet */
    .cpu_funcs = { scal_cpu_func, scal_sse_func, NULL },
#ifdef STARPU_USE_CUDA
    /* CUDA implementation of the codelet */
    .cuda_funcs = { scal_cuda_func, NULL },
#endif
#ifdef STARPU_USE_OPENCL
    /* OpenCL implementation of the codelet */
    .opencl_funcs = { scal_opencl_func, NULL },
#endif
    .nbuffers = 1,
    .modes = { STARPU_RW }
};

#ifdef STARPU_USE_OPENCL
struct starpu_opencl_program programs;
#endif

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

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

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

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

    /* Tell StaPU to associate the "vector" vector with the "vector_handle"
     * identifier. When a task needs to access a piece of data, it should
     * refer to the handle that is associated to it.
     * In the case of the "vector" data interface:
```

```
     *  - the first argument of the registration method is a pointer to the
     *    handle that should describe the data
     *  - the second argument is the memory node where the data (ie. "vector")
     *    resides initially: 0 stands for an address in main memory, as
     *    opposed to an adress on a GPU for instance.
     *  - the third argument is the adress of the vector in RAM
     *  - the fourth argument is the number of elements in the vector
     *  - the fifth argument is the size of each element.
     */
    starpu_data_handle_t vector_handle;
    starpu_vector_data_register(&vector_handle, 0, (uintptr_t)vector,
                                NX, sizeof(vector[0]));

    float factor = 3.14;

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

    task->cl = &cl;

    /* the codelet manipulates one buffer in RW mode */
    task->handles[0] = vector_handle;

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

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

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

#ifdef STARPU_USE_OPENCL
    starpu_opencl_unload_opencl(&programs);
#endif

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

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

    return 0;
}
```

21.2 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];
        }
    }
}
```

21.3 CUDA Kernel

```
#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);

        cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

21.4 OpenCL Kernel

21.4.1 Invoking the Kernel

```
#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;

        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);
    }
}
```

21.4.2 Source of the Kernel

```
__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;
        }
}
```

Chapter 22

GNU Free Documentation License

Version 1.3, 3 November 2008

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. Secondarily, 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” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

    “CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    “Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

22.1 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.