StarPU Handbook - StarPU Extensions

for StarPU 1.4.9

Generated by Doxygen.

This manual documents the usage of StarPU version 1.4.9. Its contents was last updated on 2025-10-24.

Chapter 1

Organization

This part explains the advanced concepts of StarPU. It is intended for users whose applications need more than basic task submission.
You can learn more knowledge about some important and core concepts in StarPU:

- After reading Chapter TasksInStarPU, you can get more information about how to manage tasks in StarPU in Chapter Advanced Tasks In StarPU.
- After reading Chapter DataManagement, you can know more about how to manage the data layout of your applications in Chapter Advanced Data Management.
- After reading Chapter Scheduling, you can get some advanced scheduling policies in StarPU in Chapters Advanced Scheduling, Scheduling Contexts and Scheduling Context Hypervisor.
- Chapter How To Define A New Scheduling Policy explains how to define a StarPU task scheduling policy either in a basic monolithic way, or in a modular way.

Other chapters cover some further usages of StarPU.

- Chapters CUDA Support and OpenCL Support show how to use GPU devices with CUDA or OpenCL. Chapter Maxeler FPGA Support explains how StarPU support Field Programmable Gate Array (FPGA) applications exploiting DFE configurations.
- If you need to store more data than what the main memory (RAM) can store, Chapter Out Of Core presents how to add a new memory node on a disk and how to use it.
- Chapter MPI Support shows how to integrate MPI processes in StarPU.
- Chapter TCP/IP Support shows a TCP/IP master slave mechanism which can execute application across many remote cores without thinking about data distribution.
- Chapter Transactions shows how to cancel a sequence of already submitted tasks based on a just-in-time decision.
- Chapter Fault Tolerance explains how StarPU provide supports for failure of tasks or even failure of complete nodes.
- Chapter FFT Support explains how StarPU provides a similar library to both `fftw` and `cufft`, but by adding a support from both CPUs and GPUs.
- Chapter SOCL OpenCL Extensions explains how OpenCL applications can transparently be run using StarPU, by givings unified access to every available OpenCL device.
- We propose a hierarchical tasks model in Chapter Hierarchical DAGS to enable tasks subgraphs at runtime for a more dynamic task graph.
- You can find how to partition a machine into parallel workers in Chapter Creating Parallel Workers On A Machine.
- Chapter Interoperability Support shows how StarPU can coexist with other parallel software elements without resulting in computing core oversubscription or undersubscription.

- Chapter SimGrid Support shows you how to simulate execution on an arbitrary platform.
- Tools to help debugging applications are presented in Chapter Debugging Tools.

And finally, chapter Helpers gives a list of StarPU utility functions.

Chapter 2

Advanced Tasks In StarPU

2.1 Task Dependencies

2.1.1 Sequential Consistency

By default, task dependencies are inferred from data dependency (sequential coherency) by StarPU. The application can however disable sequential coherency for some data, and dependencies can be specifically expressed.
Setting (or unsetting) sequential consistency can be done at the data level by calling starpu_data_set_sequential_consistency_flag() for a specific data (an example is in the file `examples/dependency/task_end_dep.c`) or starpu_data_set_default_sequential for all data (an example is in the file `tests/main/subgraph_repeat.c`).
The sequential consistency mode can also be gotten by calling starpu_data_get_sequential_consistency_flag() for a specific data or get the default sequential consistency flag by calling starpu_data_get_default_sequential_consistency_flag().
Setting (or unsetting) sequential consistency can also be done at task level by setting the field starpu_task::sequential_consistency to `0` (an example is in the file `tests/main/deploop.c`).
Sequential consistency can also be set (or unset) for each handle of a specific task, this is done by using the field starpu_task::handles_sequential_consistency. When set, its value should be an array with the number of elements being the number of handles for the task, each element of the array being the sequential consistency for the `i-th` handle of the task. The field can easily be set when calling starpu_task_insert() with the flag STARPU_HANDLES_SEQUENTIAL_CONSISTENCY

```
char *seq_consistency = malloc(cl.nbuffers * sizeof(char));
seq_consistency[0] = 1;
seq_consistency[1] = 1;
seq_consistency[2] = 0;
ret = starpu_task_insert(&cl,
    STARPU_RW, handleA, STARPU_RW, handleB, STARPU_RW, handleC,
    STARPU_HANDLES_SEQUENTIAL_CONSISTENCY, seq_consistency,
    0);
free(seq_consistency);
```

A full code example is available in the file `examples/dependency/sequential_consistency.c`.
The internal algorithm used by StarPU to set up implicit dependency is as follows:

```
if (sequential_consistency(task) == 1)
    for(i=0 ; i<STARPU_TASK_GET_NBUFFERS(task) ; i++)
      if (sequential_consistency(i-th data, task) == 1)
        if (sequential_consistency(i-th data) == 1)
           create_implicit_dependency(...)
```

2.1.2 Tasks And Tags Dependencies

One can explicitly set dependencies between tasks using starpu_task_declare_deps() or starpu_task_declare_deps_array().
Dependencies between tasks can be expressed through tags associated to a tag with the field starpu_task::tag_id and using the function starpu_tag_declare_deps() or starpu_tag_declare_deps_array(). The example `tests/main/tag_task_data_deps.c` shows how to set dependencies between tasks with different functions.
The termination of a task can be delayed through the function starpu_task_end_dep_add() which specifies the number of calls to the function starpu_task_end_dep_release() needed to trigger the task termination. One can also use starpu_task_declare_end_deps() or starpu_task_declare_end_deps_array() to delay the termination of a task until the termination of other tasks. A simple example is available in the file `tests/main/task_end_↩dep.c`.
starpu_tag_notify_from_apps() can be used to explicitly unlock a specific tag, but if it is called several times on the

same tag, notification will be done only on first call. However, one can call starpu_tag_restart() to clear the already notified status of a tag which is not associated with a task, and then calling starpu_tag_notify_from_apps() again will notify the successors. Alternatively, starpu_tag_notify_restart_from_apps() can be used to atomically call both starpu_tag_notify_from_apps() and starpu_tag_restart() on a specific tag.
To get the task associated to a specific tag, one can call starpu_tag_get_task(). Once the corresponding task has been executed and when there is no other tag that depend on this tag anymore, one can call starpu_tag_remove() to release the resources associated to the specific tag. One can use starpu_tag_clear() to clear all the tags (but it requires that no starpu_tag_wait_array() call is currently pending).

2.2 Waiting For Tasks

StarPU provides several advanced functions to wait for termination of tasks. One can wait for some explicit tasks, or for some tag attached to some tasks, or for some data results.
starpu_task_wait_array() is a function that waits for an array of tasks to complete their execution. starpu_task_wait_for_all_in_ctx() is a function that waits for all tasks in a specific context to complete their execution. starpu_task_wait_for_n_submitted_in_ctx() is a function that waits for a specified number of tasks to be submitted to a specific context. starpu_task_wait_for_no_ready() is a function that waits for all tasks to become unready, which means that they are either completed or blocked on a data dependency. In order to successfully call these functions to wait for termination of tasks, starpu_task::detach should be set to 0 before task submission.
The function starpu_task_nready() returns the number of tasks that are ready to execute, which means that all their data dependencies are satisfied and they are waiting to be scheduled, while the function starpu_task_nsubmitted() returns the number of tasks that have been submitted and not completed yet.
The function starpu_task_finished() can be used to determine whether a specific task has completed its execution. starpu_tag_wait() and starpu_tag_wait_array() are two blocking functions that can be used to wait for tasks with specific tags to complete their execution. The former one waits for a specified task to complete while the latter one waits for a group of tasks to complete.
When using e.g. starup_task_insert(), it may be more convenient to wait for the *result* of a task rather than waiting for a given task explicitly. That can be done thanks to starpu_data_acquire() or starpu_data_acquire_cb() that wait for the result to be available in the home node of the data. That will thus wait for all the tasks that lead to that result. One can also use starpu_data_acquire_on_node() and give it STARPU_ACQUIRE_NO_NODE to tell to just wait for tasks to complete, but not wait for the data to be available in the home node. One can also use starpu_data_acquire_try() or starpu_data_acquire_on_node_try() to just test for the termination.
If a task is created by using starpu_task_create() or starpu_task_insert(), the field starpu_task::destroy is set to 1 by default, which means that the task structure will be automatically freed after termination. On the other hand, if the task is initialized by using starpu_task_init(), the field starpu_task::destroy is set to 0 by default, which means that the task structure will not be freed until starpu_task_destroy() is called explicitly. Otherwise, we can manually set starpu_task::destroy to 1 before submission or call starpu_task_set_destroy() after submission to activate the automatic freeing of the task structure.

2.3 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 =
{
    .cpu_funcs = { scal_cpu_func, scal_sse_func },
    .cpu_funcs_name = { "scal_cpu_func", "scal_sse_func" },
    .nbuffers = 1,
    .modes = { STARPU_RW }
```

```
};
```

The full code of this example is available in the file `examples/basic_examples/vector_scal.c`.
Schedulers which are multi-implementation aware (only `dmda` and `pheft` for now) will use the performance models of all the provided implementations, and pick the one which seems to be the fastest.

2.4 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 =
{
    .can_execute = can_execute,
    .cpu_funcs = { cpu_func },
    .cpu_funcs_name = { "cpu_func" },
    .cuda_funcs = { gpu_func }
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

A full example is available in the file `examples/reductions/dot_product.c`.
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 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 =
{
    .can_execute = can_execute,
    .cpu_funcs = { cpu_func },
    .cpu_funcs_name = { "cpu_func" },
    .cuda_funcs = { scal_gpu_13, scal_gpu_20 },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

Another example is having specialized implementations for some given common sizes, for instance here we have a specialized implementation for 1024x1024 matrices:

```
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 */
  switch (nimpl)
  {
    case 0:
      /* Trying to execute the generic capability variant.    */
      return 1;
    case 1:
    {
      /* Trying to execute the size == 1024 specific variant.    */
      struct starpu_matrix_interface *interface = starpu_data_get_interface_on_node(task->handles[0]);
      return STARPU_MATRIX_GET_NX(interface) == 1024 && STARPU_MATRIX_GET_NY(interface == 1024);
    }
  }
}
struct starpu_codelet cl =
{
    .can_execute = can_execute,
    .cpu_funcs = { cpu_func },
    .cpu_funcs_name = { "cpu_func" },
    .cuda_funcs = { potrf_gpu_generic, potrf_gpu_1024 },
    .nbuffers = 1,
    .modes = { STARPU_RW }
};
```

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

2.5 Getting Task Children

It may be interesting to get the list of tasks which depend on a given task, notably when using implicit dependencies, since this list is computed by StarPU. starpu_task_get_task_succs() or starpu_task_get_task_scheduled_succs() provides it. For instance:

```
struct starpu_task *tasks[4];
ret = starpu_task_get_task_succs(task, sizeof(tasks)/sizeof(*tasks), tasks);
```

And the full example of getting task children is available in the file `tests/main/get_children_tasks.c`

2.6 Parallel Tasks

StarPU can leverage existing parallel computation libraries by the means of parallel tasks. A parallel task is a task which is run 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 accommodate with existing usages.

2.6.1 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},
    .cpu_funcs_name = {"scal_cpu_func"},
    .nbuffers = 1,
};
```

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

2.6.2 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 },
    .type = STARPU_SPMD,
    .max_parallelism = INT_MAX,
    .cpu_funcs = { func },
    .cpu_funcs_name = { "func" },
    .nbuffers = 1,
}
```

A full example is available in `examples/spmd/vector_scal_spmd.c`.
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.

2.6.3 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. Examples using parallel-task-aware StarPU scheduler are available in `tests/parallel_tasks/parallel_kernels.c` and `tests/parallel_tasks/parallel_kernels_spmd.c`.
This is however for now only proof of concept, and has not really been optimized yet.

2.6.4 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`).

2.6.5 Concurrent Parallel Tasks

Unfortunately, many environments and libraries 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 try varying combined worker sizes to look for the most efficient ones. A full example is available in `examples/spmd/vector_scal_spmd.c`.

2.7 Synchronization Tasks

For the application convenience, it may be useful to define tasks which do not actually make any computation, but wear for instance dependencies between other tasks or tags, or to be submitted in callbacks, etc.
The obvious way is of course to make kernel functions empty, but such task will thus have to wait for a worker to become ready, transfer data, etc.
A much lighter way to define a synchronization task is to set its field starpu_task::cl to `NULL`. The task will thus be a mere synchronization point, without any data access or execution content: as soon as its dependencies become available, it will terminate, call the callbacks, and release dependencies.
An intermediate solution is to define a codelet with its field starpu_codelet::where set to STARPU_NOWHERE, for instance:

```
struct starpu_codelet cl =
{
    .where = STARPU_NOWHERE,
    .nbuffers = 1,
    .modes = { STARPU_R },
}
task = starpu_task_create();
task->cl = &cl;
task->handles[0] = handle;
starpu_task_submit(task);
```

will create a task which simply waits for the value of `handle` to be available for read. This task can then be depended on, etc. A full example is available in `examples/filters/fmultiple_manual.c`.
StarPU provides starpu_task_create_sync() to create a new synchronization task, the same as the previous example but without submitting the task. The function starpu_create_sync_task() is also used to create a new synchronization task and submit it, which is a task that waits for specific tags and calls the specified callback function when the task is finished. The function starpu_create_callback_task() can create and submit a synchronization task, which is a task that completes immediately and calls the specified callback function right after.

Chapter 3

Advanced Data Management

3.1 Data Interface with Variable Size

Besides the data interfaces already available in StarPU, mentioned in DataInterface, tasks are actually allowed to change the size of data interfaces.
The simplest case is just changing the amount of data actually used within the allocated buffer. This is for instance implemented for the matrix interface: one can set the new NX/NY values with STARPU_MATRIX_SET_NX(), STARPU_MATRIX_SET_NY(), and STARPU_MATRIX_SET_LD() at the end of the task implementation. Data transfers achieved by StarPU will then use these values instead of the whole allocated size. The values of course need to be set within the original allocation. To reserve room for increasing the NX/↩ NY values, one can use starpu_matrix_data_register_allocsize() instead of starpu_matrix_data_register(), to specify the allocation size to be used instead of the default NX∗NY∗ELEMSIZE. It is also available for a vector by using starpu_vector_data_register_allocsize() to specify the allocation size to be used instead of the default NX∗ELEMSIZE. To support this, the data interface has to implement the functions starpu_data_interface_ops::alloc_footprint, starpu_data_interface_ops::alloc_compare, and starpu_data_interface_ops::reuse_data_
for proper StarPU allocation management. It might be useful to implement starpu_data_interface_ops::cache_data_on_node, otherwise StarPU will just call `memcpy()`.
A more involved case is changing the amount of allocated data. The task implementation can just reallocate the buffer during its execution, and set the proper new values in the interface structure, e.g. nx, ny, ld, etc. so that the StarPU core knows the new data layout. The structure starpu_data_interface_ops however then needs to have the field starpu_data_interface_ops::dontcache set to 1, to prevent StarPU from trying to perform any cached allocation, since the allocated size will vary. An example is available in `tests/datawizard/variable_size.c`. The example uses its own data interface to contain some simulation information for data growth, but the principle can be applied for any data interface.
The principle is to use starpu_malloc_on_node_flags() to make the new allocation, and use starpu_free_on_node_flags() to release any previous allocation. The flags have to be precisely like in the example:

```
unsigned workerid = starpu_worker_get_id_check();
unsigned dst_node = starpu_worker_get_memory_node(workerid);
interface->ptr = starpu_malloc_on_node_flags(dst_node, size + increase, STARPU_MALLOC_PINNED |
      STARPU_MALLOC_COUNT | STARPU_MEMORY_OVERFLOW);
starpu_free_on_node_flags(dst_node, old, size, STARPU_MALLOC_PINNED | STARPU_MALLOC_COUNT |
      STARPU_MEMORY_OVERFLOW);
interface->size += increase;
```

so that the allocated area has the expected properties and the allocation is properly accounted for.
Depending on the interface (vector, CSR, etc.) you may have to fix several fields of the data interface: e.g. both `nx` and `allocsize` for vectors, and store the pointer both in `ptr` and `dev_handle`.
Some interfaces make a distinction between the actual number of elements stored in the data and the actually allocated buffer. For instance, the vector interface uses the `nx` field for the former, and the `allocsize` for the latter. This allows for lazy reallocation to avoid reallocating the buffer every time to exactly match the actual number of elements. Computations and data transfers will use the field `nx`, while allocation functions will use the field `allocsize`. One just has to make sure that `allocsize` is always bigger or equal to `nx`.
Important note: one can not change the size of a partitioned data.

3.2 Data Management Allocation

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 (a basic example is in `examples/basic_examples/block.c`), or to use the function starpu_memory_pin() to pin memory allocated by other ways, such as local arrays (a basic example is in `examples/basic_examples/vector_scal.c`). This is needed to permit asynchronous data transfer, i.e. permit data transfer to overlap with computations. Otherwise, the trace will show that the state `DriverCopyAsync` takes a lot of time, this is because CUDA or OpenCL then reverts to synchronous transfers. Before shutting down StarPU, the application should deallocate any memory that has previously been allocated with starpu_malloc(), by calling either starpu_free() or starpu_free_noflag() which is more recommended. If the application has pinned memory using starpu_memory_pin(), it should unpin the memory using starpu_memory_unpin() before freeing the memory.

If an application requires a specific alignment constraint for memory allocations made with starpu_malloc(), it can use the starpu_malloc_set_align() function to set the alignment requirement.

The application can provide its own allocation function by calling starpu_malloc_set_hooks(). StarPU will then use them for all data handle allocations in the main memory. An example is in `examples/basic_examples/hooks.c`.

StarPU provides several functions to monitor the memory usage and availability on the system. The application can use the starpu_memory_get_used() function to monitor its own memory usage on a node, and the starpu_memory_get_total_all_nodes() function to monitor the amount of total memory on all memory nodes, and the starpu_memory_get_available_all_nodes() function to monitor the amount of available memory on all memory nodes. Additionally, the starpu_memory_get_used_all_nodes() function can be used to monitor the amount of used memory on all memory nodes.

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 this 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. An example is available in `examples/pi/pi.c`.

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

will request to always automatically broadcast the updated data to all memory nodes. An example is available in `tests/datawizard/wt_broadcast.c`.

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 scarce.

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 this data.

In the same vein, accumulation of results in the same data can become a bottleneck. The use of the mode STARPU_REDUX permits to optimize such accumulation (see DataReduction). To a lesser extent, the use of the flag STARPU_COMMUTE keeps the bottleneck (see DataCommute), but at least permits the accumulation to happen in any order.

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 anymore, 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.

One can also unregister the data handle by calling:

```
starpu_data_unregister_no_coherency(handle);
```

Different from starpu_data_unregister(), a valid copy of the data is not put back into the home node in the buffer that was initially registered.

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

```
starpu_data_deinitialize(handle);
```

So that the value will be ignored and not written back to main memory.

Or instead it can even be invalidated (the buffers containing the current value will then be freed, and reallocated only when another task writes some value to the handle):

```
starpu_data_invalidate(handle);
```

if the data transfer is asynchronous, one can use the submit versions:

```
starpu_data_deinitialize_submit(handle);
```

or

```
starpu_data_invalidate_submit(handle);
```

A basic example is available in the files `tests/datawizard/data_deinitialize.c` and `tests/datawizard/data_invalidation.c`.

3.3 Data Access

To access registered data outside tasks we can call the function starpu_data_acquire(). The access mode can be read-only mode STARPU_R, write-only mode STARPU_W, and read-write mode STARPU_RW. We will get an up-to-date copy of handle in memory located where the data was originally registered. The application can also call starpu_data_acquire_try() instead of starpu_data_acquire() to acquire the data, but if previously-submitted tasks have not completed when we ask to acquire the data, the program will crash. starpu_data_release() must be called once the application no longer needs to access the piece of data. Or call starpu_data_release_to() to partly release the piece of data acquired. We can also access registered data from a given memory node by calling the function starpu_data_acquire_on_node(), or calling starpu_data_acquire_on_node_try() if all previously-submitted tasks have completed. Correspondingly, starpu_data_release_on_node() must be called once the application no longer needs to access the piece of data and the node parameter must be exactly the same as the corresponding starpu_data_acquire_on_node() call. Or call starpu_data_release_to_on_node() to partly release the piece of data acquired.

The application may access the requested data asynchronous during the execution of callback by calling starpu_data_acquire_cb(), and by calling starpu_data_acquire_cb_sequential_consistency() with the possibility of enabling or disabling data dependencies. The callback function must call starpu_data_release() once the application no longer needs to access the piece of data. Or call starpu_data_release_to() to partly release the piece of data acquired. The application can also access registered data from a given memory node instead of main memory by calling the function starpu_data_acquire_on_node_cb(), and by calling starpu_data_acquire_on_node_cb_sequential_consistency() with the possibility of enabling or disabling data dependencies. starpu_data_release_on_node() must be called once the application no longer needs to access the piece of data. Or call starpu_data_release_to_on_node() to partly release the piece of data acquired.

3.4 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. An example is available in the file `tests/microbenchs/prefetch_data_on_node.c`. The variant starpu_data_idle_prefetch_on_node() can be used to issue the transfer only when the bus is idle. One can also call starpu_data_request_allocation() for the allocation of a piece of data on the specified memory node. We can know whether the allocation is done on the specified memory node by using starpu_data_test_if_allocated_on_node(). We can also know whether the map is done on the specified memory node by using starpu_data_test_if_mapped_on_node().

If we want higher priority to request data to be replicated to a given node as soon as possible, so that it is available there for tasks, we can call starpu_data_fetch_on_node(). We can call starpu_data_prefetch_on_node_prio() to have a priority than starpu_data_prefetch_on_node(). And call starpu_data_idle_prefetch_on_node_prio() to have a bit higher priority than starpu_data_idle_prefetch_on_node().

Conversely, one can advise StarPU that some data will not be useful in the close future by calling starpu_data_wont_use(). StarPU will then write its value back to its home node, and evict it from GPUs when room is needed. An example is available in the file `tests/datawizard/partition_wontuse.c`. One can

also advise StarPU to evict data from the memory node directly by calling starpu_data_evict_from_node(), but it may fail if e.g. some tasks are still working on the memory node. To avoid failure one can call starpu_data_can_evict() to check whether data can be evicted from the memory node. Anyway it is more recommended to use starpu_data_wont_use().
One can query the status of handle on the specified memory node by calling starpu_data_query_status2() or starpu_data_query_status(). One can call starpu_memchunk_tidy() to tidy the available memory on the specified memory node periodically.

3.5 Manual Partitioning

Except the partitioning functions described in PartitioningData and AsynchronousPartitioning, one can also handle partitioning by hand, by registering several views on the same piece of data. The idea is then to manage the coherency of the various views through the common buffer in the main memory. `examples/filters/fmultiple_manual.c` is a complete example using this technique.
In short, we first register the same matrix several times:

```
starpu_matrix_data_register(&handle, STARPU_MAIN_RAM, (uintptr_t)matrix, NX, NX, NY, sizeof(matrix[0]));
for (i = 0; i < PARTS; i++)
    starpu_matrix_data_register(&vert_handle[i], STARPU_MAIN_RAM, (uintptr_t)&matrix[0][i*(NX/PARTS)], NX,
      NX/PARTS, NY, sizeof(matrix[0][0]));
```

Since StarPU is not aware that the two handles are actually pointing to the same data, we have a danger of inadvertently submitting tasks to both views, which will bring a mess since StarPU will not guarantee any coherency between the two views. To make sure we don't do this, we invalidate the view that we will not use:

```
for (i = 0; i < PARTS; i++)
    starpu_data_invalidate(vert_handle[i]);
```

Then we can safely work on `handle`.
When we want to switch to the vertical slice view, all we need to do is bring coherency between them by running an empty task on the home node of the data:

```
struct starpu_codelet cl_switch =
{
    .where = STARPU_NOWHERE,
    .nbuffers = 3,
    .specific_nodes = 1,
    .nodes = { STARPU_MAIN_RAM, STARPU_MAIN_RAM, STARPU_MAIN_RAM },
};
ret = starpu_task_insert(&cl_switch, STARPU_RW, handle,
            STARPU_W, vert_handle[0],
            STARPU_W, vert_handle[1],
            0);
```

The execution of the task `switch` will get back the matrix data into the main memory, and thus the vertical slices will get the updated value there.
Again, we prefer to make sure that we don't accidentally access the matrix through the whole-matrix handle:

```
starpu_data_invalidate_submit(handle);
```

Note: when enabling a set of handles in this way, the set must not have any overlapping, i.e. the handles of the set must not have any part of data in common, otherwise StarPU will not properly handle concurrent accesses between them.
And now we can start using vertical slices, etc.

3.6 Data handles helpers

Functions starpu_data_set_user_data() and starpu_data_get_user_data() are used to associate user-defined data with a specific data handle. One can set or retrieve the field `user_data` of the data handle by calling these two functions respectively. Similarly, functions starpu_data_set_sched_data() and starpu_data_get_sched_data() are used to associate scheduling-related data with a specific data handle. One can set or retrieve the field `sched_data` of the data handle by calling these two functions respectively. One can set a name for a data handle by calling starpu_data_set_name().
One can call starpu_data_register_same() to register a new piece of data into a data handle with the same interface as the specified data handle. If necessary, one can register a void interface by using starpu_void_data_register(). There is no data really associated to this interface, but it may be used as a synchronization mechanism.
One can call starpu_data_cpy() or starpu_data_cpy_priority() to copy data from one memory location to another memory location, but the latter one allows the application to specify a priority value for the copy operation. The higher the priority value, the sonner the copy operation will be scheduled and executed. One can also call starpu_data_dup_ro() function for duplicating, but this function only creates a new read-only data block that is an exact copy of the original data block. The new data block can be used independently of the original data block

for read-only access.
starpu_data_pack_node() and starpu_data_pack() are functions that are used to pack a data item into a binary buffer on a node or on local memory node. starpu_data_peek_node() and starpu_data_peek() are functions that allow you to read in handle's node or local node replicate the data located at the given pointer. starpu_data_unpack_node() and starpu_data_unpack() are functions that are used to unpack a data item from a binary buffer on a node or on local memory node.
StarPU provides several functions for querying the size and memory allocation of variable size data items, such as: starpu_data_get_size() is a function that returns the size of a data associated with handle in bytes. This is the size of the actual data stored in memory. starpu_data_get_alloc_size() is a function that returns the amount of memory that has been allocated for a data associated with handle in anticipation. This may be larger than the actual size of the data item, due to alignment requirements or other implementation details. starpu_data_get_max_size() is a function that returns the maximum size of a handle data that can be allocated by StarPU.
One can call starpu_data_get_home_node() to retrieve the identifier of the node on which the data handle is originally stored. One can call starpu_data_print() to print basic information about the data handle and the node to the specified file.

3.7 Handles data buffer pointers

A simple understanding of StarPU handles is that it's a collection of buffers on each memory node of the machine, which contain the same data. The picture is however made more complex with the OpenCL support and with partitioning.
When partitioning a handle, the data buffers of the subhandles will indeed be inside the data buffers of the main handle (to save transferring data back and forth between the main handle and the subhandles). But in OpenCL, a `cl_mem` is not a pointer, but an opaque value on which pointer arithmetic can not be used. That is why data interfaces contain three fields: `dev_handle`, `offset`, and `ptr`.

- The field `dev_handle` is what the allocation function returned, and one can not do arithmetic on it.
- The field `offset` is the offset inside the allocated area, most often it will be 0 because data start at the beginning of the allocated area, but when the handle is partitioned, the subhandles will have varying `offset` values, for each subpiece.
- The field `ptr`, in the non-OpenCL case, i.e. when pointer arithmetic can be used on `dev_handle`, is just the sum of `dev_handle` and `offset`, provided for convenience.

This means that:

- computation kernels can use `ptr` in non-OpenCL implementations.
- computation kernels have to use `dev_handle` and `offset` in the OpenCL implementation.
- allocation methods of data interfaces have to store the value returned by starpu_malloc_on_node() in `dev_handle` and `ptr`, and set `offset` to 0.
- partitioning filters have to copy over `dev_handle` without modifying it, set in the child different values of `offset`, and set `ptr` accordingly as the sum of `dev_handle` and `offset`.

We can call starpu_data_handle_to_pointer() to get `ptr` associated with the data handle, or call starpu_data_get_local_ptr() to get the local pointer associated with the data handle.
Examples in the directory `examples/interface/complex_dev_handle/` show how to generate and implement an interface supporting OpenCL.
To better notice the difference between simple `ptr` and `dev_handle + offset`, one can compare `examples/interface/complex_interface.c` vs `examples/interface/complex_dev_handle/complex_dev_handle_interface.c` and `examples/interface/complex_filters.c` vs `examples/interface/complex_dev_handle/complex_dev_handle_filters.c`.

3.8 Defining A New Data Filter

StarPU provides a series of predefined filters in Data Partition, but additional filters can be defined by the application. The principle is that the filter function just fills the memory location of the `i-th` subpart of a data. Examples are

provided in `src/datawizard/interfaces/*_filters.c`, check starpu_data_filter::filter_func for further details. The helper function starpu_filter_nparts_compute_chunk_size_and_offset() can be used to compute the division of pieces of data.

3.9 Defining A New Data Interface

This section proposes an example how to define your own interface, when the StarPU-provided interface do not fit your needs. Here we take a simple example of an array of complex numbers represented by two arrays of double values. The full source code is in `examples/interface/complex_interface.c` and `examples/interface/complex_interface.h`

Let's thus define a new data interface to manage arrays of complex numbers:

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

That structure stores enough to describe **one** buffer of such kind of data. It is used for the buffer stored in the main memory, another instance is used for the buffer stored in a GPU, etc. A *data handle* is thus a collection of such structures, to describe each buffer on each memory node.

Note: one should not make pointers that point into such structures, because StarPU needs to be able to copy over the content of it to various places, for instance to efficiently migrate a data buffer from one data handle to another data handle, so the actual address of the structure may vary.

3.9.1 Data registration

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 *handleptr,
     unsigned home_node, double *real, double *imaginary, int nx)
{
        struct starpu_complex_interface complex =
        {
                .real = real,
                .imaginary = imaginary,
                .nx = nx
        };
        starpu_data_register(handleptr, home_node, &complex, &interface_complex_ops);
}
```

The `struct starpu_complex_interface complex` is here used just to store the parameters provided by users to `starpu_complex_data_register`. starpu_data_register() will first allocate the handle, and then pass the structure `starpu_complex_interface` to the method starpu_data_interface_ops::register_data_handle, which records them within the data handle (it is called once per node by starpu_data_register()):

```
static void complex_register_data_handle(starpu_data_handle_t handle, int home_node, void *data_interface)
{
    struct starpu_complex_interface *complex_interface = (struct starpu_complex_interface *) data_interface;
    unsigned node;
    for (node = 0; node < STARPU_MAXNODES; node++)
    {
        struct starpu_complex_interface *local_interface = (struct starpu_complex_interface *)
            starpu_data_get_interface_on_node(handle, node);
        local_interface->nx = complex_interface->nx;
        if (node == home_node)
        {
            local_interface->real = complex_interface->real;
            local_interface->imaginary = complex_interface->imaginary;
        }
        else
        {
            local_interface->real = NULL;
            local_interface->imaginary = NULL;
        }
    }
}
```

If the application provided a home node, the corresponding pointers will be recorded for that node. Others have no buffer allocated yet. Possibly the interface needs some dynamic allocation (e.g. to store an array of dimensions that can have variable size). The corresponding deallocation will then be done in starpu_data_interface_ops::unregister_data_handle.

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, the details of the hooks to be provided are documented in starpu_data_interface_ops .

```
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),
};
```

The field starpu_data_interface_ops::interfaceid should be defined to STARPU_UNKNOWN_INTERFACE_ID when defining the interface, its value will be updated the first time a data is registered through the new data interface.

Convenience functions can be defined to access the different fields of the complex interface from a StarPU data handle after a call to starpu_data_acquire():

```
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, STARPU_MAIN_RAM);
        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 implementations.

```
#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, STARPU_MAIN_RAM, &real, &imaginary, 1);
starpu_task_insert(&cl_display, STARPU_R, handle1, 0);
```

and used by codelets.

```
void display_complex_codelet(void *descr[], 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/.

3.9.2 Data footprint

We need to pass a custom footprint function to the method starpu_data_interface_ops::footprint which computes data size footprint. StarPU provides several functions to compute different type of value↩ : starpu_hash_crc32c_be_n() is used to compute the CRC of a byte buffer, starpu_hash_crc32c_be_ptr() is used to compute the CRC of a pointer value, starpu_hash_crc32c_be() is used to compute the CRC of a 32bit number, starpu_hash_crc32c_string() is used to compute the CRC of a string.

3.9.3 Data allocation

To be able to run tasks on GPUs etc. StarPU needs to know how to allocate a buffer for the interface. In our example, two allocations are needed in the allocation method complex_allocate_data_on_node(): one for the real part and one for the imaginary part.

```
static starpu_ssize_t complex_allocate_data_on_node(void *data_interface, unsigned node)
{
    struct starpu_complex_interface *complex_interface = (struct starpu_complex_interface *) data_interface;
    double *addr_real = NULL;
    double *addr_imaginary = NULL;
    starpu_ssize_t requested_memory = complex_interface->nx * sizeof(complex_interface->real[0]);
    addr_real = (double*) starpu_malloc_on_node(node, requested_memory);
    if (!addr_real)
        goto fail_real;
    addr_imaginary = (double*) starpu_malloc_on_node(node, requested_memory);
    if (!addr_imaginary)
        goto fail_imaginary;
    /* update the data properly in consequence */
```

```
    complex_interface->real = addr_real;
    complex_interface->imaginary = addr_imaginary;
    return 2*requested_memory;
fail_imaginary:
    starpu_free_on_node(node, (uintptr_t) addr_real, requested_memory);
fail_real:
    return -ENOMEM;
}
```

Here we try to allocate the two parts. If either of them fails, we return `-ENOMEM`. If they succeed, we can record the obtained pointers and returned the amount of allocated memory (for memory usage accounting).

Conversely, `complex_free_data_on_node()` frees the two parts:

```
static void complex_free_data_on_node(void *data_interface, unsigned node)
{
    struct starpu_complex_interface *complex_interface = (struct starpu_complex_interface *) data_interface;
    starpu_ssize_t requested_memory = complex_interface->nx * sizeof(complex_interface->real[0]);
    starpu_free_on_node(node, (uintptr_t) complex_interface->real, requested_memory);
    starpu_free_on_node(node, (uintptr_t) complex_interface->imaginary, requested_memory);
}
```

We can call starpu_opencl_allocate_memory() to allocate memory on an OpenCL device.

We have not made anything particular for GPUs or whatsoever: it is starpu_free_on_node() which knows how to actually make the allocation, and returns the resulting pointer, be it in main memory, in GPU memory, etc.

3.9.4 Data copy

Now that StarPU knows how to allocate/free a buffer, it needs to be able to copy over data into/from it. Defining a method `copy_any_to_any()` allows StarPU to perform direct transfers between main memory and GPU memory.

```
static int copy_any_to_any(void *src_interface, unsigned src_node,
               void *dst_interface, unsigned dst_node,
               void *async_data)
{
    struct starpu_complex_interface *src_complex = src_interface;
    struct starpu_complex_interface *dst_complex = dst_interface;
    int ret = 0;
    if (starpu_interface_copy((uintptr_t) src_complex->real, 0, src_node,
                  (uintptr_t) dst_complex->real, 0, dst_node,
                   src_complex->nx*sizeof(src_complex->real[0]),
                   async_data))
        ret = -EAGAIN;
    if (starpu_interface_copy((uintptr_t) src_complex->imaginary, 0, src_node,
                  (uintptr_t) dst_complex->imaginary, 0, dst_node,
                   src_complex->nx*sizeof(src_complex->imaginary[0]),
                   async_data))
        ret = -EAGAIN;
    return ret;
}
```

We here again have no idea what is main memory or GPU memory, or even if the copy is synchronous or asynchronous: we just call starpu_interface_copy() according to the interface, passing it the pointers, and checking whether it returned `-EAGAIN`, which means the copy is asynchronous, and StarPU will appropriately wait for it thanks to the pointer `async_data`. This copy method is also available for 2D matrices starpu_interface_copy2d(), 3D matrices starpu_interface_copy3d(), 4D matrices starpu_interface_copy4d() and N-dim matrices starpu_interface_copynd().

starpu_interface_copy() will also manage copies between other devices such as CUDA devices, OpenCL devices, etc. But if necessary, we may manage these copies by ourselves as well. StarPU provides three functions starpu_cuda_copy_async_sync(), starpu_cuda_copy2d_async_sync() and starpu_cuda_copy3d_async_sync() that enable copying of 1D, 2D or 3D data between main memory and CUDA device memories. They first try to copy the data asynchronous, if fail or `stream` is `NULL` then copy the data synchronously. StarPU also provides several functions that are used to transfer data between RAM and OpenCL devices. starpu_opencl_copy_ram_to_opencl() copies data from RAM to an OpenCL device. starpu_opencl_copy_opencl_to_ram() copies data from an OpenCL device to RAM. starpu_opencl_copy_opencl_to_opencl() copies data between two OpenCL devices. starpu_opencl_copy_async_sync() copies data between two devices. If `event` is `NULL`, the copy is synchronous, and checking whether `ret` is set to `-EAGAIN`, which means the copy is asynchronous.

This copy method is referenced in a structure starpu_data_copy_methods

```
static const struct starpu_data_copy_methods complex_copy_methods =
{
    .any_to_any = copy_any_to_any
};
```

which was referenced in the structure starpu_data_interface_ops above.

Other fields of starpu_data_copy_methods allow providing optimized variants, notably for the case of 2D or 3D matrix tiles with non-trivial ld.

We can call starpu_interface_data_copy() to record in offline execution traces the copy.

When an asynchronous implementation of the data transfer is implemented, we can call starpu_interface_start_driver_copy_async() and starpu_interface_end_driver_copy_async() to initiate and complete asynchronous data transfers between main memory and GPU memory.

3.9.5 Data pack/peek/unpack

The copy methods allow for RAM/GPU transfers, but is not enough for e.g. transferring over MPI. That requires defining the pack/peek/unpack methods. The principle is that the method starpu_data_interface_ops::pack_data concatenates the buffer data into a newly-allocated contiguous bytes array, conversely starpu_data_interface_ops::peek_data extracts from a bytes array into the buffer data, and starpu_data_interface_ops::unpack_data does the same as starpu_data_interface_ops::peek_data but also frees the bytes array.

```
static int complex_pack_data(starpu_data_handle_t handle, unsigned node, void **ptr, starpu_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);
    if (ptr != NULL)
    {
        char *data;
        data = (void*) starpu_malloc_on_node_flags(node, *count, 0);
        *ptr = data;
        memcpy(data, complex_interface->real, complex_interface->nx*sizeof(double));
        memcpy(data+complex_interface->nx*sizeof(double), complex_interface->imaginary,
      complex_interface->nx*sizeof(double));
    }
    return 0;
}
```

`complex_pack_data()` first computes the size to be allocated, then allocates it, and copies over into it the content of the two real and imaginary arrays.

```
static int complex_peek_data(starpu_data_handle_t handle, unsigned node, void *ptr, size_t count)
{
    char *data = ptr;
    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);
    STARPU_ASSERT(count == 2 * complex_interface->nx * sizeof(double));
    memcpy(complex_interface->real, data, complex_interface->nx*sizeof(double));
    memcpy(complex_interface->imaginary, data+complex_interface->nx*sizeof(double),
      complex_interface->nx*sizeof(double));
    return 0;
}
```

`complex_peek_data()` simply uses `memcpy()` to copy over from the bytes array into the data buffer.

```
static int complex_unpack_data(starpu_data_handle_t handle, unsigned node, void *ptr, size_t count)
{
    complex_peek_data(handle, node, ptr, count);
    starpu_free_on_node_flags(node, (uintptr_t) ptr, count, 0);
    return 0;
}
```

And `complex_unpack_data()` just calls `complex_peek_data()` and releases the bytes array.

3.9.6 Pointers inside the data interface

In the example described above, the two pointers stored in the data interface are data buffers, which may point into main memory, GPU memory, etc. One may also want to store pointers to meta-data for the interface, for instance the list of dimensions sizes for the n-dimension matrix interface, but such pointers are to be handled completely differently. More examples are provided in `src/datawizard/interfaces/*_interface.c`
More precisely, there are two types of pointers:

- Data pointers, which point to the actual data in RAM/GPU/etc. memory. They may be NULL when the data is not allocated (yet). StarPU will automatically call starpu_data_interface_ops::allocate_data_on_node to allocate the data pointers whenever needed, and call starpu_data_interface_ops::free_data_on_node when memory gets scarce. For instance, for the n-dimension matrix interface the pointers to the actual data (`ptr`, `dev_handle`, `offset`) are data pointers.
- Meta-data pointers, which always point to RAM memory. They are usually always allocated so that they can always be used. For instance, for the n-dimension matrix interface the array of dimension sizes and the array of ld are meta-data pointers. These are typically allocated at data registration time in starpu_data_interface_ops::register_data_handle, and released at data unregistration time in starpu_data_interface_ops::unregister_data_handle

This means that:

- The starpu_data_interface_ops::register_data_handle method has to allocate the meta-data pointers. If users provided a buffer for the initial value of the handle, starpu_data_interface_ops::register_data_handle sets the data pointers of the home_node interface to that buffer.
- The interface can additionally provide a `ptr_register` helper to set the data pointer of a given node. One can call starpu_data_ptr_register() to realise.
- The starpu_data_interface_ops::unregister_data_handle method has to deallocate the meta-data pointers.
- The starpu_data_interface_ops::allocate_data_on_node method has to allocate the data pointers on the given node.
- The starpu_data_interface_ops::free_data_on_node method has to deallocate the data pointers on the given node.
- The optional starpu_data_interface_ops::cache_data_on_node transfers the data pointers from a source interface to a cached interface. If undefined, a mere memcpy is used instead. This can notably take the opportunity to clear pointers in the source interface. This also needs to copy the properties that starpu_data_interface_ops::compare (or starpu_data_interface_ops::alloc_compare if defined) needs for comparing interfaces for caching compatibility.
- The starpu_data_interface_ops::reuse_data_on_node transfers the data pointers from a cached interface to the destination interface. If undefined, a mere memcpy is used instead.
- The starpu_data_interface_ops::map_data has to map the data pointers on the given node. One should define function starpu_interface_map() to set this field.
- The starpu_data_interface_ops::unmap_data has to unmap the data pointers on the given node. One should define function starpu_interface_unmap() to set this field.
- The starpu_data_interface_ops::update_map has to update the data pointers on the given node. One should define function starpu_interface_update_map() to set this field.
- The filtering functions have to allocate the meta-data pointers for the child interface, and when the parent interface has data pointers, it has to set the child data pointers to point into the parent data buffers.

Put another way:

- starpu_data_register() initializes the handle structure and calls starpu_data_interface_ops::register_data_handle.
- Then StarPU may call starpu_data_interface_ops::allocate_data_on_node and starpu_data_interface_ops::free_data_on_node as it sees fit when it needs the data allocated on some node or not.
- Eventually, starpu_data_unregister() releases the handle buffers for all nodes (except the home node given to starpu_data_register()), which either means calling starpu_data_interface_ops::free_data_on_node (if allocation cache is disabled), or putting them into the allocation cache. It then calls starpu_data_interface_ops::unregister_data_han and releases the handle structure.

Note: for compressed matrices such as CSR, BCSR, COO, the `colind` and `rowptr` arrays are not meta-data pointers, but data pointers like `nzval`, because they need to be available in GPU memory for the GPU kernels.
Note: when the interface does not contain meta-data pointers, starpu_data_interface_ops::reuse_data_on_node does not need to be implemented, StarPU will just use a memcpy. Otherwise, either starpu_data_interface_ops::reuse_data_on_node must be used to transfer only the data pointers and not the meta-data pointers, or the allocation cache should be disabled by setting starpu_data_interface_ops::dontcache to 1.
Note: It should be noted that because of the allocation cache, starpu_data_interface_ops::free_data_on_node may be called on an interface which is not attached to a handle anymore. This means that the meta-data pointers will have been deallocated by starpu_data_interface_ops::unregister_data_handle, and cannot be used by starpu_data_interface_ops::free_data_on_node to e.g. compute the size to be deallocated. For instance, the n-dimension matrix interface uses an additional scalar allocsize field to store the allocation size, thus still available even when the interface is in the allocation cache.
Note: if starpu_data_interface_ops::unregister_data_handle is implemented and checks that pointers are NULL, starpu_data_interface_ops::cache_data_on_node needs to be implemented to clear the pointers when caching the allocation.

3.9.7 Helpers

We can get the unique identifier of the interface associated with the data handle by calling starpu_data_get_interface_id(), and get the next available identifier for a newly created data interface by calling starpu_data_interface_get_next_id().

3.10 The Multiformat Interface

It may be interesting to represent the same piece of data using two different data structures: one only used on CPUs, and one only 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. Users 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 },
    .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 },
    .cpu_funcs_name = { "opencl_to_cpu_func" },
    .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, STARPU_MAIN_RAM, &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`.

3.11 Specifying A Target Node For Task Data

When executing a task on GPU, for instance, StarPU would normally copy all the needed data for the tasks to the embedded memory of the GPU. It may however happen that the task kernel would rather have some of the data kept in the main memory instead of copied in the GPU, a pivoting vector for instance. This can be achieved by setting the flag starpu_codelet::specific_nodes to 1, and then fill the array starpu_codelet::nodes (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 STARPU_SPECIFIC_NODE_LOCAL to let StarPU copy it to the memory node where the task will be executed.
The function starpu_task_get_current_data_node() can be used to retrieve the memory node associated with the current task being executed.
STARPU_SPECIFIC_NODE_CPU can also be used to request data to be put in CPU-accessible memory (and let StarPU choose the NUMA node). STARPU_SPECIFIC_NODE_FAST and STARPU_SPECIFIC_NODE_SLOW can also be used
For instance, with the following codelet:

```
struct starpu_codelet cl =
{
    .cuda_funcs = { kernel },
    .nbuffers = 2,
    .modes = {STARPU_RW, STARPU_RW},
    .specific_nodes = 1,
    .nodes = {STARPU_SPECIFIC_NODE_CPU, STARPU_SPECIFIC_NODE_LOCAL},
};
```

the first data of the task will be kept in the CPU memory, while the second data will be copied to the CUDA GPU as usual. A working example is available in `tests/datawizard/specific_node.c`
With the following codelet:

```
struct starpu_codelet cl =
{
    .cuda_funcs = { kernel },
    .nbuffers = 2,
    .modes = {STARPU_RW, STARPU_RW},
    .specific_nodes = 1,
    .nodes = {STARPU_SPECIFIC_NODE_LOCAL, STARPU_SPECIFIC_NODE_SLOW},
};
```

The first data will be copied into fast (but probably size-limited) local memory, while the second data will be left in slow (but large) memory. This makes sense when the kernel does not make so many accesses to the second data, and thus data being remote e.g. over a PCI bus is not a performance problem, and avoids filling the fast local memory with data which does not need the performance.
In cases where the kernel is fine with some data being either local or in the main memory, STARPU_SPECIFIC_NODE_LOCAL_OR_C can be used. StarPU will then be free to leave the data in the main memory and let the kernel access it from accelerators, or to move it to the accelerator before starting the kernel, for instance:

```
struct starpu_codelet cl =
{
    .cuda_funcs = { kernel },
    .nbuffers = 2,
    .modes = {STARPU_RW, STARPU_R},
    .specific_nodes = 1,
    .nodes = {STARPU_SPECIFIC_NODE_LOCAL, STARPU_SPECIFIC_NODE_LOCAL_OR_CPU},
};
```

An example for specifying target node is available in `tests/datawizard/specific_node.c`.

Chapter 4

Advanced Scheduling

4.1 Energy-based Scheduling

Note: by default, StarPU does not let CPU workers sleep, to let them react to task release as quickly as possible. For idle time to really let CPU cores save energy, one needs to use the `configure` option --enable-blocking-drivers.
If the application can provide some energy consumption performance model (through the field starpu_codelet::energy_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` (STARPU_SCHED_GAMMA) 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 energy consumption.
This is however not sufficient to correctly optimize energy: the scheduler would simply tend to run all computations on the most energy-conservative processing unit. To account for the consumption of the whole machine (including idle processing units), the idle power of the machine should be given by setting `export STARPU_IDLE_POWER=200` (STARPU_IDLE_POWER) for 200W, for instance. This value can often be obtained from the machine power supplier, e.g. by running

```
ipmitool -I lanplus -H mymachine-ipmi -U myuser -P mypasswd sdr type Current
```

The energy actually consumed by the total execution can be displayed by setting `export STARPU_PROFILING=1 STARPU_WORKER_STATS=1` (STARPU_PROFILING and STARPU_WORKER_STATS).
For OpenCL devices, on-line task consumption measurement is currently supported through the OpenCL extension `CL_PROFILING_POWER_CONSUMED`, implemented in the MoviSim simulator.
For CUDA devices, on-line task consumption measurement is supported on V100 cards and beyond. This however only works for quite long tasks, since the measurement granularity is about 10ms.
Applications can however provide explicit measurements by feeding the energy performance model by hand. Fine-grain measurement is often not feasible with the feedback provided by the hardware, so users 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.
For CUDA devices starting with V100, the starpu_energy_start() and starpu_energy_stop() helpers, described in Measuring energy and power with StarPU below, make it easy.
For older models, one can use `nvidia-smi -q -d POWER` to get the current consumption in Watt. Multiplying this 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(). (examplified in PerformanceModelExample with the performance model `energy_model`).
Another way to provide the energy performance is to define a perfmodel with starpu_perfmodel::type STARPU_PER_ARCH or STARPU_PER_WORKER , and set the field starpu_perfmodel::arch_cost_function or starpu_perfmodel::worker_cost_function to a function which shall return the estimated consumption of the task in Joules. Such a function can for instance use starpu_task_expected_length() on the task (in µs), multiplied by the typical power consumption of the device, e.g. in W, and divided by 1000000. to get Joules. An example is in the file `tests/perfmodels/regression_based_energy.c`.
There are other functions in StarPU that are used to measure the energy consumed by the system during execution. The starpu_energy_use() function declares that there are the energy consumptions of the task, while the starpu_energy_used() function returns the total energy consumed since the start of measurement.

4.1.1 Measuring energy and power with StarPU

We have extended the performance model of StarPU to measure energy and power values of CPUs. These values are measured using the existing Performance API (PAPI) analysis library. PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

- To measure energy consumption of CPUs, we use the `RAPL` events, which are available on CPU architecture: `RAPL_ENERGY_PKG` that represents the whole CPU socket power consumption, and `RAPL_ENERGY_DRAM` that represents the RAM power consumption.

PAPI provides a generic, portable interface for the hardware performance counters available on all modern CPUs and some other components of interest that are scattered across the chip and system.
In order to use the right `rapl events` for energy measurement, user should check the `rapl events` available on the machine, using this command:

```
$ papi_native_avail
```

Depending on the system configuration, users may have to run this as **root** to get the performance counter values. Since the measurement is for all the CPUs and the memory, the approach taken here is to run a series of tasks on all of them and to take the overall measurement.

- The example below illustrates the energy and power measurements, using the functions starpu_energy_start() and starpu_energy_stop().

In this example, we launch several tasks of the same type in parallel. To perform the energy requirement measurement of a program, we call starpu_energy_start(), which initializes energy measurement counters and starpu_energy_stop(struct starpu_perfmodel *model, struct starpu_task *task, unsigned nimpl, unsigned ntasks, int workerid, enum to stop counting and update the performance model. This ends up yielding the average energy requirement of a single task. The example below illustrates this for a given task type.

```
  unsigned N = starpu_cpu_worker_get_count() * 40;
  starpu_energy_start(-1, STARPU_CPU_WORKER);
  for (i = 0; i < N; i++)
starpu_task_insert(&cl, STARPU_EXECUTE_WHERE, STARPU_CPU, STARPU_R, arg1, STARPU_RW, arg2, 0);
  starpu_task_t *specimen = starpu_task_build(&cl, STARPU_R, arg1, STARPU_RW, arg2, 0);
  starpu_energy_stop(&codelet.energy_model, specimen, 0, N, -1, STARPU_CPU_WORKER);
 .  .  .
```

The example starts 40 times more tasks of the same type than there are CPU execution units. Once the tasks are distributed over all CPUs, the latter are all executing the same type of tasks (with the same data size and parameters); each CPU will in the end execute 40 tasks. A specimen task is then constructed and passed to starpu_energy_stop(), which will fold into the performance model the energy requirement measurement for that type and size of task.
For the energy and power measurements, depending on the system configuration, users may have to run applications as **root** to use PAPI library.
The function starpu_energy_stop() uses `PAPI_stop()` to stop counting and store the values into the array. We calculate both energy in `Joules` and power consumption in `Watt`. We call the function starpu_perfmodel_update_history() in the performance model to provide explicit measurements.

- In the CUDA case, nvml provides per-GPU energy measurement. We can thus calibrate the performance models per GPU:

```
    unsigned N = 40;
for (i = 0; i < starpu_cuda_worker_get_count(); i++) {
    int workerid = starpu_worker_get_by_type(STARPU_CUDA_WORKER, i);
    starpu_energy_start(workerid, STARPU_CUDA_WORKER);
    for (i = 0; i < N; i++)
      starpu_task_insert(&cl, STARPU_EXECUTE_ON_WORKER, workerid, STARPU_R, arg1, STARPU_RW, arg2, 0);
    starpu_task_t *specimen = starpu_task_build(&cl, STARPU_R, arg1, STARPU_RW, arg2, 0);
    starpu_energy_stop(&codelet.energy_model, specimen, 0, N, workerid, STARPU_CUDA_WORKER);
   }
```

- A complete example is available in `tests/perfmodels/regression_based_memset.c`

4.2 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->workerid = starpu_worker_get_by_type(STARPU_CUDA_WORKER, 0);
```

An example is in the file `tests/errorcheck/invalid_tasks.c`.

or equivalently

```
starpu_task_insert(&cl, ..., STARPU_EXECUTE_ON_WORKER, starpu_worker_get_by_type(STARPU_CUDA_WORKER, 0),
      ...);
```

One can also specify a set of worker(s) which are allowed to take the task, as an array of bit, for instance to allow workers 2 and 42:

```
task->workerids = calloc(2,sizeof(uint32_t));
task->workerids[2/32] |= (1 « (2%32));
task->workerids[42/32] |= (1 « (42%32));
task->workerids_len = 2;
```

One can also specify the order in which tasks must be executed by setting the field starpu_task::workerorder. An example is available in the file `tests/main/execute_schedule.c`. If this field is set to a non-zero value, it provides the per-worker consecutive order in which tasks will be executed, starting from 1. For a given of such task, the worker will thus not execute it before all the tasks with smaller order value have been executed, notably in case those tasks are not available yet due to some dependencies. This eventually gives total control of task scheduling, and StarPU will only serve as a "self-timed" task runtime. Of course, the provided order has to be runnable, i.e. a task should not depend on another task bound to the same worker with a bigger order.

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.

4.3 Configuring Heteroprio

Within Heteroprio, one priority per processing unit type is assigned to each task, such that a task has several priorities. Each worker pops the task that has the highest priority for the hardware type it uses, which could be CPU or CUDA for example. Therefore, the priorities has to be used to manage the critical path, but also to promote the consumption of tasks by the more appropriate workers.

The tasks are stored inside buckets, where each bucket corresponds to a priority set. Then each worker uses an indirect access array to know the order in which it should access the buckets. Moreover, all the tasks inside a bucket must be compatible with all the processing units that may access it (at least).

These priorities are now automatically assigned by Heteroprio in auto calibration mode using heuristics. If you want to set these priorities manually, you can change STARPU_HETEROPRIO_USE_AUTO_CALIBRATION and follow the example below.

In this example code, we have 5 types of tasks. CPU workers can compute all of them, but CUDA workers can only execute tasks of types 0 and 1, and are expected to go 20 and 30 time faster than the CPU, respectively.

```
#include <starpu_heteroprio.h>
 // Before calling starpu_init
struct starpu_conf conf;
starpu_conf_init(&conf);
 // Inform StarPU to use Heteroprio
conf.sched_policy_name = "heteroprio";
 // Inform StarPU about the function that will init the priorities in Heteroprio
 // where init_heteroprio is a function to implement
conf.sched_policy_callback = &init_heteroprio;
 // Do other things with conf if needed, then init StarPU
starpu_init(&conf);
void init_heteroprio(unsigned sched_ctx) {
  // CPU uses 5 buckets and visits them in the natural order
  starpu_heteroprio_set_nb_prios(sched_ctx, STARPU_CPU_WORKER, 5);
  // It uses direct mapping idx => idx
  for(unsigned idx = 0; idx < 5; ++idx){
    starpu_heteroprio_set_mapping(sched_ctx, STARPU_CPU_WORKER, idx, idx);
    // If there is no CUDA worker we must tell that CPU is faster
    starpu_heteroprio_set_faster_arch(sched_ctx, STARPU_CPU_WORKER, idx);
  }
  if(starpu_cuda_worker_get_count()){
    // CUDA is enabled and uses 2 buckets
    starpu_heteroprio_set_nb_prios(sched_ctx, STARPU_CUDA_WORKER, 2);
    // CUDA will first look at bucket 1
    starpu_heteroprio_set_mapping(sched_ctx, STARPU_CUDA_WORKER, 0, 1);
```

```
    // CUDA will then look at bucket 2
    starpu_heteroprio_set_mapping(sched_ctx, STARPU_CUDA_WORKER, 1, 2);
    // For bucket 1 CUDA is the fastest
    starpu_heteroprio_set_faster_arch(sched_ctx, STARPU_CUDA_WORKER, 1);
    // And CPU is 30 times slower
    starpu_heteroprio_set_arch_slow_factor(sched_ctx, STARPU_CPU_WORKER, 1, 30.0f);
    // For bucket 0 CUDA is the fastest
    starpu_heteroprio_set_faster_arch(sched_ctx, STARPU_CUDA_WORKER, 0);
    // And CPU is 20 times slower
    starpu_heteroprio_set_arch_slow_factor(sched_ctx, STARPU_CPU_WORKER, 0, 20.0f);
  }
}
```

Then, when a task is inserted, **the priority of the task will be used to select in which bucket is has to be stored**. So, in the given example, the priority of a task will be between 0 and 4 included. However, tasks of priorities 0-1 must provide CPU and CUDA kernels, and tasks of priorities 2-4 must provide CPU kernels (at least). The full source code of this example is available in the file `examples/scheduler/heteroprio_test.c`

4.3.1 Using locality aware Heteroprio

Heteroprio supports a mode where locality is evaluated to guide the distribution of the tasks (see https://peerj.com/articles/cs-190.pdf). Currently, this mode is available using the dedicated function or an environment variable STARPU_HETEROPRIO_USE_LA, and can be configured using environment variables.

```
void starpu_heteroprio_set_use_locality(unsigned sched_ctx_id, unsigned use_locality);
```

In this mode, multiple strategies are available to determine which memory node's workers are the most qualified for executing a specific task. This strategy can be set with STARPU_LAHETEROPRIO_PUSH and available strategies are:

- WORKER: the worker which pushed the task is preferred for the execution.
- LcS: the node with the shortest data transfer time (estimated by StarPU) is the most qualified
- LS_SDH: the node with the smallest data amount to be transferred will be preferred.
- LS_SDH2: similar to LS_SDH, but data in write access is counted in a quadratic manner to give them more importance.
- LS_SDHB: similar to LS_SDH, but data in write access is balanced with a coefficient (its value is set to 1000) and for the same amount of data, the one with fewer pieces of data to be transferred will be preferred.
- LC_SMWB: similar to LS_SDH, but the amount of data in write access gets multiplied by a coefficient which gets closer to 2 as the amount of data in read access gets larger than the data in write access.
- AUTO: strategy by default, this one selects the best strategy and changes it in runtime to improve performance

Other environment variables to configure LaHeteteroprio are documented in ConfiguringLaHeteroprio

4.3.2 Using Heteroprio in auto-calibration mode

In this mode, Heteroprio saves data about each program execution, in order to improve future ones. By default, these files are stored in the folder used by perfmodel, but this can be changed using the STARPU_HETEROPRIO_DATA_DIR environment variable. You can also specify the data filename directly using STARPU_HETEROPRIO_DATA_FILE.

Additionally, to assign priorities to tasks, Heteroprio needs a way to detect that some tasks are similar. By default, Heteroprio looks for tasks with the same perfmodel, or with the same codelet's name if no perfmodel was assigned. This behavior can be changed to only consider the codelet's name by setting STARPU_HETEROPRIO_CODELET_GROUPING_STRATEGY to `1`

Other environment variables to configure AutoHeteteroprio are documented in ConfiguringAutoHeteroprio

Chapter 5

Scheduling Contexts

TODO: improve!

5.1 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 kernels. The main goal is to minimize interferences between the execution of multiple parallel kernels, by partitioning the underlying pool of workers using contexts. Scheduling contexts additionally allow a user to make use of a different scheduling policy depending on the target resource set.

5.2 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 kernels simultaneously, by default these kernels will be executed within this initial context, using a single scheduler policy (see Task↩ SchedulingPolicy). 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 to 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), the amount of workers (or `-1` to designate all workers on the platform) and a list of optional parameters such as the scheduling policy, terminated by a `0`. The scheduling policy 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. A basic example is available in the file `examples/sched_ctx/sched_ctx.c`.

```
/* the list of resources the context will manage */
int workerids[3] = {1, 3, 10};
/* indicate the list of workers assigned to it, the number of workers,
the name of the context and the scheduling policy to be used within
the context */
int id_ctx = starpu_sched_ctx_create(workerids, 3, "my_ctx", STARPU_SCHED_CTX_POLICY_NAME, "dmda", 0);
/* let StarPU know that the following tasks will be submitted to this context */
starpu_sched_ctx_set_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.

5.2.1 Creating A Context With The Default Behavior

If **no scheduling policy** is specified when creating the context, it will be used as **another type of resource**: a parallel worker. A parallel worker is a context without scheduler (eventually delegated to another runtime). For more information, see Creating Parallel Workers On A Machine. It is therefore **mandatory** to stipulate a scheduler to use the contexts in this traditional way.
To create a **context** with the default scheduler, that is either controlled through the environment variable `STARPU_SCHED` or the StarPU default scheduler, one can explicitly use the option `STARPU_SCHED_CTX_POLICY_NAME, ""` as in the following example:

```
/* the list of resources the context will manage */
int workerids[3] = {1, 3, 10};
/* indicate the list of workers assigned to it, the number of workers,
and use the default scheduling policy.  */
int id_ctx = starpu_sched_ctx_create(workerids, 3, "my_ctx", STARPU_SCHED_CTX_POLICY_NAME, "", 0);
/* ....  */
```

A full example is available in the file `examples/sched_ctx/two_cpu_contexts.c`.

5.3 Creating A Context To Partition a GPU

The contexts can also be used to group a set of SMs of an NVIDIA GPU in order to isolate the parallel kernels and allow them to coexecution on a specified partition of the GPU.
Each context will be mapped to a stream and users can indicate the number of SMs. The context can be added to a larger context already grouping CPU cores. This larger context can use a scheduling policy that assigns tasks to both CPUs and contexts (partitions of the GPU) based on performance models adjusted to the number of SMs.
The GPU implementation of the task has to be modified accordingly and receive as a parameter the number of SMs.

```
/* get the available streams (suppose we have nstreams = 2 by specifying them with STARPU_NWORKER_PER_CUDA=2
      */
int nstreams = starpu_worker_get_stream_workerids(gpu_devid, stream_workerids, STARPU_CUDA_WORKER);
int sched_ctx[nstreams];
sched_ctx[0] = starpu_sched_ctx_create(&stream_workerids[0], 1, "subctx",  STARPU_SCHED_CTX_CUDA_NSMS, 6,
      0);
sched_ctx[1] = starpu_sched_ctx_create(&stream_workerids[1], 1, "subctx",  STARPU_SCHED_CTX_CUDA_NSMS, 7,
      0);
int ncpus = 4;
int workers[ncpus+nstreams];
workers[ncpus+0] = stream_workerids[0];
workers[ncpus+1] = stream_workerids[1];
big_sched_ctx = starpu_sched_ctx_create(workers, ncpus+nstreams, "ctx1", STARPU_SCHED_CTX_SUB_CTXS,
      sched_ctxs, nstreams, STARPU_SCHED_CTX_POLICY_NAME, "dmdas", 0);
starpu_task_submit_to_ctx(task, big_sched_ctx);
```

A full example is available in the file `examples/sched_ctx/gpu_partition.c`.

5.4 Modifying A Context

A scheduling context can be modified dynamically. The application may change its requirements during the execution, and the programmer can add additional workers to a context or remove those 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 resources 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);
```

An example is available in the file `examples/sched_ctx/sched_ctx_remove.c`.

5.5 Submitting Tasks To A Context

The application may submit tasks to several contexts, either simultaneously or sequentially. 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 corresponding context. An example is available in the file `examples/sched_ctx/gpu_partition.c`.
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_task_insert(). An example is available in the file `examples/sched_ctx/sched_ctx.c`.

5.6 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);
```

A full example is available in the file `examples/sched_ctx/sched_ctx.c`.

5.7 Emptying A Context

A context may have no resources at the beginning or at a certain moment of the execution. Tasks can still be submitted to these contexts, they will be executed as soon as the contexts will have resources. A list of tasks pending to be executed is kept and will be submitted when workers are added to the contexts.

```
/* create a empty context */
unsigned sched_ctx_id = starpu_sched_ctx_create(NULL, 0, "ctx", 0);
/* submit a task to this context */
starpu_sched_ctx_set_context(&sched_ctx_id);
ret = starpu_task_insert(&codelet, 0);
STARPU_CHECK_RETURN_VALUE(ret, "starpu_task_insert");
/* add CPU workers to the context */
int procs[STARPU_NMAXWORKERS];
int nprocs = starpu_cpu_worker_get_count();
starpu_worker_get_ids_by_type(STARPU_CPU_WORKER, procs, nprocs);
starpu_sched_ctx_add_workers(procs, nprocs, sched_ctx_id);
/* and wait for the task termination */
starpu_task_wait_for_all();
```

The full example is available in the file `examples/sched_ctx/sched_ctx_empty.c`.

However, if resources are never allocated to the context, the application will not terminate. If these tasks have low priority, the application can inform StarPU to not submit them by calling the function starpu_sched_ctx_stop_task_submission().

Chapter 6

Scheduling Context Hypervisor

6.1 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 users know exactly how many workers each scheduling context needs, they 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 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.
Several examples of hypervisor are provided in `sc_hypervisor/examples/*.c`

6.2 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. The basic example is available in the file `sc_hypervisor/examples/sched_ctx_utils/sched_ctx_utils.c`.

6.3 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 too slow.

6.4 Trigger the Hypervisor

The resizing is triggered either when the application requires it (sc_hypervisor_resize_ctxs()) or when the initial distribution of resources alters the performance of the application (the application is too slow or the resource are idle for too long time). An example is available in the file `sc_hypervisor/examples/hierarchical_ctxs/resize_hierarchical_ctxs.c`.
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.

6.5 Resizing Strategies

The plugin proposes several strategies for resizing the scheduling context.
The **Application driven** strategy uses users's input concerning the moment when they want to resize the contexts. Thus, users tag the task that should trigger the resizing process. One can set directly the field starpu_task::hypervisor_tag or use the macro STARPU_HYPERVISOR_TAG in the function starpu_task_insert().

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

or

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

Then users have to indicate that when a task with the specified tag is executed, the contexts should resize.

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

Users can use the same tag to change the resizing configuration of the contexts if they consider 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 Scheduling Context Hypervisor - Regular usage)

```
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/s 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, users have 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_task_insert() 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_task_insert(&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/s rate** strategy, the programmers have 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_flops() is called in order to add or to 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 allocate 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 determined 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 users, 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).

6.6 Defining 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 is 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"
};
```

Examples are provided in `sc_hypervisor/src/hypervisor_policies/*_policy.c`

Chapter 7

How To Define a New Scheduling Policy

7.1 Introduction

StarPU provides two ways of defining a scheduling policy, a basic monolithic way, and a modular way.
The basic monolithic way is directly connected with the core of StarPU, which means that the policy then has to handle all performance details, such as data prefetching, task performance model calibration, worker locking, etc. `examples/scheduler/dummy_sched.c` is a trivial example which does not handle this, and thus e.g. does not achieve any data prefetching or smart scheduling.
The modular way allows implementing just one component, and reuse existing components to cope with all these details. `examples/scheduler/dummy_modular_sched.c` is a trivial example very similar to `dummy_sched.c`, but implemented as a component, which allows assembling it with other components, and notably get data prefetching support for free, and task performance model calibration is properly performed, which allows to easily extend it into taking task duration into account, etc.

7.2 Helper functions for defining a scheduling policy (Basic or modular)

Make sure to have a look at the Scheduling Policy section, which provides a complete list of the functions available for writing advanced schedulers.
This includes getting an estimation for a task computation completion with starpu_task_expected_length(), for a speedup factor relative to CPU speed with starpu_worker_get_relative_speedup(), for the expected data transfer time in micro-seconds with starpu_task_expected_data_transfer_time(), starpu_task_expected_data_transfer_time_for(), or starpu_data_expected_transfer_time(), for the expected conversion time in micro-seconds with starpu_task_expected_conversion_t for the required energy with starpu_task_expected_energy() or starpu_task_worker_expected_energy(), etc. Per-worker variants are also available with starpu_task_worker_expected_length(), etc. The average over workers is also available with starpu_task_expected_length_average() and starpu_task_expected_energy_average(). Other useful functions include starpu_transfer_bandwidth(), starpu_transfer_latency(), starpu_transfer_predict(), ... The successors of a task can be obtained with starpu_task_get_task_succs(). One can also directly test the presence of a data handle with starpu_data_is_on_node(). Prefetches can be triggered by calling either starpu_prefetch_task_input_for(), starpu_idle_prefetch_task_input_for(), starpu_prefetch_task_input_for_prio(), or starpu_idle_prefetch_task_input_for_prio(). And prefetching data on a specified node can use either starpu_prefetch_task_input_on_node(), starpu_prefetch_task_input_on_node_prio(), starpu_idle_prefetch_task_input_on_node(), or starpu_idle_prefetch_task_input_on_node_prio(). The `_prio` versions allow specifying a priority for the transfer (instead of taking the task priority by default). These prefetches are only processed when there are no fetch data requests (i.e. a task is waiting for it) to process. The `_idle` versions queue the transfers on the idle prefetch queue, which is only processed when there are no non-idle prefetches to process. starpu_get_prefetch_flag() is a convenient helper for checking the value of the STARPU_PREFETCH environment variable. When a scheduler does such prefetching, it should set the `prefetches` field of the `starpu_sched_policy` to 1, to prevent the core from triggering its own prefetching.
For applications that need to prefetch data or to perform other pre-execution setup before a task is executed, it is useful to call the function starpu_task_notify_ready_soon_register() which registers a callback function when a task is about to become ready for execution. starpu_worker_set_going_to_sleep_callback() and starpu_worker_set_waking_up_callback() allow to register an external resource manager callback function that will be notified about workers going to sleep or waking up, when StarPU is compiled with support for blocking drivers

and worker callbacks.
Schedulers should call starpu_task_set_implementation() or starpu_task_get_implementation() to specify or to retrieve the codelet implementation to be executed when executing a specific task.
One can determine if a worker type is capable of executing a specific task by calling the function starpu_worker_type_can_execute_task(). The function starpu_sched_find_all_worker_combinations() must be used to identify all viable worker combinations that can execute a parallel task. starpu_combined_worker_get_count() and starpu_worker_is_combined_worker() can be used to determine the number of different combined workers and whether a particular worker is a combined worker respectively. starpu_combined_worker_get_id() allows to get the identifier of the current combined worker. starpu_combined_worker_assign_workerid() allow users to or register a new combined worker and get its identifier, it then needs to be given to a worker collection with the starpu_worker_collection::add. starpu_combined_worker_get_desceiption() returns the description of a combined worker. Additionally, the function starpu_worker_is_blocked_in_parallel() is utilized to determine if a worker is currently blocked in a parallel task, whereas starpu_worker_is_slave_somewhere() can be called to determine if a worker is presently functioning as a slave for another worker. StarPU also provides two functions for initializing and preparing the execution of parallel tasks: starpu_parallel_task_barrier_init() and starpu_parallel_task_barrier_init_n().
Usual functions can be used on tasks, for instance one can use the following to get the data size for a task.

```
size = 0;
write = 0;
if (task->cl)
    for (i = 0; i < STARPU_TASK_GET_NBUFFERS(task); i++)
    {
        starpu_data_handle_t data = STARPU_TASK_GET_HANDLE(task, i)
        size_t datasize = starpu_data_get_size(data);
        size += datasize;
        if (STARPU_TASK_GET_MODE(task, i) & STARPU_W)
            write += datasize;
    }
```

Task queues can be implemented with the starpu_task_list functions. The function starpu_task_list_init() is used to initialize an empty list structure. Once the list is initialized, new tasks can be added to it using the starpu_task_list_push_front() and starpu_task_list_push_back() to add a task to the front or back of the list respectively. starpu_task_list_front() and starpu_task_list_back() can be used to get the first or last task in the list without removing it. starpu_task_list_begin() and starpu_task_list_end() can be used to get the task iterators from the beginning of the list and check whether it is the end of the list respectively. starpu_task_list_next() can be used to get the next task in the list, which is not erase-safe. starpu_task_list_empty() can be used to check whether the list is empty. To remove tasks from the queue, the function starpu_task_list_erase() is used to remove a specific task from the list. starpu_task_list_pop_front() and starpu_task_list_pop_back() can be used to remove the first or last task from the list. Finally, the function starpu_task_list_ismember() is used to check whether a given task is contained in the list. The function starpu_task_list_move() is used to move list from one head to another.
Access to the `hwloc` topology is available with starpu_worker_get_hwloc_obj().

7.3 Defining A New Basic Scheduling Policy

A full example showing how to define a new scheduling policy is available in the StarPU sources in `examples/scheduler/dummy_sched.c`.
The scheduler has to provide methods:

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

The idea is that when a task becomes ready for execution, the starpu_sched_policy::push_task method is called to give the ready task to the scheduler. Then call starpu_push_task_end() to notify that the specified task has been pushed. When a worker is idle, the starpu_sched_policy::pop_task method is called to get a task from the scheduler. It is up to the scheduler to implement what is between. A simple eager scheduler is for instance to make starpu_sched_policy::push_task push the task to a global list, and make starpu_sched_policy::pop_task pop from this list. A scheduler can also use starpu_push_local_task() to directly push tasks to a per-worker queue, and then StarPU does not even need to implement starpu_sched_policy::pop_task. If there are no ready tasks within the scheduler, it can just return `NULL`, and the worker will sleep.

starpu_sched_policy::add_workers and starpu_sched_policy::remove_workers are used to add or remove workers to or from a scheduling policy, so that the number of workers in a policy can be dynamically adjusted. After adding or removing workers from a scheduling policy, the worker task lists should be updated to ensure that the workers are assigned tasks appropriately. By calling starpu_sched_ctx_worker_shares_tasks_lists(), you can specify whether a worker may pop tasks from the task list of other workers or if there is a central list with tasks for all the workers.
The starpu_sched_policy section provides the exact rules that govern the methods of the policy.
One can enumerate the workers with this iterator:

```
struct starpu_worker_collection *workers = starpu_sched_ctx_get_worker_collection(sched_ctx_id);
struct starpu_sched_ctx_iterator it;
workers->init_iterator(workers, &it);
while(workers->has_next(workers, &it))
{
        unsigned worker = workers->get_next(workers, &it);
        ...
}
```

To provide synchronization between workers, a per-worker lock exists to protect the data structures of a given worker. It is acquired around scheduler methods, so that the scheduler does not need any additional mutex to protect its per-worker data.
In case the scheduler wants to access another scheduler's data, it should use starpu_worker_lock() and starpu_worker_unlock(), or use starpu_worker_trylock() which will not block if the lock is not immediately available, or use starpu_worker_lock_self() and starpu_worker_unlock_self() to acquire and to release a lock on the worker associated with the current thread.
Calling

```
starpu_worker_lock(B)
```

from a worker `A` will however thus make worker `A` wait for worker `B` to complete its scheduling method. That may be a problem if that method takes a long time, because it is e.g. computing a heuristic or waiting for another mutex, or even cause deadlocks if worker `B` is calling

```
starpu_worker_lock(A)
```

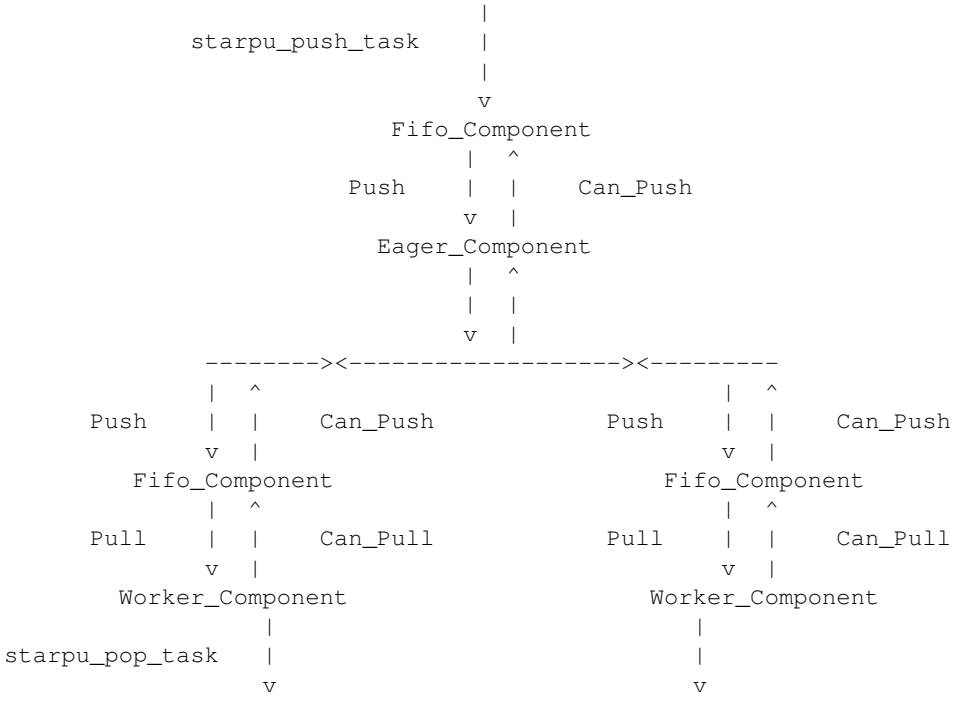
at the same time. In such a case, worker `B` must call starpu_worker_relax_on() and starpu_worker_relax_off() around the section which potentially blocks (and does not actually need protection). While a worker is in relaxed mode, e.g. between a pair of starpu_worker_relax_on() and starpu_worker_relax_off() calls, its state can be altered by other threads: for instance, worker `A` can push tasks for worker `B`. In consequence, worker `B` must re-assess its state after

```
starpu_worker_relax_off(B)
```

, such as taking possible new tasks pushed to its queue into account. Calling starpu_worker_get_relax_state() to query the relaxation state of a worker.
When the starpu_sched_policy::push_task method has pushed a task for another worker, one has to call starpu_wake_worker_relax(), starpu_wake_worker_relax_light(), starpu_wake_worker_no_relax() or starpu_wake_worker_locked() so that the worker wakes up and picks it. If the task was pushed on a shared queue, one may want to only wake one idle worker. An example doing this is available in `src/sched_policies/eager_central_policy.c`. When the scheduling policy makes a scheduling decision for a task, it shouhld call starpu_sched_task_break().
Schedulers can set the minimum or maximum task priority level supported by the scheduling policy by calling starpu_sched_set_min_priority() or starpu_sched_set_max_priority(), and then applications can call starpu_sched_get_min_priority() or starpu_sched_get_max_priority() to retrieve the minimum or maximum priority value. The file `src/sched_policies/heteroprio.c` shows how to uses these functions.
When scheduling a task, it is important to check whether the specified worker can execute the codelet before assigning the task to that worker. This is done using the starpu_worker_can_execute_task() function, or starpu_combined_worker_can_execute_task() which is compatible with combined workers, or starpu_worker_can_execute_task_impl() which also returns the list of implementation numbers that can be used by the worker to execute the task, or starpu_worker_can_execute_task_first_impl() which also returns the first implementation number that can be used.
A pointer to one data structure specific to the scheduler can be set with starpu_sched_ctx_set_policy_data() and fetched with starpu_sched_ctx_get_policy_data(). Per-worker data structures can then be stored in it by allocating a STARPU_NMAXWORKERS -sized array of structures indexed by workers.
A variety of examples of advanced schedulers can be read in `src/sched_policies`, for instance `random_policy.c`, `eager_central_policy.c`, `work_stealing_policy.c` Code protected by `if (starpu_get_nsched_ctxs() > 1)` can be ignored, this is for scheduling contexts, which is an experimental feature.

7.4 Defining A New Modular Scheduling Policy

StarPU's Modularized Schedulers are made of individual Scheduling Components Modularizedly assembled as a Scheduling Tree. Each Scheduling Component has a unique purpose, such as prioritizing tasks or mapping tasks over resources. A typical Scheduling Tree is shown below.



When a task is pushed by StarPU in a Modularized Scheduler, the task moves from a Scheduling Component to another, following the hierarchy of the Scheduling Tree, and is stored in one of the Scheduling Components of the strategy. When a worker wants to pop a task from the Modularized Scheduler, the corresponding Worker Component of the Scheduling Tree tries to pull a task from its parents, following the hierarchy, and gives it to the worker if it succeeded to get one.

7.4.1 Interface

Each Scheduling Component must follow the following pre-defined Interface to be able to interact with other Scheduling Components.

- push_task (child_component, Task)
  The calling Scheduling Component transfers a task to its Child Component. When the Push function returns, the task no longer belongs to the calling Component. The Modularized Schedulers' model relies on this function to perform prefetching. See starpu_sched_component::push_task for more details
- pull_task (parent_component, caller_component) -> Task
  The calling Scheduling Component requests a task from its Parent Component. When the Pull function ends, the returned task belongs to the calling Component. See starpu_sched_component::pull_task for more details
- can_push (caller_component, parent_component)
  The calling Scheduling Component notifies its Parent Component that it is ready to accept new tasks. See starpu_sched_component::can_push for more details
- can_pull (caller_component, child_component)
  The calling Scheduling Component notifies its Child Component that it is ready to give new tasks. See starpu_sched_component::can_pull for more details

The components also provide the following useful methods:

- starpu_sched_component::estimated_load provides an estimated load of the component

- starpu_sched_component::estimated_end provides an estimated date of availability of workers behind the component, after processing tasks in the component and below. This is computed only if the estimated field of the tasks have been set before passing it to the component.

7.4.2 Building a Modularized Scheduler

7.4.2.1 Pre-implemented Components

StarPU is currently shipped with the following four Scheduling Components :

- Storage Components : Fifo, Prio
  Components which store tasks. They can also prioritize them if they have a defined priority. It is possible to define a threshold for those Components following two criteria : the number of tasks stored in the Component, or the sum of the expected length of all tasks stored in the Component. When a push operation tries to queue a task beyond the threshold, the push fails. When some task leaves the queue (and thus possibly more tasks can fit), this component calls can_push from ancestors.
- Resource-Mapping Components : Mct, Heft, Eager, Random, Work-Stealing
  "Core" of the Scheduling Strategy, those Components are the ones who make scheduling choices between their children components.
- Worker Components : Worker
  Each Worker Component modelizes a concrete worker, and copes with the technical tricks of interacting with the StarPU core. Modular schedulers thus usually have them at the bottom of their component tree.
- Special-Purpose Components : Perfmodel_Select, Best_Implementation
  Components dedicated to original purposes. The Perfmodel_Select Component decides which Resource-↩ Mapping Component should be used to schedule a task: a component that assumes tasks with a calibrated performance model; a component for non-yet-calibrated tasks, that will distribute them to get measurements done as quickly as possible; and a component that takes the tasks without performance models.
  The Best_Implementation Component chooses which implementation of a task should be used on the chosen resource.

7.4.2.2 Progression And Validation Rules

Some rules must be followed to ensure the correctness of a Modularized Scheduler :

- At least one Storage Component without threshold is needed in a Modularized Scheduler, to store incoming tasks from StarPU. It can for instance be a global component at the top of the tree, or one component per worker at the bottom of the tree, or intermediate assemblies. The important point is that the starpu_sched_component::push_task call at the top can not fail, so there has to be a storage component without threshold between the top of the tree and the first storage component with threshold, or the workers themselves.
- At least one Resource-Mapping Component is needed in a Modularized Scheduler. Resource-Mapping Components are the only ones which can make scheduling choices, and so the only ones which can have several children.

7.4.2.3 Locking in modularized schedulers

Most often, components do not need to take locks. This allows e.g. the push operation to be called in parallel when tasks get released in parallel from different workers which have completed different ancestor tasks.
When a component has internal information which needs to be kept coherent, the component can define its own lock to take it as it sees fit, e.g. to protect a task queue. This may however limit scalability of the scheduler. Conversely, since push and pull operations will be called concurrently from different workers, the component might prefer to use a central mutex to serialize all scheduling decisions to avoid pathological cases (all push calls decide to put their task on the same target)

7.4.2.4 Implementing a Modularized Scheduler

The following code shows how to implement a Tree-Eager-Prefetching Scheduler.

```
static void initialize_eager_prefetching_center_policy(unsigned sched_ctx_id)
{
  /* The eager component will decide for each task which worker will run it,
* and we want fifos both above and below the component */
  starpu_sched_component_initialize_simple_scheduler(
    starpu_sched_component_eager_create, NULL,
    STARPU_SCHED_SIMPLE_DECIDE_WORKERS |
    STARPU_SCHED_SIMPLE_FIFO_ABOVE |
    STARPU_SCHED_SIMPLE_FIFOS_BELOW,
    sched_ctx_id);
}
/* Initializing the starpu_sched_policy struct associated to the Modularized
* Scheduler :  only the init_sched and deinit_sched needs to be defined to
* implement a Modularized Scheduler */
struct starpu_sched_policy _starpu_sched_tree_eager_prefetching_policy =
{
  .init_sched = initialize_eager_prefetching_center_policy,
  .deinit_sched = starpu_sched_tree_deinitialize,
  .add_workers = starpu_sched_tree_add_workers,
  .remove_workers = starpu_sched_tree_remove_workers,
  .push_task = starpu_sched_tree_push_task,
  .pop_task = starpu_sched_tree_pop_task,
  .pre_exec_hook = starpu_sched_component_worker_pre_exec_hook,
  .post_exec_hook = starpu_sched_component_worker_post_exec_hook,
  .policy_name = "tree-eager-prefetching",
  .policy_description = "eager with prefetching tree policy"
};
```

starpu_sched_component_initialize_simple_scheduler() is a helper function which makes it very trivial to assemble a modular scheduler around a scheduling decision component as seen above (here, a dumb eager decision component). Most often, a modular scheduler can be implemented that way.

A modular scheduler can also be constructed hierarchically with starpu_sched_component_composed_recipe_create(). To retrieve the current scheduling tree of a task, starpu_sched_tree_get() can be called.

That modular scheduler can also be built by hand in the following way:

```
#define _STARPU_SCHED_NTASKS_THRESHOLD_DEFAULT 2
#define _STARPU_SCHED_EXP_LEN_THRESHOLD_DEFAULT 1000000000.0
static void initialize_eager_prefetching_center_policy(unsigned sched_ctx_id)
{
  unsigned ntasks_threshold = _STARPU_SCHED_NTASKS_THRESHOLD_DEFAULT;
  double exp_len_threshold = _STARPU_SCHED_EXP_LEN_THRESHOLD_DEFAULT;
  [...]
  starpu_sched_ctx_create_worker_collection
    (sched_ctx_id, STARPU_WORKER_LIST);
  /* Create the Scheduling Tree */
  struct starpu_sched_tree * t = starpu_sched_tree_create(sched_ctx_id);
  /* The Root Component is a Flow-control Fifo Component */
   t->root = starpu_sched_component_fifo_create(NULL);
  /* The Resource-mapping Component of the strategy is an Eager Component
*/
  struct starpu_sched_component *eager_component = starpu_sched_component_eager_create(NULL);
  /* Create links between Components :  the Eager Component is the child
* of the Root Component */
  starpu_sched_component_connect(t->root, eager_component);
  /* A task threshold is set for the Flow-control Components which will
* be connected to Worker Components.  By doing so, this Modularized
* Scheduler will be able to perform some prefetching on the resources
*/
  struct starpu_sched_component_fifo_data fifo_data =
  {
    .ntasks_threshold = ntasks_threshold,
    .exp_len_threshold = exp_len_threshold,
  };
  unsigned i;
  for(i = 0; i < starpu_worker_get_count() + starpu_combined_worker_get_count(); i++)
  {
    /* Each Worker Component has a Flow-control Fifo Component as
* father */
    struct starpu_sched_component * worker_component = starpu_sched_component_worker_new(i);
    struct starpu_sched_component * fifo_component = starpu_sched_component_fifo_create(&fifo_data);
    starpu_sched_component_connect(fifo_component, worker_component);
    /* Each Flow-control Fifo Component associated to a Worker
* Component is linked to the Eager Component as one of its
* children */
    starpu_sched_component_connect(eager_component, fifo_component);
  }
  starpu_sched_tree_update_workers(t);
  starpu_sched_ctx_set_policy_data(sched_ctx_id, (void*)t);
}
/* Properly destroy the Scheduling Tree and all its Components */
static void deinitialize_eager_prefetching_center_policy(unsigned sched_ctx_id)
{
  struct starpu_sched_tree * tree = (struct
```

```
      starpu_sched_tree*)starpu_sched_ctx_get_policy_data(sched_ctx_id);
  starpu_sched_tree_destroy(tree);
  starpu_sched_ctx_delete_worker_collection(sched_ctx_id);
}
/* Initializing the starpu_sched_policy struct associated to the Modularized
* Scheduler :  only the init_sched and deinit_sched needs to be defined to
* implement a Modularized Scheduler */
struct starpu_sched_policy _starpu_sched_tree_eager_prefetching_policy =
{
  .init_sched = initialize_eager_prefetching_center_policy,
  .deinit_sched = deinitialize_eager_prefetching_center_policy,
  .add_workers = starpu_sched_tree_add_workers,
  .remove_workers = starpu_sched_tree_remove_workers,
  .push_task = starpu_sched_tree_push_task,
  .pop_task = starpu_sched_tree_pop_task,
  .pre_exec_hook = starpu_sched_component_worker_pre_exec_hook,
  .post_exec_hook = starpu_sched_component_worker_post_exec_hook,
  .policy_name = "tree-eager-prefetching",
  .policy_description = "eager with prefetching tree policy"
};
```

Instead of calling starpu_sched_tree_update_workers(), one can call starpu_sched_tree_update_workers_in_ctx() to update the set of workers that are available to execute tasks in a given scheduling tree within a specific StarPU context.
Other modular scheduler examples can be seen in `src/sched_policies/modular_*.c`
For instance, `modular-heft-prio` needs performance models, decides memory nodes, uses prioritized fifos above and below, and decides the best implementation.
If unsure on the result of the modular scheduler construction, you can run a simple application with FxT enabled (see GeneratingTracesWithFxT), and open the generated file `trace.html` in a web-browser.

7.4.3 Management of parallel task

At the moment, parallel tasks can be managed in modularized schedulers through combined workers: instead of connecting a scheduling component to a worker component, one can connect it to a combined worker component (i.e. a worker component created with a combined worker id). That component will handle creating task aliases for parallel execution and push them to the different workers components.

7.4.4 Writing a Scheduling Component

7.4.4.1 Generic Scheduling Component

Each Scheduling Component is instantiated from a Generic Scheduling Component, which implements a generic version of the Interface. The generic implementation of Pull, Can_Pull and Can_Push functions are recursive calls to their parents (respectively to their children). However, as a Generic Scheduling Component do not know how many children it will have when it will be instantiated, it does not implement the Push function.

7.4.4.2 Instantiation : Redefining the Interface

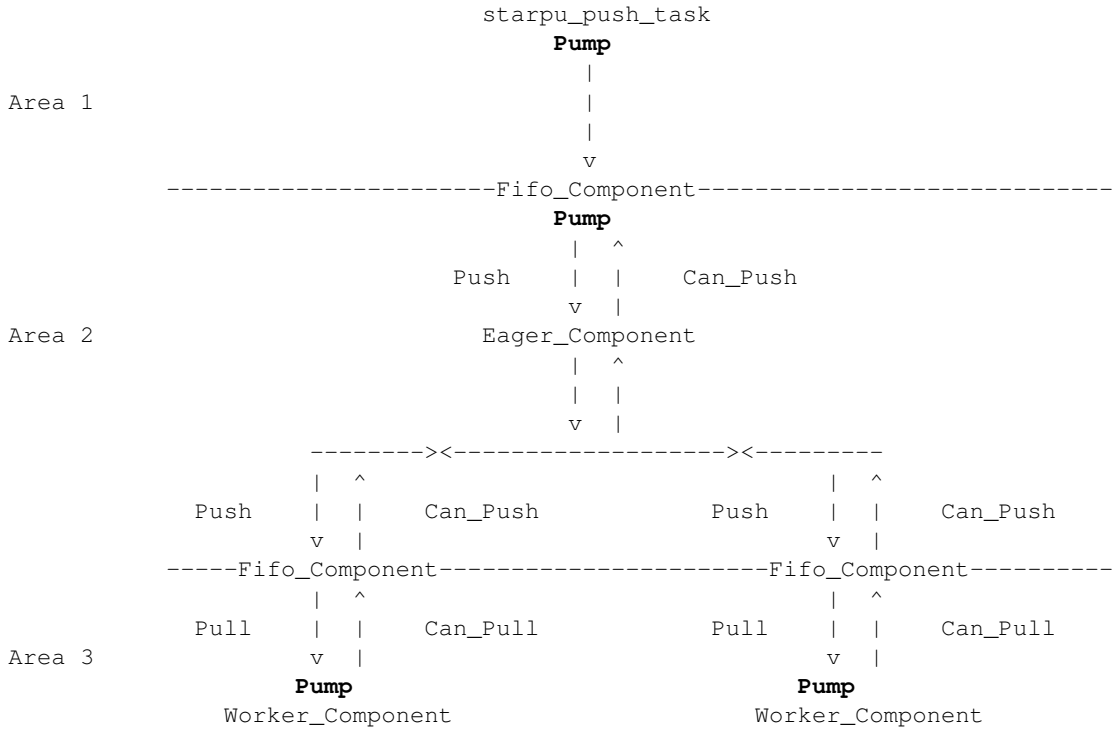
A Scheduling Component must implement all the functions of the Interface. It is so necessary to implement a Push function to instantiate a Scheduling Component. The implemented Push function is the "fingerprint" of a Scheduling Component. Depending on how functionalities or properties programmers want to give to the Scheduling Component they are implementing, it is possible to reimplement all the functions of the Interface. For example, a Flow-control Component reimplements the Pull and the Can_Push functions of the Interface, allowing to catch the generic recursive calls of these functions. The Pull function of a Flow-control Component can, for example, pop a task from the local storage queue of the Component, and give it to the calling Component which asks for it.

7.4.4.3 Detailed Progression and Validation Rules

- A Reservoir is a Scheduling Component which redefines a Push and a Pull function, in order to store tasks into it. A Reservoir delimit Scheduling Areas in the Scheduling Tree.
- A Pump is the engine source of the Scheduler : it pushes/pulls tasks to/from a Scheduling Component to another. Native Pumps of a Scheduling Tree are located at the root of the Tree (incoming Push calls from StarPU), and at the leafs of the Tree (Pop calls coming from StarPU Workers). Pre-implemented Scheduling Components currently shipped with Pumps are Flow-Control Components and the Resource-Mapping Component Heft, within their defined Can_Push functions.

- A correct Scheduling Tree requires a Pump per Scheduling Area and per Execution Flow.

The Tree-Eager-Prefetching Scheduler shown in Section Implementing a Modularized Scheduler follows the previous assumptions :



7.5 Using a New Scheduling Policy

There are two ways to use a new scheduling policy.

- If the code is directly available from your application, you can set the field starpu_conf::sched_policy with a pointer to your new defined scheduling policy.

```
starpu_conf_init(&conf);
conf.sched_policy = &dummy_sched_policy,
ret = starpu_init(&conf);
```

- You can also load the new policy dynamically using the environment variable STARPU_SCHED_LIB. An example is given in `examples/scheduler/libdummy_sched.c` and `examples/scheduler/libdummy_sched.sh`.

  The variable STARPU_SCHED_LIB needs to give the location of a .so file which needs to define a function `struct starpu_sched_policy *starpu_get_sched_lib_policy(const char *name)`

```
struct starpu_sched_policy *get_sched_policy(const char *name)
{
        if (!strcmp(name, "dummy"))
                return &dummy_sched_policy;
        return NULL;
}
```

  To use it, you need to define both variables STARPU_SCHED_LIB and STARPU_SCHED

```
STARPU_SCHED_LIB=libdummy_sched.so STARPU_SCHED=dummy yourapplication
```

  If the library defines a function `struct starpu_sched_policy **starpu_get_sched_lib_policies()`, the policies defined by the library can be displayed using the `help` functionality.

```
STARPU_SCHED_LIB=libdummy_sched.so STARPU_SCHED=help yourapplication
```

7.6 Graph-based Scheduling

For performance reasons, most of the schedulers shipped with StarPU use simple list-scheduling heuristics, assuming that the application has already set priorities. This is why they do their scheduling between when tasks become available for execution and when a worker becomes idle, without looking at the task graph.

Other heuristics can however look at the task graph. Recording the task graph is expensive, so it is not available by default, the scheduling heuristic has to set `_starpu_graph_record` to `1` from the initialization function, to make it available. Then the `_starpu_graph*` functions can be used.
`src/sched_policies/graph_test_policy.c` is an example of simple greedy policy which automatically computes priorities by bottom-up rank.
The idea is that while the application submits tasks, they are only pushed to a bag of tasks. When the application is finished with submitting tasks, it calls starpu_do_schedule() (or starpu_task_wait_for_all(), which calls starpu_do_schedule()), and the starpu_sched_policy::do_schedule method of the scheduler is called. This method calls `_starpu_graph_compute_depths()` to compute the bottom-up ranks, and then uses these ranks to set priorities over tasks.
It then has two priority queues, one for CPUs, and one for GPUs, and uses a dumb heuristic based on the duration of the task over CPUs and GPUs to decide between the two queues. CPU workers can then pop from the CPU priority queue, and GPU workers from the GPU priority queue.

7.7 Debugging Scheduling

All the OnlinePerformanceTools and OfflinePerformanceTools can be used to get information about how well the execution proceeded, and thus the overall quality of the execution.
Precise debugging can also be performed by using the STARPU_TASK_BREAK_ON_PUSH, STARPU_TASK↩
_BREAK_ON_SCHED, STARPU_TASK_BREAK_ON_POP, and STARPU_TASK_BREAK_ON_EXEC environment variables. By setting the job_id of a task in these environment variables, StarPU will raise `SIGTRAP` when the task is being scheduled, pushed, or popped by the scheduler. This means that when one notices that a task is being scheduled in a seemingly odd way, one can just re-execute the application in a debugger, with some of those variables set, and the execution will stop exactly at the scheduling points of this task, thus allowing to inspect the scheduler state, etc.

Chapter 8

CUDA Support

StarPU sets the current CUDA device by calling starpu_cuda_set_device() which takes an integer argument representing the device number, and sets the current device to the specified device number. By setting the current device, applications can select which CUDA device to use for their computations, enabling efficient management of multiple CUDA devices in a system.
We can call starpu_cuda_get_nvmldev() to get identifier of the NVML device associated with a given CUDA device. Three macros STARPU_CUDA_REPORT_ERROR(), STARPU_CUBLAS_REPORT_ERROR(), and STARPU_↩ CUSPARSE_REPORT_ERROR() are useful for debugging and troubleshooting, as they provide detailed information about the error that occur during CUDA or CUBLAS execution.

Chapter 9

OpenCL Support

StarPU provides several functions for managing OpenCL programs and kernels. starpu_opencl_load_program_source() and starpu_opencl_load_program_source_malloc() load the OpenCL program source from a file, but the latter one also allocates buffer for the program source. starpu_opencl_compile_opencl_from_file() and starpu_opencl_compile_opencl_from_string() are used to compile an OpenCL kernel from a source file or a string respectively. starpu_opencl_load_binary_opencl() is used to compile the binary OpenCL kernel. An example is available in `examples/binary/binary.c`.
starpu_opencl_load_opencl_from_file() and starpu_opencl_load_opencl_from_string() are used to compile an OpenCL source code from a file or a string respectively. starpu_opencl_unload_opencl() is used to unload an OpenCL compiled program or kernel from memory. starpu_opencl_load_opencl() is used to create an OpenCL kernel for specified device. starpu_opencl_release_kernel() is used to release the specified OpenCL kernel. An example illustrating the usage of OpenCL support is available in `examples/basic_examples/vector_↩ scal_opencl.c`.
For managing OpenCL contexts, devices, and command queues, there are several functions: starpu_opencl_get_context(), starpu_opencl_get_device() and starpu_opencl_get_queue() are used to retrieve the OpenCL context, device and command queue associated with a given device number respectively. starpu_opencl_get_current_context() and starpu_opencl_get_current_queue() are used to retrieve the OpenCL context or command queue of the current worker that is being used by the calling thread. We can call starpu_opencl_set_kernel_args() to set the arguments for an OpenCL kernel. Examples are available in `examples/filters/custom_mf/`.
Two functions are useful for debugging and error reporting in OpenCL applications. starpu_opencl_error_string() takes an OpenCL error code as an argument and returns a string containing a description of the error. starpu_opencl_display_error() takes an OpenCL error code as an argument and prints the corresponding error message to the standard error stream.

Chapter 10

Maxeler FPGA Support

10.1 Introduction

Maxeler provides hardware and software solutions for accelerating computing applications on dataflow engines (DFEs). DFEs are in-house designed accelerators that encapsulate reconfigurable high-end FPGAs at their core and are equipped with large amounts of DDR memory.
We extend the StarPU task programming library that initially targets heterogeneous architectures to support Field Programmable Gate Array (FPGA).
To create `StarPU/FPGA` applications exploiting DFE configurations, MaxCompiler allows an application to be split into three parts:

- `Kernel`, which implements the computational components of the application in hardware.
- `Manager configuration`, which connects Kernels to the CPU, engine RAM, other Kernels and other DFEs via MaxRing.
- `CPU application`, which interacts with the DFEs to read and write data to the Kernels and engine RAM.

The Simple Live CPU interface (SLiC) is Maxeler's application programming interface for seamless CPU-DFE integration. SLiC allows CPU applications to configure and load a number of DFEs as well as to subsequently schedule and run actions on those DFEs using simple function calls. In StarPU/FPGA applications, we use *Dynamic SLiC Interface* to exchange data streams between the CPU (Main Memory) and DFE (Local Memory).

10.2 Porting Applications to Maxeler FPGA

The way to port an application to FPGA is to set the field starpu_codelet::max_fpga_funcs, to provide StarPU with the function for FPGA implementation, so for instance:

```
struct starpu_codelet cl =
{
    .max_fpga_funcs = {myfunc},
    .nbuffers = 1,
}
```

A basic example is available in the file `tests/maxfpga/max_fpga_basic_static.c`.

10.2.1 StarPU/Maxeler FPGA Application

To give you an idea of the interface that we used to exchange data between `host` (CPU) and `FPGA` (DFE), here is an example, based on one of the examples of Maxeler (https://trac.version.fz-juelich.↩ de/reconfigurable/wiki/Public).
`StreamFMAKernel.maxj` represents the Java kernel code; it implements a very simple kernel (`c=a+b`), and `Test.c` starts it from the `fpga_add` function; it first sets streaming up from the CPU pointers, triggers execution and waits for the result. The API to interact with DFEs is called *SLiC* which then also involves the `MaxelerOS` runtime.

- `StreamFMAKernel.maxj`: the DFE part is described in the MaxJ programming language, which is a Java-based metaprogramming approach.

```
package tests;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
class StreamFMAKernel extends Kernel
{
   private static final DFEType type = dfeInt(32);
   protected StreamFMAKernel(KernelParameters parameters)
   {
            super(parameters);
            DFEVar a = io.input("a", type);
            DFEVar b = io.input("b", type);
            DFEVar c;
            c = a+b;
            io.output("output", c, type);
   }
}
```

- `StreamFMAManager.maxj`: is also described in the MaxJ programming language and orchestrates data movement between the host and the DFE.

```
package tests;
import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.platform.max5.manager.Max5LimaManager;
class StreamFMAManager extends Max5LimaManager
{
        private static final String kernel_name = "StreamFMAKernel";
        public StreamFMAManager(EngineParameters arg0)
        {
                super(arg0);
                KernelBlock kernel = addKernel(new StreamFMAKernel(makeKernelParameters(kernel_name)));
                kernel.getInput("a") <== addStreamFromCPU("a");
                kernel.getInput("b") <== addStreamFromCPU("b");
                addStreamToCPU("output") <== kernel.getOutput("output");
        }
        public static void main(String[] args)
        {
                StreamFMAManager manager = new StreamFMAManager(new EngineParameters(args));
                manager.build();
        }
}
```

Once `StreamFMAKernel.maxj` and `StreamFMAManager.maxj` are written, there are other steps to do:

- Building the JAVA program: (for Kernel and Manager (.maxj))

```
$ maxjc -1.7 -cp $MAXCLASSPATH streamfma/
```

- Running the Java program to generate a DFE implementation (a .max file) that can be called from a StarPU/FPGA application and slic headers (.h) for simulation:

```
$ java -XX:+UseSerialGC -Xmx2048m -cp $MAXCLASSPATH:. streamfma.StreamFMAManager DFEModel=MAIA maxFileName=Str
```

- Build the slic object file (simulation):

```
$ sliccompile StreamFMA.max
```

- `Test.c` :

to interface StarPU task-based runtime system with Maxeler's DFE devices, we use the advanced dynamic interface of *SLiC* in **non_blocking** mode.
Test code must include `MaxSLiCInterface.h` and `MaxFile.h`. The .max file contains the bitstream. The StarPU/FPGA application can be written in C, C++, etc. Some examples are available in the directory `tests/maxfpga`.

```
#include "StreamFMA.h"
#include "MaxSLiCInterface.h"
void fpga_add(void *buffers[], void *cl_arg)
{
    (void)cl_arg;
    int *a = (int*) STARPU_VECTOR_GET_PTR(buffers[0]);
    int *b = (int*) STARPU_VECTOR_GET_PTR(buffers[1]);
    int *c = (int*) STARPU_VECTOR_GET_PTR(buffers[2]);
    int size = STARPU_VECTOR_GET_NX(buffers[0]);
    /* actions to run on an engine */
```

```
    max_actions_t *act = max_actions_init(maxfile, NULL);
    /* set the number of ticks for a kernel */
    max_set_ticks  (act, "StreamFMAKernel", size);
    /* send input streams */
    max_queue_input(act, "a", a, size *sizeof(a[0]));
    max_queue_input(act, "b", b, size*sizeof(b[0]));
    /* store output stream */
    max_queue_output(act,"output", c, size*sizeof(c[0]));
    /* run actions on the engine */
     printf("**** Run actions in non blocking mode **** \n");
    /* run actions in non_blocking mode */
    max_run_t *run0= max_run_nonblock(engine, act);
    printf("*** wait for the actions on DFE to complete *** \n");
    max_wait(run0);
  }
  static struct starpu_codelet cl =
  {
    .cpu_funcs = {cpu_func},
    .cpu_funcs_name = {"cpu_func"},
    .max_fpga_funcs = {fpga_add},
    .nbuffers = 3,
    .modes = {STARPU_R, STARPU_R, STARPU_W}
  };
int main(int argc, char **argv)
{
    ...
    /* Implementation of a maxfile */
    max_file_t *maxfile = StreamFMA_init();
    /* Implementation of an engine */
    max_engine_t *engine = max_load(maxfile, "*");
    starpu_init(NULL);
    ...  Task submission etc.  ...
    starpu_shutdown();
    /* deallocate the set of actions */
    max_actions_free(act);
    /* unload and deallocate an engine obtained by way of max_load */
    max_unload(engine);
    return 0;
}
```

To write the StarPU/FPGA application: first, the programmer must describe the codelet using StarPU's C API. This codelet provides both a CPU implementation and an FPGA one. It also specifies that the task has two inputs and one output through the starpu_codelet::nbuffers and starpu_codelet::modes attributes.
`fpga_add` function is the name of the FPGA implementation and is mainly divided in four steps:

- Init actions to be run on DFE.
- Add data to an input stream for an action.
- Add data storage space for an output stream.
- Run actions on DFE in **non_blocking** mode; a non-blocking call returns immediately, allowing the calling code to do more CPU work in parallel while the actions are run.
- Wait for the actions to complete.

In the `main` function, there are four important steps:

- Implement a maxfile.
- Load a DFE.
- Free actions.
- Unload and deallocate the DFE.

The rest of the application (data registration, task submission, etc.) is as usual with StarPU.
The design load can also be delegated to StarPU by specifying an array of load specifications in `starpu_conf::max_fpga_load`, and use starpu_max_fpga_get_local_engine() to access the loaded max engines.
Complete examples are available in `tests/fpga/*.c`

10.2.2 Data Transfers in StarPU/Maxeler FPGA Applications

The communication between the host and the DFE is done through the *Dynamic advance interface* to exchange data between the main memory and the local memory of the DFE.
For the moment, we use STARPU_MAIN_RAM to send and store data to/from DFE's local memory. However, we aim to use a multiplexer to choose which memory node we will use to read/write data. So, users can tell that the computational kernel will take data from the main memory or DFE's local memory, for example.
In StarPU applications, when starpu_codelet::specific_nodes is set to 1, this specifies the memory nodes where each data should be sent to for task execution.

10.2.3 Maxeler FPGA Configuration

To configure StarPU with Maxeler FPGA accelerators, make sure that the `slic-config` is available from your `PATH` environment variable.

10.2.4 Launching Programs: Simulation

Maxeler provides a simple tutorial to use MaxCompiler (https://trac.version.fz-juelich.↩ de/reconfigurable/wiki/Public). Running the Java program to generate maxfile and slic headers (hardware) on Maxeler's DFE device, takes a VERY long time, approx. 2 hours even for this very small example. That's why we use the simulation.

- To start the simulation on Maxeler's DFE device:

```
$ maxcompilersim -c LIMA -n StreamFMA restart
```

- To run the binary (simulation)

```
$ export LD_LIBRARY_PATH=$MAXELEROSDIR/lib:$LD_LIBRARY_PATH
$ export SLIC_CONF="use_simulation=StreamFMA"
```

- To force tasks to be scheduled on the FPGA, one can disable the use of CPU cores by setting the STARPU↩ _NCPU environment variable to 0.

```
$ STARPU_NCPU=0 ./StreamFMA
```

- To stop the simulation

```
$ maxcompilersim -c LIMA -n StreamFMA stop
```

Chapter 11

Out Of Core

11.1 Introduction

When using StarPU, one may need to store more data than what the main memory (RAM) can store. This part describes the method to add a new memory node on a disk and to use it.
Similarly to what happens with GPUs (it's actually exactly the same code), when available main memory becomes scarce, StarPU will evict unused data to the disk, thus leaving room for new allocations. Whenever some evicted data is needed again for a task, StarPU will automatically fetch it back from the disk.
The principle is that one first registers a disk memory node with a set of functions to manipulate data by calling starpu_disk_register(), and then registers a disk location, seen by StarPU as a `void*`, which can be for instance a Unix path for the `stdio`, `unistd` or `unistd_o_direct` backends, or a leveldb database for the `leveldb` backend, an HDF5 file path for the `HDF5` backend, etc. The `disk` backend opens this place with the plug() method. StarPU can then start using it to allocate room and store data there with the disk write method, without user intervention.
Users can also use starpu_disk_open() to explicitly open an object within the disk, e.g. a file name in the `stdio` or `unistd` cases, or a database key in the `leveldb` case, and then use `starpu_*_register` functions to turn it into a StarPU data handle. StarPU will then use this file as an external source of data, and automatically read and write data as appropriate. In the end use starpu_disk_close() to close an existing object.
In any case, users also need to set STARPU_LIMIT_CPU_MEM to the amount of data that StarPU will be allowed to afford. By default, StarPU will use the machine memory size, but part of it is taken by the kernel, the system, daemons, and the application's own allocated data, whose size can not be predicted. That is why users need to specify what StarPU can afford.
Some Out-of-core tests are worth giving a read, see `tests/disk/*.c`

11.2 Use a new disk memory

To use a disk memory node, you have to register it with this function:

```
int new_dd = starpu_disk_register(&starpu_disk_unistd_ops, (void *) "/tmp/", 1024*1024*200);
```

Here, we use the `unistd` library to realize the read/write operations, i.e. `fread/fwrite`. This structure must have a path where to store files, as well as the maximum size the software can afford to store on the disk.
Don't forget to check if the result is correct!
This can also be achieved by just setting environment variables STARPU_DISK_SWAP, STARPU_DISK_SWAP_↩ BACKEND and STARPU_DISK_SWAP_SIZE :

```
export STARPU_DISK_SWAP=/tmp
export STARPU_DISK_SWAP_BACKEND=unistd
export STARPU_DISK_SWAP_SIZE=200
```

The backend can be set to `stdio` (some caching is done by `libc` and the kernel), `unistd` (only caching in the kernel), `unistd_o_direct` (no caching), `leveldb`, or `hdf5`.
It is important to understand that when the backend is not set to `unistd_o_direct`, some caching will occur at the kernel level (the page cache), which will also consume memory... STARPU_LIMIT_CPU_MEM might need to be set to less than half of the machine memory just to leave room for the kernel's page cache, otherwise the kernel will struggle to get memory. Using `unistd_o_direct` avoids this caching, thus allowing to set STARPU_LIMIT_↩ CPU_MEM to the machine memory size (minus some memory for normal kernel operations, system daemons, and application data).

When the register call is made, StarPU will benchmark the disk. This can take some time.
Warning: the size thus has to be at least STARPU_DISK_SIZE_MIN bytes !
StarPU will then automatically try to evict unused data to this new disk. One can also use the standard StarPU memory node API to prefetch data etc., see the Standard Memory Library and the Data Interfaces.
The disk is unregistered during the execution of starpu_shutdown().

11.3 Data Registration

StarPU will only be able to achieve Out-Of-Core eviction if it controls memory allocation. For instance, if the application does the following:

```
p = malloc(1024*1024*sizeof(float));
fill_with_data(p);
starpu_matrix_data_register(&h, STARPU_MAIN_RAM, (uintptr_t) p, 1024, 1024, 1024, sizeof(float));
```

StarPU will not be able to release the corresponding memory since it's the application which allocated it, and StarPU can not know how, and thus how to release it. One thus have to use the following instead:

```
starpu_matrix_data_register(&h, -1, NULL, 1024, 1024, 1024, sizeof(float));
starpu_task_insert(cl_fill_with_data, STARPU_W, h, 0);
```

Which makes StarPU automatically do the allocation when the task running cl_fill_with_data gets executed. And then if it needs to, it will be able to release it after having pushed the data to the disk. Since no initial buffer is provided to starpu_matrix_data_register(), the handle does not have any initial value right after this call, and thus the very first task using the handle needs to use the STARPU_W mode like above, STARPU_R or STARPU_RW would not make sense.
By default, StarPU will try to push any data handle to the disk. To specify whether a given handle should be pushed to the disk, starpu_data_set_ooc_flag() should be used. To get to know whether a given handle should be pushed to the disk, starpu_data_get_ooc_flag() should be used.

11.4 Using Wont Use

By default, StarPU uses a Least-Recently-Used (LRU) algorithm to determine which data should be evicted to the disk. This algorithm can be hinted by telling which data will not be used in the coming future thanks to starpu_data_wont_use(), for instance:

```
starpu_task_insert(&cl_work, STARPU_RW, h, 0);
starpu_data_wont_use(h);
```

StarPU will mark the data as "inactive" and tend to evict to the disk that data rather than others.

11.5 Examples: disk_copy

```
/* Try to write into disk memory
* Use mechanism to push data from main ram to disk ram
*/
#include <starpu.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
/* size of one vector */
#define NX      (30*1000000/sizeof(double))
#define FPRINTF(ofile, fmt, ...)  do { if (!getenv("STARPU_SSILENT")) {fprintf(ofile, fmt, ## __VA_ARGS__);
      }} while(0)
int main(int argc, char **argv)
{
        double *A, *F;
        /* limit main ram to force to push in disk */
        setenv("STARPU_LIMIT_CPU_MEM", "160", 1);
        /* Initialize StarPU with default configuration */
        int ret = starpu_init(NULL);
        if (ret == -ENODEV) goto enodev;
        /* register a disk */
        int new_dd = starpu_disk_register(&starpu_disk_unistd_ops, (void *) "/tmp/", 1024*1024*200);
        /* can't write on /tmp/ */
        if (new_dd == -ENOENT) goto enoent;
        /* allocate two memory spaces */
        starpu_malloc_flags((void **)&A, NX*sizeof(double), STARPU_MALLOC_COUNT);
        starpu_malloc_flags((void **)&F, NX*sizeof(double), STARPU_MALLOC_COUNT);
        FPRINTF(stderr, "TEST DISK MEMORY \n");
        unsigned int j;
        /* initialization with bad values */
        for(j = 0; j < NX; ++j)
        {
                A[j] = j;
```

```
                F[j] = -j;
        }
        starpu_data_handle_t vector_handleA, vector_handleB, vector_handleC, vector_handleD, vector_handleE,
      vector_handleF;
        /* register vector in starpu */
        starpu_vector_data_register(&vector_handleA, STARPU_MAIN_RAM, (uintptr_t)A, NX, sizeof(double));
        starpu_vector_data_register(&vector_handleB, -1, (uintptr_t) NULL, NX, sizeof(double));
        starpu_vector_data_register(&vector_handleC, -1, (uintptr_t) NULL, NX, sizeof(double));
        starpu_vector_data_register(&vector_handleD, -1, (uintptr_t) NULL, NX, sizeof(double));
        starpu_vector_data_register(&vector_handleE, -1, (uintptr_t) NULL, NX, sizeof(double));
        starpu_vector_data_register(&vector_handleF, STARPU_MAIN_RAM, (uintptr_t)F, NX, sizeof(double));
        /* copy vector A->B, B->C...  */
        starpu_data_cpy(vector_handleB, vector_handleA, 0, NULL, NULL);
        starpu_data_cpy(vector_handleC, vector_handleB, 0, NULL, NULL);
        starpu_data_cpy(vector_handleD, vector_handleC, 0, NULL, NULL);
        starpu_data_cpy(vector_handleE, vector_handleD, 0, NULL, NULL);
        starpu_data_cpy(vector_handleF, vector_handleE, 0, NULL, NULL);
        /* StarPU does not need to manipulate the array anymore so we can stop
* monitoring it */
        /* free them */
        starpu_data_unregister(vector_handleA);
        starpu_data_unregister(vector_handleB);
        starpu_data_unregister(vector_handleC);
        starpu_data_unregister(vector_handleD);
        starpu_data_unregister(vector_handleE);
        starpu_data_unregister(vector_handleF);
        /* check if computation is correct */
        int try = 1;
        for (j = 0; j < NX; ++j)
                if (A[j] != F[j])
                {
                        printf("Fail A %f != F %f \n", A[j], F[j]);
                        try = 0;
                }
        /* free last vectors */
        starpu_free_flags(A, NX*sizeof(double), STARPU_MALLOC_COUNT);
        starpu_free_flags(F, NX*sizeof(double), STARPU_MALLOC_COUNT);
        /* terminate StarPU, no task can be submitted after */
        starpu_shutdown();
        if(try)
                FPRINTF(stderr, "TEST SUCCESS\n");
        else
                FPRINTF(stderr, "TEST FAIL\n");
        return (try ?  EXIT_SUCCESS : EXIT_FAILURE);
enodev:
        return 77;
enoent:
        return 77;
}
```

The full code is provided in the file `tests/disk/disk_copy.c`

11.6 Examples: disk_compute

```
/* Try to write into disk memory
* Use mechanism to push data from main ram to disk ram
*/
#include <starpu.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <math.h>
#define NX (1024)
int main(int argc, char **argv)
{
        /* Initialize StarPU with default configuration */
        int ret = starpu_init(NULL);
        if (ret == -ENODEV) goto enodev;
        /* Initialize path and name */
        char pid_str[16];
        int pid = getpid();
        snprintf(pid_str, sizeof(pid_str), "%d", pid);
        const char *name_file_start = "STARPU_DISK_COMPUTE_DATA_";
        const char *name_file_end = "STARPU_DISK_COMPUTE_DATA_RESULT_";
        char * path_file_start = malloc(strlen(base) + 1 + strlen(name_file_start) + 1);
        strcpy(path_file_start, base);
        strcat(path_file_start, "/");
        strcat(path_file_start, name_file_start);
        char * path_file_end = malloc(strlen(base) + 1 + strlen(name_file_end) + 1);
        strcpy(path_file_end, base);
        strcat(path_file_end, "/");
        strcat(path_file_end, name_file_end);
        /* register a disk */
        int new_dd = starpu_disk_register(&starpu_disk_unistd_ops, (void *) base, 1024*1024*1);
```

```
	/* can't write on /tmp/ */
	if (new_dd == -ENOENT) goto enoent;
	unsigned dd = (unsigned) new_dd;
	printf("TEST DISK MEMORY \n");
	/* Imagine, you want to compute data */
	int *A;
	int *C;
	starpu_malloc_flags((void **)&A, NX*sizeof(int), STARPU_MALLOC_COUNT);
	starpu_malloc_flags((void **)&C, NX*sizeof(int), STARPU_MALLOC_COUNT);
	unsigned int j;
	/* you register them in a vector */
	for(j = 0; j < NX; ++j)
	{
		A[j] = j;
		C[j] = 0;
	}
	/* you create a file to store the vector ON the disk */
	FILE * f = fopen(path_file_start, "wb+");
	if (f == NULL)
		goto enoent2;
	/* store it in the file */
	fwrite(A, sizeof(int), NX, f);
	/* close the file */
	fclose(f);
	/* create a file to store result */
	f = fopen(path_file_end, "wb+");
	if (f == NULL)
		goto enoent2;
	/* replace all data by 0 */
	fwrite(C, sizeof(int), NX, f);
	/* close the file */
	fclose(f);
	/* And now, you want to use your data in StarPU */
	/* Open the file ON the disk */
	void * data = starpu_disk_open(dd, (void *) name_file_start, NX*sizeof(int));
	void * data_result = starpu_disk_open(dd, (void *) name_file_end, NX*sizeof(int));
	starpu_data_handle_t vector_handleA, vector_handleC;
	/* register vector in starpu */
	starpu_vector_data_register(&vector_handleA, dd, (uintptr_t) data, NX, sizeof(int));
	/* and do what you want with it, here we copy it into an other vector */
	starpu_vector_data_register(&vector_handleC, dd, (uintptr_t) data_result, NX, sizeof(int));
	starpu_data_cpy(vector_handleC, vector_handleA, 0, NULL, NULL);
	/* free them */
	starpu_data_unregister(vector_handleA);
	starpu_data_unregister(vector_handleC);
	/* close them in StarPU */
	starpu_disk_close(dd, data, NX*sizeof(int));
	starpu_disk_close(dd, data_result, NX*sizeof(int));
	/* check results */
	f = fopen(path_file_end, "rb+");
	if (f == NULL)
		goto enoent;
	/* take data */
	fread(C, sizeof(int), NX, f);
	/* close the file */
	fclose(f);
	int try = 1;
	for (j = 0; j < NX; ++j)
		if (A[j] != C[j])
		{
			printf("Fail A %d != C %d \n", A[j], C[j]);
			try = 0;
		}
	starpu_free_flags(A, NX*sizeof(int), STARPU_MALLOC_COUNT);
	starpu_free_flags(C, NX*sizeof(int), STARPU_MALLOC_COUNT);
	unlink(path_file_start);
	unlink(path_file_end);
	free(path_file_start);
	free(path_file_end);
	/* terminate StarPU, no task can be submitted after */
	starpu_shutdown();
	if(try)
		printf("TEST SUCCESS\n");
	else
		printf("TEST FAIL\n");
	return (try ?  EXIT_SUCCESS : EXIT_FAILURE);
enodev:
	return 77;
enoent2:
	starpu_free_flags(A, NX*sizeof(int), STARPU_MALLOC_COUNT);
	starpu_free_flags(C, NX*sizeof(int), STARPU_MALLOC_COUNT);
enoent:
	unlink(path_file_start);
	unlink(path_file_end);
	free(path_file_start);
	free(path_file_end);
	starpu_shutdown();
```

```
        return 77;
}
```

The full code is provided in the file `tests/disk/disk_compute.c`

11.7 Performances

Scheduling heuristics for Out-of-core are still relatively experimental. The tricky part is that you usually have to find a compromise between privileging locality (which avoids back and forth with the disk) and privileging the critical path, i.e. taking into account priorities to avoid lack of parallelism at the end of the task graph.
It is notably better to avoid defining different priorities to tasks with low priority, since that will make the scheduler want to schedule them by levels of priority, at the expense of locality.
The scheduling algorithms worth trying are thus `dmdar` and `lws`, which privilege data locality over priorities. There will be work on this area in the coming future.

11.8 Feedback Figures

Beyond pure performance feedback, some figures are interesting to have a look at.
Using `export STARPU_BUS_STATS=1` (STARPU_BUS_STATS and STARPU_BUS_STATS_FILE to define a filename in which to display statistics, by default the standard error stream is used) gives an overview of the data transfers which were needed. The values can also be obtained at runtime by using starpu_bus_get_profiling_info(). An example can be read in `src/profiling/profiling_helpers.c`.

```
#---------------------
Data transfer speed for /tmp/sthibault-disk-DJzhAj (node 1):
0 -> 1: 99 MB/s
1 -> 0: 99 MB/s
0 -> 1: 23858 µs
1 -> 0: 23858 µs

#---------------------
TEST DISK MEMORY

#---------------------
Data transfer stats:
        Disk 0 -> NUMA 0        0.0000 GB       0.0000 MB/s     (transfers : 0 - avg -nan MB)
        NUMA 0 -> Disk 0        0.0625 GB       63.6816 MB/s    (transfers : 2 - avg 32.0000 MB)
Total transfers: 0.0625 GB
#---------------------
```

Using `export STARPU_ENABLE_STATS=1` gives information for each memory node on data miss/hit and allocation miss/hit.

```
#---------------------
MSI cache stats :
memory node NUMA 0
        hit : 32 (66.67 %)
        miss : 16 (33.33 %)
memory node Disk 0
        hit : 0 (0.00 %)
        miss : 0 (0.00 %)
#---------------------

#---------------------
Allocation cache stats:
memory node NUMA 0
        total alloc : 16
        cached alloc: 0 (0.00 %)
memory node Disk 0
        total alloc : 8
        cached alloc: 0 (0.00 %)
#---------------------
```

11.9 Disk functions

There are various ways to operate a disk memory node, described by the structure starpu_disk_ops. For instance, the variable starpu_disk_unistd_ops uses read/write functions.

All structures are in Out Of Core.
Examples are provided in `src/core/disk_ops/disk_*.c`

Chapter 12

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. Users have 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.
Some source codes are available in the directory `mpi/`.

12.1 Building with MPI support

If a `mpicc` compiler is already in your PATH, StarPU will automatically enable MPI support in the build. If `mpicc` is not in PATH, you can specify its location by passing `-with-mpicc=/where/there/is/mpicc` to `./configure`
It can be useful to enable MPI tests during `make check` by passing `-enable-mpi-check` to `./configure`. And similarly to `mpicc`, if `mpiexec` in not in PATH, you can specify its location by passing `-with-mpiexec=/where/there/is/mpiexec` to `./configure`, but this is not needed if it is next to `mpicc`, configure will look there in addition to PATH.
Similarly, Fortran examples use `mpif90`, which can be specified manually with `-with-mpifort` if it can't be found automatically.
If users want to run several MPI processes by machine (e.g. one per NUMA node), STARPU_WORKERS_↩ GETBIND needs to be left to its default value 1 to make StarPU take into account the binding set by the MPI launcher (otherwise each StarPU instance would try to bind on all cores of the machine...)
However, depending on the architecture of your machine, one may end up with StarPU-MPI nodes not having any CPU workers. If a node only gets 1 CPU, it will be bound to the MPI thread, and none will be left to start a CPU worker.
One can check that with the following commands.

```
$ mpirun -np 2 starpu_machine_display --worker CPU --count --notopology
1 CPU worker
1 CPU worker
$ mpirun -np 4 starpu_machine_display --worker CPU --count --notopology
4 CPU workers
4 CPU workers
4 CPU workers
4 CPU workers
$ mpirun --bind-to socket -np 2 starpu_machine_display --worker CPU --count --notopology
4 CPU workers
4 CPU workers
$ STARPU_WORKERS_GETBIND=0 mpirun -np 4 starpu_machine_display --worker CPU --count --notopology
4 CPU workers
4 CPU workers
4 CPU workers
4 CPU workers
$ STARPU_WORKERS_GETBIND=0 mpirun -np 2 starpu_machine_display --worker CPU --count --notopology
4 CPU workers
4 CPU workers
```

or with `hwloc`

```
mpirun --bind-to socket -np 2 hwloc-ls --restrict binding --no-io
mpirun -np 2 hwloc-ls --restrict binding --no-io
```

12.2 Example Used In This Documentation

The example below will be used as the base for this documentation. It initializes a token on node 0, and the token is passed from node to node, incremented by one on each step. The code is not using StarPU yet.

```
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
    {
        MPI_Recv(&token, 1, MPI_INT, (rank+size-1)%size, tag, MPI_COMM_WORLD);
    }
    token++;
    if (loop == last_loop && rank == last_rank)
    {
        fprintf(stdout, "Finished:  token value %d\n", token);
    }
    else
    {
        MPI_Send(&token, 1, MPI_INT, (rank+1)%size, tag+1, MPI_COMM_WORLD);
    }
}
```

12.3 About Not Using The MPI Support

Although StarPU provides MPI support, the application programmer may want to keep his MPI communications as they are for a start, and only delegate task execution to StarPU. This is possible by just using starpu_data_acquire(), for instance:

```
for (loop = 0; loop < nloops; loop++)
{
    int tag = loop*size + rank;
    /* Acquire the data to be able to write to it */
    starpu_data_acquire(token_handle, STARPU_W);
    if (loop == 0 && rank == 0)
    {
        token = 0;
        fprintf(stdout, "Start with token value %d\n", token);
    }
    else
    {
        MPI_Recv(&token, 1, MPI_INT, (rank+size-1)%size, tag, MPI_COMM_WORLD);
    }
    starpu_data_release(token_handle);
    /* Task delegation to StarPU to increment the token.  The execution might
* be performed on a CPU, a GPU, etc.  */
    increment_token();
    /* Acquire the update data to be able to read from it */
    starpu_data_acquire(token_handle, STARPU_R);
    if (loop == last_loop && rank == last_rank)
    {
        fprintf(stdout, "Finished:  token value %d\n", token);
    }
    else
    {
        MPI_Send(&token, 1, MPI_INT, (rank+1)%size, tag+1, MPI_COMM_WORLD);
    }
    starpu_data_release(token_handle);
}
```

In that case, `libstarpumpi` is not needed. One can also use `MPI_Isend()` and `MPI_Irecv()`, by calling starpu_data_release() after `MPI_Wait()` or `MPI_Test()` have notified completion.

It is however better to use `libstarpumpi`, to save the application from having to synchronize with starpu_data_acquire(), and instead just submit all tasks and communications asynchronously, and wait for the overall completion.

12.4 Simple Example

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

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

```
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_mpi_init_conf(&argc, &argv, 1, MPI_COMM_WORLD, NULL);
    starpu_mpi_comm_rank(MPI_COMM_WORLD, &rank);
    starpu_mpi_comm_size(MPI_COMM_WORLD, &size);
    starpu_vector_data_register(&token_handle, STARPU_MAIN_RAM, (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)
        {
            starpu_data_acquire(token_handle, STARPU_W);
            token = 0;
            fprintf(stdout, "Start with token value %d\n", token);
            starpu_data_release(token_handle);
        }
        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();
    if (rank == last_rank)
    {
        fprintf(stderr, "[%d] token = %d == %d * %d ?\n", rank, token, nloops, size);
        STARPU_ASSERT(token == nloops*size);
    }
```

We have here replaced `MPI_Recv()` and `MPI_Send()` with starpu_mpi_irecv_detached() and starpu_mpi_isend_detached(), which just submit the communication to be performed. The implicit sequential consistency dependencies provide synchronization between MPI reception and emission and the corresponding tasks. The only remaining synchronization with starpu_data_acquire() is at the beginning and the end.
The full source code is available in the file `mpi/tests/ring.c`.

12.5 How to Initialize StarPU-MPI

As seen in the previous example, one has to call starpu_mpi_init_conf() to initialize StarPU-MPI. The third parameter of the function indicates if MPI should be initialized by StarPU, or if the application did it itself. If the application initializes MPI itself, it must call `MPI_Init_thread()` with `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`, since StarPU-MPI uses a separate thread to perform the communications. `MPI_THREAD_MULTIPLE` is necessary if the application also performs some MPI communications, or if STARPU_MPI_THREAD_MULTIPLE_SEND is set to non-zero.

12.6 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. An MPI request will only be submitted when the data is available in the main memory of the node submitting the request.
There are 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. As shown in the example `mpi/tests/async_ring.c`. 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 send communications, data is acquired with the mode STARPU_R. When using the `configure` option --enable-mpi-pedantic-isend, the mode STARPU_RW is used to make sure there is no more than 1 concurrent `MPI_Isend()` call accessing a data and StarPU does not read from it from tasks during the communication.
Internally, all communication are divided in 2 communications, a first message is used to exchange an envelope describing the data (i.e. its tag and its size), the data itself is sent in a second message. All MPI communications submitted by StarPU uses a unique tag, which has a default value. This value can be accessed with the function starpu_mpi_get_communication_tag() and changed with the function starpu_mpi_set_communication_tag(). The matching of tags with corresponding requests is done within StarPU-MPI.
For any userland communication, the call of the corresponding function (e.g. starpu_mpi_isend()) 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 ready and the corresponding buffer has already been received by MPI, it will be copied in the memory of the data, otherwise the request is stored in the *early requests list*. Sending requests are stored in the *ready requests list*.
While requests need to be processed, the StarPU-MPI progression thread does the following:

1. it polls the *ready requests list*. For all the ready requests, the appropriate function is called to post the corresponding MPI call. For example, an initial call to starpu_mpi_isend() will result in a call to `MPI_Isend()`. If the request is marked as detached, the request will then be added to the *detached requests list*.
2. it posts an `MPI_Irecv()` to retrieve a data envelope.
3. it polls the *detached requests list*. For all the detached requests, it tests its 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.
4. finally, it checks if a data envelope has been received. If so, if the data envelope matches a request in the *early requests list* (i.e. the request has already been posted by the application), the corresponding MPI call is posted (similarly to the first step above).

   If the data envelope does not match any application request, a temporary handle is created to receive the data, a StarPU-MPI request is created and added into the *ready requests list*, and thus will be processed in the first step of the next loop.

To prevent putting too much pressure on the MPI library, only a limited number of requests are emitted concurrently. This behavior can be tuned with the environment variable STARPU_MPI_NDETACHED_SEND. In the same fashion, the progression thread will poll for termination of existing requests after submitting a defined number of requests. This behavior can be tuned with the environment variable STARPU_MPI_NREADY_PROCESS.
The function starpu_mpi_issend() allows to perform a synchronous-mode, non-blocking send of a data. It can also be specified when using starpu_mpi_task_insert() with the parameter STARPU_SSEND.
MPIPtpCommunication gives the list of all the point to point communications defined in StarPU-MPI.

12.7 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 allocated with

```
starpu_malloc_flags(ptr, size, 0)
```

along with its size, where data to be conveyed to another node should be copied.

```
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 = starpu_malloc_on_node_flags(node, *count, 0);
    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;
}
```

The inverse 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_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));
    starpu_free_on_node_flags(node, (uintptr_t) ptr, count, 0);
    return 0;
}
```

And the starpu_data_interface_ops::peek_data operation does the same, but without freeing the buffer. Of course, one can implement starpu_data_interface_ops::unpack_data as merely calling starpu_data_interface_ops::peek_data and do the free:

```
static int complex_peek_data(starpu_data_handle_t handle, unsigned node, void *ptr, size_t count)
{
    STARPU_ASSERT(starpu_data_test_if_allocated_on_node(handle, node));
    STARPU_ASSERT(count == complex_get_size(handle));
    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,
    .peek_data = complex_peek_data
    .unpack_data = complex_unpack_data
};
```

Instead of defining pack and unpack operations, users may want to attach an MPI type to their user-defined data interface. The function starpu_mpi_interface_datatype_register() allows doing so. This function takes 3 parameters: the interface ID for which the MPI datatype is going to be defined, a function's pointer that will create the MPI datatype, and a function's pointer that will free the MPI datatype. If for some data an MPI datatype can not be built (e.g. complex data structure), the creation function can return -1, StarPU-MPI will then fallback to using pack/unpack.

The functions to create and free the MPI datatype are defined and registered as follows.

```
void starpu_complex_interface_datatype_allocate(starpu_data_handle_t handle, MPI_Datatype *mpi_datatype)
{
    int ret;
    int blocklengths[2];
    MPI_Aint displacements[2];
    MPI_Datatype types[2] = {MPI_DOUBLE, MPI_DOUBLE};
    struct starpu_complex_interface *complex_interface = (struct starpu_complex_interface *)
      starpu_data_get_interface_on_node(handle, STARPU_MAIN_RAM);
    MPI_Get_address(complex_interface, displacements);
    MPI_Get_address(&complex_interface->imaginary, displacements+1);
    displacements[1] -= displacements[0];
    displacements[0] = 0;
    blocklengths[0] = complex_interface->nx;
    blocklengths[1] = complex_interface->nx;
    ret = MPI_Type_create_struct(2, blocklengths, displacements, types, mpi_datatype);
    STARPU_ASSERT_MSG(ret == MPI_SUCCESS, "MPI_Type_contiguous failed");
    ret = MPI_Type_commit(mpi_datatype);
    STARPU_ASSERT_MSG(ret == MPI_SUCCESS, "MPI_Type_commit failed");
}
void starpu_complex_interface_datatype_free(MPI_Datatype *mpi_datatype)
{
    MPI_Type_free(mpi_datatype);
}
static struct starpu_data_interface_ops interface_complex_ops =
{
    ...
};
```

```
interface_complex_ops.interfaceid = starpu_data_interface_get_next_id();
starpu_mpi_interface_datatype_register(interface_complex_ops.interfaceid,
      starpu_complex_interface_datatype_allocate, starpu_complex_interface_datatype_free);
starpu_data_interface handle;
starpu_complex_data_register(&handle, STARPU_MAIN_RAM, real, imaginary, 2);
...
```

An example is provided in the file `mpi/examples/user_datatype/my_interface.c`.
It is also possible to use starpu_mpi_datatype_register() to register the functions through a handle rather than the interface ID, but note that in that case it is important to make sure no communication is going to occur before the function starpu_mpi_datatype_register() is called. This would otherwise produce an undefined result as the data may be received before the function is called, and so the MPI datatype would not be known by the StarPU-MPI communication engine, and the data would be processed with the pack and unpack operations. One would thus need to synchronize all nodes:

```
starpu_data_interface handle;
starpu_complex_data_register(&handle, STARPU_MAIN_RAM, real, imaginary, 2);
starpu_mpi_datatype_register(handle, starpu_complex_interface_datatype_allocate,
      starpu_complex_interface_datatype_free);
starpu_mpi_barrier(MPI_COMM_WORLD);
```

12.8 MPI Insert Task Utility

To save the programmer from having to specify all communications, StarPU provides an "MPI Insert Task Utility". The principle is that the application decides a distribution of the data over the MPI nodes by allocating it and notifying StarPU of this 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 MPIInsertTask.
Here is an stencil example showing how to use starpu_mpi_task_insert(). 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 setting the distribution information and the MPI tag which should be used when communicating the data. It also allows to automatically clear the MPI communication cache when unregistering the data. A basic example is in the file `mpi/tests/insert_task.c`.

```
/* 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], STARPU_MAIN_RAM, (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 this 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_task_insert() 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_task_insert(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();
```

The full source code is available in the file `mpi/examples/stencil/stencil5.c`.

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.

To tune the placement of tasks among MPI nodes, one can use STARPU_EXECUTE_ON_NODE or STARPU_EXECUTE_ON_DATA to specify an explicit node (an example can be found in `mpi/tests/insert_task_node_choice.c`), or the node of a given data (e.g. one of the parameters), or use starpu_mpi_node_selection_register_poli and STARPU_NODE_SELECTION_POLICY to provide a dynamic policy (an example can be found in `mpi/tests/policy_register.c`). The default policy is to execute the task on the node which owns a data that require write access; if the task requires several data handles with write access, the node executing the task is selected in order to minimize the amount of data to transfer between nodes.

A function starpu_mpi_task_build() is also provided with the aim to only construct the task structure. All MPI nodes need to call the function, which posts the required send/recv on the various nodes as needed. Only the node which is to execute the task will then return a valid task structure, others will return `NULL`. This node must submit the task. All nodes then need to call the function starpu_mpi_task_post_build() – with the same list of arguments as starpu_mpi_task_build() – to post all the necessary data communications meant to happen after the task execution.

```
struct starpu_task *task;
task = starpu_mpi_task_build(MPI_COMM_WORLD, &cl,
                             STARPU_RW, data_handles[0],
                             STARPU_R, data_handles[1],
                             0);
if (task) starpu_task_submit(task);
starpu_mpi_task_post_build(MPI_COMM_WORLD, &cl,
                           STARPU_RW, data_handles[0],
                           STARPU_R, data_handles[1],
                           0);
```

A full source code using these functions is available in the file `mpi/tests/insert_task_compute.c`.

It is also possible to create and submit the task outside of StarPU-MPI functions and call the functions starpu_mpi_task_exchange_data_before_execution() and starpu_mpi_task_exchange_data_after_execution() to exchange data as required by the data ownership's nodes.

```
struct starpu_mpi_task_exchange_params params;
struct starpu_data_descr descrs[2];
struct starpu_task *task;
task = starpu_task_create();
task->cl = &mycodelet;
task->handles[0] = data_handles[0];
task->handles[1] = data_handles[1];
starpu_mpi_task_exchange_data_before_execution(MPI_COMM_WORLD, task, descrs, &params);
if (params.do_execute) starpu_task_submit(task);
starpu_mpi_task_exchange_data_after_execution(MPI_COMM_WORLD, descrs, 2, params);
```

A full source code using these functions is available in the file `mpi/tests/mpi_task_submit.c`.

If many data handles must be registered with unique tag ids, or if multiple applications are concurrently submitting tasks to StarPU, it is then difficult to keep the uniqueness of the tags for each piece of data. StarPU provides a tag management system to allocate/free a unique range of tags when registering the data to prevent conflict from one application to another. The previous code then becomes:

```
unsigned matrix[X][Y];
starpu_data_handle_t data_handles[X][Y];
int64_t mintag = starpu_mpi_tags_allocate(X*Y);
for(x = 0; x < X; x++)
{
    for (y = 0; y < Y; y++)
    {
            ...
        if (data_handles[x][y])
        {
            starpu_mpi_data_register(data_handles[x][y], mintag + y*Y+x, mpi_rank);
        }
    }
}
```

Then, when all these pieces of data have been unregistered, you may free the range of tags by calling:

```
starpu_mpi_tags_free(mintag);
```

where `mintag` was the value returned by starpu_mpi_tags_allocate().

Note that both these functions should be called by all nodes involved in the computations in the exact same order and with the same parameters to keep the tags synchronized between all nodes.
Also note that StarPU will not check if a tag given to starpu_mpi_data_register() has been previously registered, this functionality only aims to prevent different parts of an application to use the same data tags.

12.9 Other MPI Utility Functions

Similarly to the function starpu_data_cpy(), the function starpu_mpi_data_cpy() can be used to transfer a data between 2 nodes. It behaves as starpu_data_cpy() if both data are owned by the same node, otherwise a transfer is initiated between the nodes. A priority and a callback function can be defined.

```
...
starpu_mpi_data_register(src_handle, 12, 0); // Data is owned by node0
starpu_mpi_data_register(dst_handle, 42, 1); // Data is owned by node1
...
starpu_mpi_data_cpy(dst_handle, src_handle, MPI_COMM_WORLD, 0, callback, NULL);
```

12.10 Pruning MPI Task Insertion

Making all MPI nodes process the whole graph 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, 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_task_insert(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.
An example is available in the file `examples/stencil/implicit-stencil-tasks.c`.
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.
To estimate quickly how long task submission takes, and notably how much pruning saves, a quick and easy way is to measure the submission time of just one of the MPI nodes. This can be achieved by running the application on just one MPI node with the following environment variables:

```
export STARPU_DISABLE_KERNELS=1
export STARPU_MPI_FAKE_RANK=2
export STARPU_MPI_FAKE_SIZE=1024
```

Here we have disabled the kernel function call to skip the actual computation time and only keep submission time, and we have asked StarPU to fake running on MPI node 2 out of 1024 nodes.

12.11 Temporary Data

To be able to use starpu_mpi_task_insert(), one has to call starpu_mpi_data_register(), so that StarPU-MPI can know what it needs to do for each data. Parameters of starpu_mpi_data_register() are normally the same on all nodes for a given data, so that all nodes agree on which node owns the data, and which tag is used to transfer its value.
It can however be useful to register e.g. some temporary data on just one node, without having to register a dumb handle on all nodes, while only one node will actually need to know about it. In this case, nodes which will not need the data can just pass `NULL` to starpu_mpi_task_insert():

```
starpu_data_handle_t data0 = NULL;
if (rank == 0)
{
```

```
    starpu_variable_data_register(&data0, STARPU_MAIN_RAM, (uintptr_t) &val0, sizeof(val0));
    starpu_mpi_data_register(data0, 0, rank);
}
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl, STARPU_W, data0, 0); /* Executes on node 0 */
```

Here, nodes whose rank is not `0` will simply not take care of the data, and consider it to be on another node.
This can be mixed various way, for instance here node `1` determines that it does not have to care about `data0`, but knows that it should send the value of its `data1` to node `0`, which owns data and thus will need the value of `data1` to execute the task:

```
starpu_data_handle_t data0 = NULL, data1, data;
if (rank == 0)
{
    starpu_variable_data_register(&data0, STARPU_MAIN_RAM, (uintptr_t) &val0, sizeof(val0));
    starpu_mpi_data_register(data0, -1, rank);
    starpu_variable_data_register(&data1, -1, 0, sizeof(val1));
    starpu_variable_data_register(&data, STARPU_MAIN_RAM, (uintptr_t) &val, sizeof(val));
}
else if (rank == 1)
{
    starpu_variable_data_register(&data1, STARPU_MAIN_RAM, (uintptr_t) &val1, sizeof(val1));
    starpu_variable_data_register(&data, -1, 0, sizeof(val));
}
starpu_mpi_data_register(data, 42, 0);
starpu_mpi_data_register(data1, 43, 1);
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl, STARPU_W, data, STARPU_R, data0, STARPU_R, data1, 0); /*
      Executes on node 0 */
```

The full source code is available in the file `mpi/tests/temporary.c`.

12.12 Per-node Data

Further than temporary data on just one node, one may want per-node data, to e.g. replicate some computation because that is less expensive than communicating the value over MPI:

```
starpu_data_handle pernode, data0, data1;
starpu_variable_data_register(&pernode, -1, 0, sizeof(val));
starpu_mpi_data_register(pernode, -1, STARPU_MPI_PER_NODE);
/* Normal data:  one on node0, one on node1 */
if (rank == 0)
{
    starpu_variable_data_register(&data0, STARPU_MAIN_RAM, (uintptr_t) &val0, sizeof(val0));
    starpu_variable_data_register(&data1, -1, 0, sizeof(val1));
}
else if (rank == 1)
{
    starpu_variable_data_register(&data0, -1, 0, sizeof(val1));
    starpu_variable_data_register(&data1, STARPU_MAIN_RAM, (uintptr_t) &val1, sizeof(val1));
}
starpu_mpi_data_register(data0, 42, 0);
starpu_mpi_data_register(data1, 43, 1);
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl, STARPU_W, pernode, 0); /* Will be replicated on all nodes */
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl2, STARPU_RW, data0, STARPU_R, pernode); /* Will execute on node
      0, using its own pernode*/
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl2, STARPU_RW, data1, STARPU_R, pernode); /* Will execute on node
      1, using its own pernode*/
```

One can turn a normal data into per-node data, by first broadcasting it to all nodes:

```
starpu_data_handle data;
starpu_variable_data_register(&data, -1, 0, sizeof(val));
starpu_mpi_data_register(data, 42, 0);
/* Compute some value */
starpu_mpi_task_insert(MPI_COMM_WORLD, &cl, STARPU_W, data, 0); /* Node 0 computes it */
/* Get it on all nodes */
starpu_mpi_get_data_on_all_nodes_detached(MPI_COMM_WORLD, data);
/* And turn it per-node */
starpu_mpi_data_set_rank(data, STARPU_MPI_PER_NODE);
```

The data can then be used just like per-node above.
The full source code is available in the file `mpi/tests/temporary.c`.

12.13 Inter-node reduction

One might want to leverage a reduction pattern across several nodes. Using STARPU_REDUX (see Data↩ Reduction), one can obtain such patterns where each core on contributing nodes spawns their own copy to work with. In the case that the required reductions are too numerous and expensive, the access mode STARPU_MPI_REDUX tells StarPU to spawn only one contribution per contributing node.
The setup and use of STARPU_MPI_REDUX is similar to STARPU_REDUX : the initialization and reduction codelets should be declared through starpu_data_set_reduction_methods() in the same fashion as STARPU_REDUX. Example `mpi/examples/mpi_redux/mpi_redux.c` shows how to use

the STARPU_MPI_REDUX mode and compare it with the standard STARPU_REDUX. The function starpu_mpi_redux_data() is automatically called either when a task reading the reduced handle is inserted through the MPI layer of StarPU through starpu_mpi_insert_task() or when users wait for all communications and tasks to be executed through starpu_mpi_wait_for_all(). The function can be called by users to fine-tune arguments such as the priority of the reduction tasks. Tasks contributing to the inter-node reduction should be registered as accessing the contribution through STARPU_RW|STARPU_COMMUTE mode, as for the STARPU_REDUX mode, as in the following example.

```
static struct starpu_codelet contrib_cl =
{
    .cpu_funcs = {cpu_contrib}, /* cpu implementation(s) of the routine */
    .nbuffers = 1, /* number of data handles referenced by this routine */
    .modes = {STARPU_RW | STARPU_COMMUTE} /* access modes for the contribution */
    .name = "contribution"
};
```

When inserting these tasks, the access mode handed out to the StarPU-MPI layer should be `STARPU_MPI_REDUX`. If a task uses a `data` owned by node 0 and is executed on the node 1, it can be inserted as in the following example.

```
starpu_mpi_task_insert(MPI_COMM_WORLD, &contrib_cl, STARPU_MPI_REDUX, data, STARPU_EXECUTE_ON_NODE, 1); /*
      Node 1 computes it */
```

Note that if the specified node is set to `-1`, the option is ignored.

More examples are available at `mpi/examples/mpi_redux/mpi_redux.c` and `mpi/examples/mpi_redux/mpi_redux_tree.c`.

12.14 Priorities

All send functions have a `_prio` variant which takes an additional priority parameter, which allows making StarPU-MPI change the order of MPI requests before submitting them to MPI. The default priority is `0`.

When using the starpu_mpi_task_insert() helper, STARPU_PRIORITY defines both the task priority and the MPI requests priority. An example is available in the file `mpi/examples/benchs/recv_wait_finalize_bench.c`.

To test how much MPI priorities have a good effect on performance, you can set the environment variable STARPU_MPI_PRIORITIES to `0` to disable the use of priorities in StarPU-MPI.

12.15 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 anymore (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`. A mere starpu_mpi_cache_flush_all_data() can for instance be added at the end of the whole algorithm, to express that no data will be reused after this (or at least that it is not interesting to keep them in cache). It may however be interesting to add fine-graph starpu_mpi_cache_flush() calls during the algorithm; the effect for the data deallocation will be the same, but it will additionally release some pressure from the StarPU-MPI cache hash table during task submission.

One can determine whether a piece of data is cached with starpu_mpi_cached_receive() and starpu_mpi_cached_send(). An example is available in the file `mpi/examples/cache/cache.c`.

Functions starpu_mpi_cached_receive_set() and starpu_mpi_cached_send_set() are automatically called by starpu_mpi_task_insert() but can also be called directly by the application. Functions starpu_mpi_cached_send_clear() and starpu_mpi_cached_receive_clear() must be called to clear data from the cache. They are also automatically called when using starpu_mpi_task_insert().

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.

12.16 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 which 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_data_migrate(MPI_COMM_WORLD, data_handles[x][y], mpi_rank);
        }
    }
}
```

The full example is available in the file `mpi/examples/stencil/stencil5.c`. From then on, further tasks submissions will use the new data distribution, which will thus change both MPI communications and task assignments.

Very importantly, since all nodes have to agree on which node owns which data 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_data_migrate(MPI_COMM_WORLD, data_handles[x][y], mpi_rank);
            /* And unregister it */
            starpu_data_unregister(data_handles[x][y]);
        }
    }
}
```

12.17 MPI Collective Operations

The functions are described in MPICollectiveOperations.

```
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], STARPU_MAIN_RAM, (uintptr_t)vector[x], blocks_size,
      sizeof(float));
    }
    else if ((mpi_rank == rank) || ((rank == mpi_rank+1 || rank == mpi_rank-1)))
    {
```

```
        /* I own this 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, NULL, NULL, NULL, NULL);
/* Calculation */
for(x = 0; x < nblocks ;  x++)
{
    if (data_handles[x])
    {
        int owner = starpu_data_get_rank(data_handles[x]);
        if (owner == rank)
        {
            starpu_task_insert(&cl, STARPU_RW, data_handles[x], 0);
        }
    }
}
/* Gather the matrix on main node */
starpu_mpi_gather_detached(data_handles, nblocks, 0, MPI_COMM_WORLD, NULL, NULL, NULL, NULL);
```

An example is available in `mpi/tests/mpi_scatter_gather.c`.

With NewMadeleine (see Using the NewMadeleine communication library), broadcasts can automatically be detected and be optimized by using routing trees. This behavior can be controlled with the environment variable STARPU_MPI_COOP_SENDS. See the corresponding `paper` for more information.

Other collective operations would be easy to define, just ask starpu-devel for them!

12.18 Make StarPU-MPI Progression Thread Execute Tasks

The default behavior of StarPU-MPI is to spawn an MPI thread to take care only of MPI communications in an active fashion (i.e. the StarPU-MPI thread sleeps only when there are no active request submitted by the application), with the goal of being as reactive as possible to communications. Knowing that, users usually leave one free core for the MPI thread when starting a distributed execution with StarPU-MPI. However, this could result in a loss of performance for applications that does not require an extreme reactivity to MPI communications.

The starpu_mpi_init_conf() routine allows users to give the starpu_conf configuration structure of StarPU (usually given to the starpu_init() routine) to StarPU-MPI, so that StarPU-MPI reserves for its own use one of the CPU drivers of the current computing node, or one of the CPU cores, and then calls starpu_init() internally.

This allows the MPI communication thread to call a StarPU CPU driver to run tasks when there is no active requests to take care of, and thus recover the computational power of the "lost" core. Since there is a trade-off between executing tasks and polling MPI requests, which is how much the application wants to lose in reactivity to MPI communications to get back the computing power of the core dedicated to the StarPU-MPI thread, there are two environment variables to pilot the behavior of the MPI thread so that users can tune this trade-off depending on the behavior of the application.

The STARPU_MPI_DRIVER_CALL_FREQUENCY environment variable sets how many times the MPI progression thread goes through the MPI_Test() loop on each active communication request (and thus try to make communications progress by going into the MPI layer) before executing tasks. The default value for this environment variable is 0, which means that the support for interleaving task execution and communication polling is deactivated, thus returning the MPI progression thread to its original behavior.

The STARPU_MPI_DRIVER_TASK_FREQUENCY environment variable sets how many tasks are executed by the MPI communication thread before checking all active requests again. While this environment variable allows a better use of the core dedicated to StarPU-MPI for computations, it also decreases the reactivity of the MPI communication thread as much.

12.19 Debugging MPI

Communication trace will be enabled when the environment variable STARPU_MPI_COMM is set to `1`, and StarPU has been configured with the option --enable-verbose.

Statistics will be enabled for the communication cache when the environment variable STARPU_MPI_CACHE_↩

STATS is set to `1`. It prints messages on the standard output when data are added or removed from the received communication cache.
When the environment variable STARPU_MPI_STATS is set to `1`, StarPU will display at the end of the execution for each node the volume and the bandwidth of data sent to all the other nodes. Communication statistics can also be enabled and disabled from the application by calling the functions starpu_mpi_comm_stats_enable() and starpu_mpi_comm_stats_disable(). If communication statistics have been enabled, calling the function starpu_mpi_comm_stats_retrieve() will give the amount of communications between the calling node and all the other nodes. Communication statistics will also be automatically displayed at the end of the execution, as exemplified below.

```
[starpu_comm_stats][3] TOTAL:   476.000000 B    0.000454 MB      0.000098 B/s    0.000000 MB/s
[starpu_comm_stats][3:0]        248.000000 B    0.000237 MB      0.000051 B/s    0.000000 MB/s
[starpu_comm_stats][3:2]        50.000000 B     0.000217 MB      0.000047 B/s    0.000000 MB/s

[starpu_comm_stats][2] TOTAL:   288.000000 B    0.000275 MB      0.000059 B/s    0.000000 MB/s
[starpu_comm_stats][2:1]        70.000000 B     0.000103 MB      0.000022 B/s    0.000000 MB/s
[starpu_comm_stats][2:3]        288.000000 B    0.000172 MB      0.000037 B/s    0.000000 MB/s

[starpu_comm_stats][1] TOTAL:   188.000000 B    0.000179 MB      0.000038 B/s    0.000000 MB/s
[starpu_comm_stats][1:0]        80.000000 B     0.000114 MB      0.000025 B/s    0.000000 MB/s
[starpu_comm_stats][1:2]        188.000000 B    0.000065 MB      0.000014 B/s    0.000000 MB/s

[starpu_comm_stats][0] TOTAL:   376.000000 B    0.000359 MB      0.000077 B/s    0.000000 MB/s
[starpu_comm_stats][0:1]        376.000000 B    0.000141 MB      0.000030 B/s    0.000000 MB/s
[starpu_comm_stats][0:3]        10.000000 B     0.000217 MB      0.000047 B/s    0.000000 MB/s
```

These statistics can be plotted as heatmaps using the StarPU tool `starpu_mpi_comm_matrix.py`, this will produce 2 PDF files, one plot for the bandwidth, and one plot for the data volume.

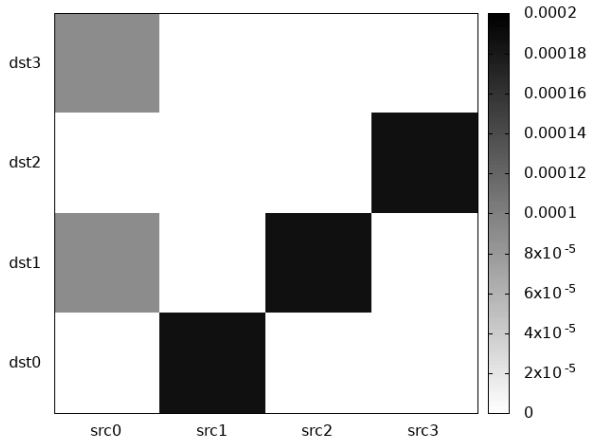


Figure 12.1 Bandwidth Heatmap

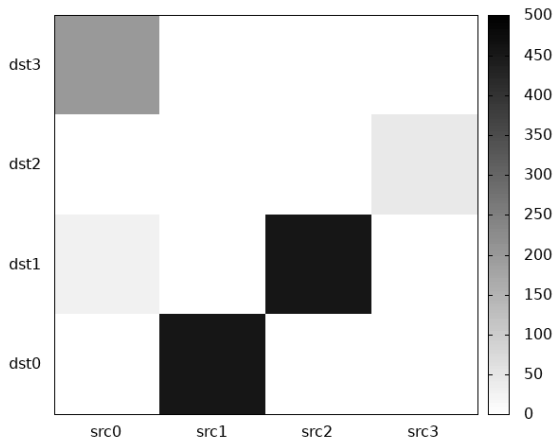


Figure 12.2 Data Volume Heatmap

12.20 More MPI examples

MPI examples are available in the StarPU source code in mpi/examples:

- `comm` shows how to use communicators with StarPU-MPI
- `complex` is a simple example using a user-define data interface over MPI (complex numbers),
- `stencil5` is a simple stencil example using starpu_mpi_task_insert(),
- `matrix_decomposition` is a cholesky decomposition example using starpu_mpi_task_insert(). The non-distributed version can check for <algorithm correctness in 1-node configuration, the distributed version uses exactly the same source code, to be used over MPI,
- `mpi_lu` is an LU decomposition example, provided in three versions: `plu_example` uses explicit MPI data transfers, `plu_implicit_example` uses implicit MPI data transfers, `plu_outofcore_example` uses implicit MPI data transfers and supports data matrices which do not fit in memory (out-of-core).

12.21 Using the NewMadeleine communication library

NewMadeleine (see https://pm2.gitlabpages.inria.fr/newmadeleine/, part of the PM2 project) is an optimizing communication library for high-performance networks. NewMadeleine provides its own interface, but also an MPI interface (called MadMPI). Thus, there are two possibilities to use NewMadeleine with StarPU:

- using the NewMadeleine's native interface. StarPU supports this interface from its release 1.3.0, by enabling the `configure` option --enable-nmad. In this case, StarPU relies directly on NewMadeleine to make communications progress and NewMadeleine has to be built with the profile `pukabi+madmpi.conf`.
- using the NewMadeleine's MPI interface (MadMPI). StarPU will use the standard MPI API and NewMadeleine will handle the calls to the MPI API. In this case, StarPU makes communications progress and thus communication progress has to be disabled in NewMadeleine by compiling it with the profile `pukabi+madmpi-mini.conf`.

To build NewMadeleine, download the latest version from the website (or, better, use the Git version to use the most recent version), then:

```
cd pm2/scripts
./pm2-build-packages ./<the profile you chose> --prefix=<installation prefix>
```

With Guix, the NewMadeleine's native interface can be used by setting the parameter `--with-input=openmpi=nmad` and MadMPI can be used with `--with-input=openmpi=nmad-mini`.
Whatever implementation (NewMadeleine or MadMPI) is used by StarPU, the public MPI interface of StarPU (described in MPI Support) is the same.

12.22 MPI Master Slave Support

StarPU provides another way to execute applications across many nodes. The Master Slave support permits to use remote cores without thinking about data distribution. This support can be activated with the `configure` option --enable-mpi-master-slave. However, you should not activate both MPI support and MPI Master-Slave support.
The existing kernels for CPU devices can be used as such. They only have to be exposed through the name of the function in the starpu_codelet::cpu_funcs_name field. Functions have to be globally-visible (i.e. not static) for StarPU to be able to look them up, and `-rdynamic` must be passed to gcc (or `-export-dynamic` to ld) so that symbols of the main program are visible.
By default, one core is dedicated on the master node to manage the entire set of slaves. If the implementation of MPI you are using has a good multiple threads support, you can set the STARPU_MPI_MS_MULTIPLE_THREAD environment variable to 1 to dedicate one core per slave.
Choosing the number of cores on each slave device is done by setting the environment variable STARPU_NMPIMSTHREADS=<number> with `<number>` being the requested number of cores. By default, all the slave's cores are used.
Setting the number of slaves nodes is done by changing the `-np` parameter when executing the application with mpirun or mpiexec.

The master node is by default the node with the MPI rank equal to 0. To select another node, use the environment variable STARPU_MPI_MASTER_NODE=<number> with <number> being the requested MPI rank node.
A simple example `tests/main/insert_task.c` can be used to test the MPI master slave support.

12.23 MPI Checkpoint Support

StarPU provides an experimental checkpoint mechanism. It is for now only a proof of concept to see what the checkpointing cost is, since the restart part has not been integrated yet.
To enable checkpointing, you should use the `configure` option --enable-mpi-ft. The application in the directory `mpi/examples/matrix_decomposition` shows how to enable checkpoints. The API documentation is available in MPI Fault Tolerance Support
Statistics can also be enabled with the `configure` option --enable-mpi-ft-stats.

Chapter 13

TCP/IP Support

13.1 TCP/IP Master Slave Support

StarPU provides a transparent way to execute applications across many nodes. The Master Slave support permits to use remote cores without thinking about data distribution. This support can be activated with the `configure` option --enable-tcpip-master-slave.
The existing kernels for CPU devices can be used as such. They only have to be exposed through the name of the function in the starpu_codelet::cpu_funcs_name field. Functions have to be globally-visible (i.e. not static) for StarPU to be able to look them up, and `-rdynamic` must be passed to gcc (or `-export-dynamic` to ld) so that symbols of the main program are visible.
By default, one core is dedicated on the master node to manage the entire set of slaves.
Choosing the number of cores on each slave device is done by setting the environment variable STARPU_↩ NTCPIPMSTHREADS=<number> with `<number>` being the requested number of cores. By default, all the slave's cores are used.
The master should be given the number of slaves that are expected to be run with the STARPU_TCPIP_MS_↩ SLAVES environment variable.
The slaves should then be started, and their number also should be given with the STARPU_TCPIP_MS_SLAVES environment variable. They should additionally be given the IP address of the master with the STARPU_TCPIP_↩ MS_MASTER environment variable.
For simple local checks, one can use the `starpu_tcpipexec` tool, which just starts the application several times. Setting the number of slaves nodes is done by changing the `-np` parameter.

Chapter 14

Transactions

14.1 General Ideas

StarPU's transactions enable the cancellation of a sequence of already submitted tasks based on a just-in-time decision. The purpose of this mechanism is typically for iterative applications to submit tasks for the next iteration ahead of time while leaving some iteration loop criterion (e.g. convergence) to be evaluated just before the first task of the next iteration is about to be scheduled. Such a sequence of collectively cancelable tasks is called a transaction *epoch*.

14.2 Usage

Some examples illustrating the usage of StarPU's transactions are available in the directory `examples/transactions`.

14.2.1 Epoch Cancellation

If the start criterion of an epoch evaluates to `False`, all the tasks for that next epoch are canceled. Thus, StarPU's transactions let applications avoid the use of synchronization barriers commonly found between the task submission sequences of subsequent iterations, and avoid breaking the flow of dependencies in the process. Moreover, while the kernel functions of canceled transaction tasks are not executed, their dependencies are still honored in the proper order.

14.2.2 Transactions Enabled Codelets

Codelets for tasks being part of a transaction should set their `nbuffers` field to STARPU_VARIABLE_NBUFFERS.

14.2.3 Transaction Creation

A `struct starpu_transaction` opaque object is created using the starpu_transaction_open() function, specifying a transaction start criterion callback and some user argument to be passed to that callback upon the first call. The start criterion callback should return `True` (e.g. `!0`) if the next transaction epoch should proceed, or `False` (e.g. `0`) if the tasks belonging to that next epoch should be canceled. `starpu_transaction_open()` submits an internal task to mark the beginning of the transaction. If submitting that internal task fails with ENODEV, `starpu_transaction_open()` will return `NULL`.

14.2.4 Transaction Tasks

Tasks governed by the same transaction object should be passed that transaction object either through the .transaction field of starpu_task structures, using the STARPU_TRANSACTION argument of starpu_task_insert().

14.2.5 Epoch Transition

The transition from one transaction epoch to the next is expressed using the starpu_transaction_next_epoch function to which the `starpu_transaction` object and a user argument are passed. Upon a call to that function,

the start criterion callback is evaluated on users argument to decide whether the next epoch should proceed or be canceled.

14.2.6 Transaction Closing

The last epoch should be ended through a call to starpu_transaction_close().

14.3 Known limitations

Support for transactions is experimental.
StarPU's transactions are currently not compatible with StarPU-MPI distributed sessions.

Chapter 15

Fault Tolerance

15.1 Introduction

Due to e.g. hardware error, some tasks may fail, or even complete nodes may fail. For now, StarPU provides some support for failure of tasks.

15.2 Retrying tasks

In case a task implementation notices that it fail to compute properly, it can call starpu_task_failed() to notify StarPU of the failure.
`tests/fault-tolerance/retry.c` is an example of coping with such failure: the principle is that when submitting the task, one sets its prologue callback to starpu_task_ft_prologue(). That prologue will turn the task into a meta task, which will manage the repeated submission of try-tasks to perform the computation until one of the computations succeeds. One can create a try-task for the meta task by using starpu_task_ft_create_retry().
By default, try-tasks will be just retried until one of them succeeds (i.e. the task implementation does not call starpu_task_failed()). One can change the behavior by passing a `check_failsafe` function as prologue parameter, which will be called at the end of the try-task attempt. It can look at `starpu_task_get_current()->failed` to determine whether the try-task succeeded, in which case it can call starpu_task_ft_success() on the meta-task to notify success, or if it failed, in which case it can call starpu_task_failsafe_create_retry() to create another try-task, and submit it with starpu_task_submit_nodeps().
This can however only work if the task input is not modified, and is thus not supported for tasks with data access mode STARPU_RW.

Chapter 16

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 be 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 preferable), 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 preferable 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.
Some examples illustrating the usage of FFT API are available in the directory `starpufft/tests`.

16.1 Compilation

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

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

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

Chapter 17

SOCL OpenCL Extensions

SOCL is an OpenCL implementation based on StarPU. It gives 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.
Setting the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` flag on a command queue also allows StarPU to reorder kernels queued on the queue, otherwise they would be serialized, and several command queues would be necessary to see kernels dispatched to the various OpenCL devices.
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 socl.icd ICD file is 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
  Platform Profile:     FULL_PROFILE
  Platform Version:     OpenCL 1.1 CUDA 4.2.1
  Platform Name:        NVIDIA CUDA
  Platform Vendor:      NVIDIA Corporation
  Platform Extensions:  cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options cl_n

  Platform Profile:     FULL_PROFILE
  Platform Version:     OpenCL 1.0 SOCL Edition (0.1.0)
  Platform Name:        SOCL Platform
  Platform Vendor:      Inria
  Platform 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 18

Hierarchical DAGS

The STF model has the intrinsic limitation of supporting static task graphs only, which leads to potential submission overhead and to a static task graph which is not necessarily adapted for execution on heterogeneous systems.
To address these problems, we have extended the STF model to enable tasks subgraphs at runtime. We refer to these tasks as *hierarchical tasks*. This approach allows for a more dynamic task graph. This allows to dynamically adapt the granularity to meet the optimal size of the targeted computing resource.
Hierarchical tasks are tasks that can transform themselves into a new task-graph dynamically at runtime. Programmers submit a coarse version of the DAG, called the bubbles graph, which represents the general shape of the application tasks graph. The execution of this bubble graph will generate and submit the computing tasks of the application. It is up to application programmers to decide how to build the bubble graph (i.e. how to structure the computation tasks graph to create some groups of tasks). Dependencies between bubbles are automatically deduced from dependencies between their computing tasks.

18.1 An Example

In order to understand the hierarchical tasks model, an example of "bubblification" is showed here. We start from a simple example, multiplying the elements of a vector.

18.1.1 Initial Version

A computation is done several times on a vector split in smaller vectors. For each step and each sub-vector, a task is generated to perform the computation.

```
void func_cpu(void *descr[], void *_args)
{
        (void) _args;
        int x;
        int nx = STARPU_VECTOR_GET_NX(descr[0]);
        TYPE *v = (TYPE *)STARPU_VECTOR_GET_PTR(descr[0]);
        for(x=0 ; x<nx ; x++)
                v[x] += 1;
}
struct starpu_codelet vector_cl =
{
        .cpu_funcs = {func_cpu},
        .nbuffers = 1,
        .modes = {STARPU_RW}
};
int vector_no_bubble()
{
        TYPE *vector;
        starpu_data_handle_t vhandle;
        /* ... */
        starpu_vector_data_register(&vhandle, 0, (uintptr_t)vector, X, sizeof(vector[0]));
        starpu_data_map_filters(vhandle, 1, &f);
        for(loop=0 ; loop<NITER; loop++)
                for (x = 0; x < SLICES; x++)
                {
                        starpu_task_insert(&vector_cl,
                                           STARPU_RW, starpu_data_get_sub_data(vhandle, 1, x),
                                           0);
                }
        starpu_data_unpartition(vhandle, STARPU_MAIN_RAM);
        starpu_data_unregister(vhandle);
        /* ... */
}
```

18.1.2 Bubble Version

The bubble version of the code replaces the inner loop that realizes the tasks insertion by a call to a bubble creation. At its execution, the bubble will insert the computing tasks. The bubble graph is built accordingly to the dependencies of the subdata.

```
void no_func(void *buffers[], void *arg)
{
        assert(0);
        return;
}
int is_bubble(struct starpu_task *t, void *arg)
{
        (void)arg;
        (void)t;
        return 1;
}
void bubble_gen_dag(struct starpu_task *t, void *arg)
{
        int i;
        starpu_data_handle_t *subdata = (starpu_data_handle_t *)arg;
        for(i=0 ; i<SLICES ; i++)
        {
                starpu_task_insert(&vector_cl,
                                   STARPU_RW, subdata[i],
                                   0);
                STARPU_CHECK_RETURN_VALUE(ret, "starpu_task_insert");
        }
}
struct starpu_codelet bubble_codelet =
{
        .cpu_funcs = {no_func},
        .bubble_func = is_bubble,
        .bubble_gen_dag_func = bubble_gen_dag,
        .nbuffers = 1
};
int vector_bubble()
{
        TYPE *vector;
        starpu_data_handle_t vhandle;
        starpu_data_handle_t sub_handles[SLICES];
        /* ... */
        starpu_vector_data_register(&vhandle, 0, (uintptr_t)vector, X, sizeof(vector[0]));
        starpu_data_partition_plan(vhandle, &f, sub_handles);
        for(loop=0 ; loop<NITER; loop++)
        {
                starpu_task_insert(&bubble_codelet,
                                   STARPU_RW, vhandle,
                                   STARPU_NAME, "B1",
                                   STARPU_BUBBLE_GEN_DAG_FUNC_ARG, sub_handles,
                                   0);
        }
        starpu_data_partition_clean(vhandle, SLICES, sub_handles);
        starpu_data_unregister(vhandle);
        /* ... */
}
```

The full example is available in the file `bubble/tests/vector/vector.c`.

To define a hierarchical task, one needs to define the fields starpu_codelet::bubble_func and starpu_codelet::bubble_gen_dag_func.

The field starpu_codelet::bubble_func is a pointer function which will be executed by StarPU to decide at runtime if the task must be transformed into a bubble. If the function returns a non-zero value, the function starpu_codelet::bubble_gen_dag_func will be executed to create the new graph of tasks.

The pointer functions can also be defined when calling starpu_task_insert() by using the arguments STARPU_BUBBLE_FUNC and STARPU_BUBBLE_GEN_DAG_FUNC. Both these functions can be passed parameters through the arguments STARPU_BUBBLE_FUNC_ARG and STARPU_BUBBLE_GEN_DAG_FUNC_ARG

When executed, the function starpu_codelet::bubble_func will be given as parameter the task being checked, and the value specified with STARPU_BUBBLE_FUNC_ARG.

When executed, the function starpu_codelet::bubble_gen_dag_func will be given as parameter the task being turned into a hierarchical task and the value specified with STARPU_BUBBLE_GEN_DAG_FUNC_ARG.

An example involving these functions is in `bubble/tests/basic/brec.c`. And more examples are available in `bubble/tests/basic/*.c`.

Chapter 19

Parallel Workers

19.1 General Ideas

Parallel workers are a concept introduced in this paper where they are called clusters.
The granularity problem is tackled by using resource aggregation: instead of dynamically splitting tasks, resources are aggregated to process coarse grain tasks in a parallel fashion. This is built on top of scheduling contexts to be able to handle any type of parallel tasks.
This comes from a basic idea, making use of two levels of parallelism in a DAG. We keep the DAG parallelism, but consider on top of it that a task can contain internal parallelism. A good example is if each task in the DAG is OpenMP enabled.
The particularity of such tasks is that we will combine the power of two runtime systems: StarPU will manage the DAG parallelism and another runtime (e.g. OpenMP) will manage the internal parallelism. The challenge is in creating an interface between the two runtime systems so that StarPU can regroup cores inside a machine (creating what we call a **parallel worker**) on top of which the parallel tasks (e.g. OpenMP tasks) will be run in a contained fashion.
The aim of the parallel worker API is to facilitate this process automatically. For this purpose, we depend on the `hwloc` tool to detect the machine configuration and then partition it into usable parallel workers.
An example of code running on parallel workers is available in `examples/sched_ctx/parallel_workers.c`.
Let's first look at how to create a parallel worker.
To enable parallel workers in StarPU, one needs to set the configure option --enable-parallel-worker.

19.2 Workers Creating Parallel Workers

Partitioning a machine into parallel workers with the parallel worker API is fairly straightforward. The simplest way is to state under which machine topology level we wish to regroup all resources. This level is a `hwloc` object, of the type `hwloc_obj_type_t`. More information can be found in the hwloc documentation.
Once a parallel worker is created, the full machine is represented with an opaque structure starpu_parallel_worker_config. This can be printed to show the current machine state.

```
struct starpu_parallel_worker_config *parallel_workers;
parallel_workers = starpu_parallel_worker_init(HWLOC_OBJ_SOCKET, 0);
starpu_parallel_worker_print(parallel_workers);
/* submit some tasks with OpenMP computations */
starpu_parallel_worker_shutdown(parallel_workers);
/* we are back to the default StarPU state */
```

The following graphic is an example of what a particular machine can look like once parallel workers are created. The main difference is that we have less worker queues and tasks which will be executed on several resources at once. The execution of these tasks will be left to the internal runtime system, represented with a dashed box around the resources.

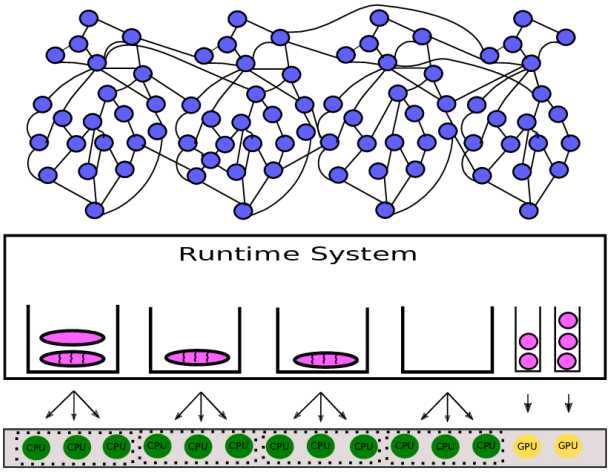


Figure 19.1 StarPU using parallel tasks

Creating parallel workers as shown in the example above will create workers able to execute OpenMP code by default. The parallel worker creation function starpu_parallel_worker_init() takes optional parameters after the `hwloc` object (always terminated by the value `0`) which allow parametrizing the parallel workers creation. These parameters can help to create parallel workers of a type different from OpenMP, or create a more precise partition of the machine.
This is explained in Section Creating Custom Parallel Workers.
Before starpu_shutdown(), we call starpu_parallel_worker_shutdown() to delete the parallel worker configuration.

19.3 Example Of Constraining OpenMP

Parallel workers require being able to constrain the runtime managing the internal task parallelism (internal runtime) to the resources set by StarPU. The purpose of this is to express how StarPU must communicate with the internal runtime to achieve the required cooperation. In the case of OpenMP, StarPU will provide an awake thread from the parallel worker to execute this liaison. It will then provide on demand the process ids of the other resources supposed to be in the region. Finally, thanks to an OpenMP region, we can create the required number of threads and bind each of them on the correct region. These will then be reused each time we encounter a `#pragma omp parallel` in the following computations of our program.
The following graphic is an example of what an OpenMP-type parallel worker looks like and how it is represented in StarPU. We can see that one StarPU (black) thread is awake, and we need to create on the other resources the OpenMP threads (in pink).

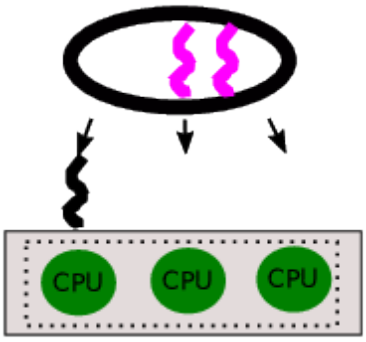


Figure 19.2 StarPU with an OpenMP parallel worker

Finally, the following code shows how to force OpenMP to cooperate with StarPU and create the aforementioned OpenMP threads constrained in the parallel worker's resources set:

```
void starpu_parallel_worker_openmp_prologue(void * sched_ctx_id)
{
  int sched_ctx = *(int*)sched_ctx_id;
  int *cpuids = NULL;
  int ncpuids = 0;
  int workerid = starpu_worker_get_id();
```

```
  //we can target only CPU workers
  if (starpu_worker_get_type(workerid) == STARPU_CPU_WORKER)
  {
    //grab all the ids inside the parallel worker
    starpu_sched_ctx_get_available_cpuids(sched_ctx, &cpuids, &ncpuids);
    //set the number of threads
    omp_set_num_threads(ncpuids);
#pragma omp parallel
    {
      //bind each threads to its respective resource
      starpu_sched_ctx_bind_current_thread_to_cpuid(cpuids[omp_get_thread_num()]);
    }
    free(cpuids);
  }
  return;
}
```

This function is the default function used when calling starpu_parallel_worker_init() without extra parameter.
Parallel workers are based on several tools and models already available within StarPU contexts, and merely extend contexts. More on contexts can be read in Section Scheduling Contexts.
A similar example is available in the file `examples/sched_ctx/parallel_code.c`.

19.4 Creating Custom Parallel Workers

Parallel workers can be created either with the predefined types provided within StarPU, or with user-defined functions to bind another runtime inside StarPU.
The predefined parallel worker types provided by StarPU are STARPU_PARALLEL_WORKER_OPENMP, STARPU_PARALLEL_WORKER_INTEL_OPENMP_MKL and STARPU_PARALLEL_WORKER_GNU_OPENMP_MKL.
If StarPU is compiled with the `MKL` library, STARPU_PARALLEL_WORKER_GNU_OPENMP_MKL uses MKL functions to set the number of threads, which is more reliable when using an OpenMP implementation different from the Intel one. Otherwise, it will behave as STARPU_PARALLEL_WORKER_INTEL_OPENMP_MKL.
The parallel worker type is set when calling the function starpu_parallel_worker_init() with the parameter STARPU_PARALLEL_WORKER_TYPE as in the example below, which is creating a `MKL` parallel worker.

```
struct starpu_parallel_worker_config *parallel_workers;
parallel_workers = starpu_parallel_worker_init(HWLOC_OBJ_SOCKET,
                                STARPU_PARALLEL_WORKER_TYPE, STARPU_PARALLEL_WORKER_GNU_OPENMP_MKL,
                                0);
```

Using the default type STARPU_PARALLEL_WORKER_OPENMP is similar to calling starpu_parallel_worker_init() without any extra parameter.
An example is available in `examples/parallel_workers/parallel_workers.c`.
Users can also define their own function.

```
void foo_func(void* foo_arg);
int foo_arg = 0;
struct starpu_parallel_worker_config *parallel_workers;
parallel_workers = starpu_parallel_worker_init(HWLOC_OBJ_SOCKET,
                                 STARPU_PARALLEL_WORKER_CREATE_FUNC, &foo_func,
                                 STARPU_PARALLEL_WORKER_CREATE_FUNC_ARG, &foo_arg,
                                 0);
```

An example is available in `examples/parallel_workers/parallel_workers_func.c`.
Parameters that can be given to starpu_parallel_worker_init() are STARPU_PARALLEL_WORKER_MIN_NB, STARPU_PARALLEL_WORKER_MAX_NB, STARPU_PARALLEL_WORKER_NB, STARPU_PARALLEL_WORKER_POLICY_NAME STARPU_PARALLEL_WORKER_POLICY_STRUCT, STARPU_PARALLEL_WORKER_KEEP_HOMOGENEOUS, STARPU_PARALLEL_WORKER_PREFERE_MIN, STARPU_PARALLEL_WORKER_CREATE_FUNC, STARPU_PARALLEL_WORK STARPU_PARALLEL_WORKER_TYPE, STARPU_PARALLEL_WORKER_AWAKE_WORKERS, STARPU_PARALLEL_WORKER_P STARPU_PARALLEL_WORKER_NEW and STARPU_PARALLEL_WORKER_NCORES.

19.5 Parallel Workers With Scheduling

As previously mentioned, the parallel worker API is implemented on top of Scheduling Contexts. Its main addition is to ease the creation of a machine CPU partition with no overlapping by using `hwloc`, whereas scheduling contexts can use any number of any type of resources.
It is therefore possible, but not recommended, to create parallel workers using the scheduling contexts API. This can be useful mostly in the most complex machine configurations, where users have to dimension precisely parallel workers by hand using their own algorithm.

```
/* the list of resources the context will manage */
int workerids[3] = {1, 3, 10};
/* indicate the list of workers assigned to it, the number of workers,
the name of the context and the scheduling policy to be used within
```

```
the context */
int id_ctx = starpu_sched_ctx_create(workerids, 3, "my_ctx", 0);
/* let StarPU know that the following tasks will be submitted to this context */
starpu_sched_ctx_set_task_context(id);
task->prologue_callback_pop_func=&runtime_interface_function_here;
/* submit the task to StarPU */
starpu_task_submit(task);
```

As this example illustrates, creating a context without scheduling policy will create a parallel worker. The interface function between StarPU and the other runtime must be specified through the field starpu_task::prologue_callback_pop_func. Such a function can be similar to the OpenMP thread team creation one (see above). An example is available in `examples/sched_ctx/parallel_tasks_reuse_handle.c`.

Note that the OpenMP mode is the default mode both for parallel workers and contexts. The result of a parallel worker creation is a woken-up master worker and sleeping "slaves" which allow the master to run tasks on their resources.

To create a parallel worker with woken-up workers, the flag STARPU_SCHED_CTX_AWAKE_WORKERS must be set when using the scheduling context API function starpu_sched_ctx_create(), or the flag STARPU_PARALLEL_WORKER_AWAKE_WORKERS must be set when using the parallel worker API function starpu_parallel worker_init().

Chapter 20

Interoperability Support

In situations where multiple parallel software elements have to coexist within the same application, uncoordinated accesses to computing units may lead such parallel software elements to collide and interfere. The purpose of the Interoperability routines of StarPU, implemented along the definition of the Resource Management APIs of Project H2020 INTERTWinE, is to enable StarPU to coexist with other parallel software elements without resulting in computing core oversubscription or undersubscription. These routines allow the programmer to dynamically control the computing resources allocated to StarPU, to add or remove processor cores and/or accelerator devices from the pool of resources used by StarPU's workers to execute tasks. They also allow multiple libraries and applicative codes using StarPU simultaneously to select distinct sets of resources independently. Internally, the Interoperability Support is built on top of Scheduling Contexts (see Scheduling Contexts).

20.1 StarPU Resource Management

The `starpurm` module is a library built on top of the `starpu` library. It exposes a series of routines prefixed with `starpurm_` defining the resource management API.
All functions are defined in Interoperability Support.

20.1.1 Linking a program with the starpurm module

The `starpurm` module must be linked explicitly with the applicative executable using it. Example Makefiles in the `starpurm/dev/` subdirectories show how to do so. If the `pkg-config` command is available and the `PKG_↩CONFIG_PATH` environment variable is properly positioned, the proper settings may be obtained with the following `Makefile` snippet:

```
CFLAGS += $(shell pkg-config --cflags starpurm-1.4)
LDFLAGS+= $(shell pkg-config --libs-only-L starpurm-1.4)
LDLIBS += $(shell pkg-config --libs-only-l starpurm-1.4)
```

20.1.2 Initialization and Shutdown

The `starpurm` module is initialized with a call to starpurm_initialize() and must be finalized with a call to starpurm_shutdown(). The basic example is available in `starpurm/tests/01_init_exit.c`. The `starpurm` module supports CPU cores as well as devices. An integer ID is assigned to each supported device type. The ID assigned to a given device type can be queried with the starpurm_get_device_type_id() routine, which currently expects one of the following strings as argument and returns the corresponding ID:

- `"cpu"`
- `"opencl"`
- `"cuda"`

The `cpu` pseudo device type is defined for convenience and designates CPU cores. The number of units of each type available for computation can be obtained with a call to starpurm_get_nb_devices_by_type().
Each CPU core unit available for computation is designated by its rank among the StarPU CPU worker threads and by its own CPUSET bit. Each non-CPU device unit can be designated both by its rank number in the type, and by the CPUSET bit corresponding to its StarPU device worker thread. The CPUSET of a computing unit or

its associated worker can be obtained from its type ID and rank with starpurm_get_device_worker_cpuset(), which returns the corresponding HWLOC CPUSET.
An example is available in `starpurm/tests/02_list_units.c`.

20.1.3 Default Context

The `starpurm` module assumes a default, global context, manipulated through a series of routines allowing to assign and withdraw computing units from the main StarPU context. Assigning CPU cores can be done with starpurm_assign_cpu_to_starpu() and starpurm_assign_cpu_mask_to_starpu(), and assigning device units can be done with starpurm_assign_device_to_starpu() and starpurm_assign_device_mask_to_starpu(). Conversely, withdrawing CPU cores can be done with starpurm_withdraw_cpu_from_starpu() and starpurm_withdraw_cpu_mask_from_starpu(), and withdrawing device units can be done with starpurm_withdraw_device_from_starpu() and starpurm_withdraw_device_mask_from_
These routine should typically be used to control resource usage for the main applicative code. An example is available in `starpurm/examples/block_test/block_test.c`.

20.1.4 Temporary Contexts

Besides the default, global context, `starpurm` can create temporary contexts and launch the computation of kernels confined to these temporary contexts. The routine starpurm_spawn_kernel_on_cpus() can be used to do so: it allocates a temporary context and spawns a kernel within this context. The temporary context is subsequently freed upon completion of the kernel. The temporary context is set as the default context for the kernel throughout its lifespan. This routine should typically be used to control resource usage for a parallel kernel, handled by an external library built on StarPU. Internally, it relies on the use of starpu_sched_ctx_set_context() to set the temporary context as the default context for the parallel kernel, and then restore the main context upon completion. Note: the maximum number of temporary contexts allocated concurrently at any time should not exceed STARPU_NMAX_SCHED_CTXS-2, otherwise, the call to starpurm_spawn_kernel_on_cpus() may block until a temporary context becomes available. The routine starpurm_spawn_kernel_on_cpus() returns upon the completion of the parallel kernel. An example is available in `starpurm/examples/spawn.c`. An asynchronous variant is available with the routine starpurm_spawn_kernel_on_cpus_callback(). This variant returns immediately, however it accepts a callback function, which is subsequently called to notify the calling code about the completion of the parallel kernel. An example is available in `starpurm/examples/async_spawn.c`.

Chapter 21

SimGrid Support

StarPU can use SimGrid in order to simulate execution on an arbitrary platform.
The principle is to first run the application natively on the platform that one wants to laterlater simulate, and let StarPU record performance models. One then recompiles StarPU and the application in simgrid mode, where everything is executed the same, except the execution of the codelet function, and the data transfers, which are replaced by virtual sleeps based on the performance models. This thus allows to use the performance model for tasks and data transfers, while executing natively all the rest (the task scheduler and the application, notably).
This was tested with SimGrid from 3.11 to 3.16, and 3.18 to 3.36. SimGrid version 3.25 needs to be configured with -Denable_msg=ON . Other versions may have compatibility issues. 3.17 notably does not build at all. MPI simulation does not work with version 3.22.
If you have installed SimGrid by hand, make sure to set `PKG_CONFIG_PATH` to the path where `simgrid.pc` was installed:

```
$ export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/where/simgrid/installed/lib/ppkgconfig/simgrid.pc
```

21.1 Preparing 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. A basic example is available in `tests/main/empty_task.c`.
For some technical reason, the application's .c file which contains `main()` has to be recompiled with `starpu_simgrid_wrap.h`, which in the SimGrid case will `# define main()` into `starpu_main()`, and it is `libstarpu` which will provide the real `main()` and will call the application's `main()`. Including `starpu.h` will already include `starpu_simgrid_wrap.h`, so usually you would not need to include `starpu_simgrid_wrap.h` explicitly, but if for some reason including the whole `starpu.h` header is not possible, you can include `starpu_simgrid_wrap.h` explicitly.
To be able to test with crazy data sizes, one may want to only allocate application data if the macro `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 starpu_codelet::cuda_funcs with (void*)1, to express that there is a CUDA implementation, even if one does not actually provide it. StarPU will not actually run it in SimGrid mode anyway by default (unless the STARPU_CODELET_SIMGRID_EXECUTE or STARPU_CODELET_SIMGRID_EXECUTE_AND_INJECT flags are set in the codelet)

```
static struct starpu_codelet cl_potrf =
{
        .cpu_funcs = {chol_cpu_codelet_update_potrf},
        .cpu_funcs_name = {"chol_cpu_codelet_update_potrf"},
#ifdef STARPU_USE_CUDA
        .cuda_funcs = {chol_cublas_codelet_update_potrf},
#elif defined(STARPU_SIMGRID)
        .cuda_funcs = {(void*)1},
#endif
        .nbuffers = 1,
        .modes = {STARPU_RW},
        .model = &chol_model_potrf
};
```

The full example is available in `examples/cholesky/cholesky_kernels.c`.

21.2 Calibration

The idea is to first compile StarPU normally, and run the application, 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.

21.3 Simulation

Then, recompile StarPU, passing --enable-simgrid to `configure`. Make sure to keep all the other `configure` options the same, and notably options such as `-enable-maxcudadev`.

```
$ ./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 is 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.

21.4 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. To use multiple performance models in different ranks, in case of smpi executions in a heterogeneous platform, it is possible to use the option `-hostfile-platform` in `starpu_smpirun`, that will define STARPU_MPI_HOSTNAMES with the hostnames of your hostfile.
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 that case. Since during SimGrid execution, the functions of the codelet are actually not called by default, one can use dummy functions such as the following to still permit CUDA or OpenCL execution.

21.5 Simulation Examples

StarPU ships a few performance models for a couple of systems: `attila`, `mirage`, `idgraf`, and `sirocco`. See Section SimulatedBenchmarks for the details.

21.6 Simulations On Fake Machines

It is possible to build fake machines which do not exist, by modifying the platform file in `$STARPU_HOME/.starpu/sampling/bus/machine.platform.xml` by hand: one can add more CPUs, add GPUs (but the performance model file has to be extended as well), change the available GPU memory size, PCI memory bandwidth, etc.

21.7 Tweaking Simulation

The simulation can be tweaked, to be able to tune it between a very accurate simulation and a very simple simulation (which is thus close to scheduling theory results), see the STARPU_SIMGRID_TRANSFER_COST, STARPU_SIMGRID_CUDA_MALLOC_COST, STARPU_SIMGRID_CUDA_QUEUE_COST, STARPU_SIMGRID_TASK_SUBMIT_COST, STARPU_SIMGRID_TASK_PUSH_COST, STARPU_SIMGRID_FETCHING_INPUT_COST and STARPU_SIMGRID_SCHED_COST environment variables.

21.8 MPI Applications

StarPU-MPI applications can also be run in SimGrid mode. smpi currently requires that StarPU be build statically only, so `-disable-shared` needs to be passed to `./configure`.
The application needs to be compiled with `smpicc`, and run using the `starpu_smpirun` script, for instance:

```
$ STARPU_SCHED=dmda starpu_smpirun -platform cluster.xml -hostfile hostfile ./mpi/tests/pingpong
```

Where `cluster.xml` is a SimGrid-MPI platform description, and `hostfile` the list of MPI nodes to be used. Examples of such files are available in `tools/perfmodels`. In homogeneous MPI clusters: for each MPI node, it will just replicate the architecture referred by STARPU_HOSTNAME. To use multiple performance models in different ranks, in case of a heterogeneous platform, it is possible to use the option `-hostfile-platform` in `starpu_smpirun`, that will define STARPU_MPI_HOSTNAMES with the hostnames of your hostfile.
To use FxT traces, libfxt itself also needs to be built statically, **and** with dynamic linking flags, i.e. with

```
CFLAGS=-fPIC ./configure --enable-static
```

21.9 Debugging Applications

By default, SimGrid uses its own implementation of threads, which prevents `gdb` from being able to inspect stacks of all threads. To be able to fully debug an application running with SimGrid, pass the `-cfg=contexts/factory:thread` option to the application, to make SimGrid use system threads, which `gdb` will be able to manipulate as usual.
It is also worth noting SimGrid 3.21's new parameter `-cfg=simix/breakpoint` which allows putting a breakpoint at a precise (deterministic!) timing of the execution. If for instance in an execution trace we see that something odd is happening at time 19000ms, we can use `-cfg=simix/breakpoint:19.000` and `SIGTRAP` will be raised at that point, which will thus interrupt execution within `gdb`, allowing to inspect e.g. scheduler state, etc.

21.10 Memory Usage

Since kernels are not actually run and data transfers are not actually performed, the data memory does not actually need to be allocated. This allows for instance to simulate the execution of applications processing very big data on a small laptop.
The application can for instance pass `1` (or whatever bogus pointer) to StarPU data registration functions, instead of allocating data. This will however require the application to take care of not trying to access the data, and will not work in MPI mode, which performs transfers.

Another way is to pass the STARPU_MALLOC_SIMULATION_FOLDED flag to the starpu_malloc_flags() function. An example is available in `examples/mult/xgemm.c` This will make it allocate a memory area which one can read/write, but optimized so that this does not actually consume memory. Of course, the values read from such area will be bogus, but this allows the application to keep e.g. data load, store, initialization as it is, and also work in MPI mode. A more aggressive alternative is to pass also the STARPU_MALLOC_SIMULATION_UNIQUE flag (alongside with STARPU_MALLOC_SIMULATION_FOLDED) to the starpu_malloc_flags() function. An example is available in `examples/cholesky/cholesky_tag.c` . This will make StarPU reuse the pointers for allocations of the same size without calling the folded allocation again, thus decreasing some pressure on memory management.
Note however that notably Linux kernels refuse obvious memory overcommitting by default, so a single allocation can typically not be bigger than the amount of physical memory, see `https://www.kernel.org/doc/Documentation/vm/overcommit-accounting` This prevents for instance from allocating a single huge matrix. Allocating a huge matrix in several tiles is not a problem, however. `sysctl vm.overcommit_memory=1` can also be used to allow such overcommit.
Note however that this folding is done by remapping the same file several times, and Linux kernels will also refuse to create too many memory areas. `sysctl vm.max_map_count` can be used to check and change the default (65535). By default, StarPU uses a 1MiB file, so it hopefully fits in the CPU cache. However, this limits the amount of such folded memory to a bit below 64GiB. The STARPU_MALLOC_SIMULATION_FOLD environment variable can be used to increase the size of the file.

Chapter 22

Debugging Tools

StarPU provides several tools to help debugging applications. Execution traces can be generated and displayed graphically, see GeneratingTracesWithFxT.

22.1 TroubleShooting In General

Generally-speaking, if you have troubles, pass --enable-debug to `configure` to enable some checks which impact performance, but will catch common issues, possibly earlier than the actual problem you are observing, which may just be a consequence of a bug that happened earlier. Also, make sure not to have the --enable-fast `configure` option, which drops very useful catchup assertions. If your program is valgrind-safe, you can use it, see Using Other Debugging Tools.
Depending on your toolchain, it might happen that you get `undefined reference to '__stack_chk_guard'` errors. In that case, use the `-disable-fstack-protector-all` option to avoid the issue.
Then, if your program crashes with an assertion error, a segfault, etc. you can send us the result of

```
thread apply all bt
```

run in `gdb` at the point of the crash.
In case your program just hangs, but it may also be useful in case of a crash too, it helps to source `gdbinit` as described in the next section to be able to run and send us the output of the following commands:

```
starpu-workers
starpu-tasks
starpu-print-requests
starpu-print-prequests
starpu-print-frrequests
starpu-print-irrequests
```

To give us an idea of what is happening within StarPU. If the outputs are not too long, you can even run

```
starpu-all-tasks
starpu-print-all-tasks
starpu-print-datas-summary
starpu-print-datas
```

22.2 Using The Gdb Debugger

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

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

For instance,

- one can print all tasks with `starpu-print-all-tasks`,
- print all data with `starpu-print-datas`,

- print all pending data transfers with `starpu-print-prequests`, `starpu-print-requests`, `starpu-print-frequests`, `starpu-print-irequests`,
- print pending MPI requests with `starpu-mpi-print-detached-requests`

Some functions can only work if --enable-debug was passed to `configure` (because they impact performance)

22.3 Using Other Debugging Tools

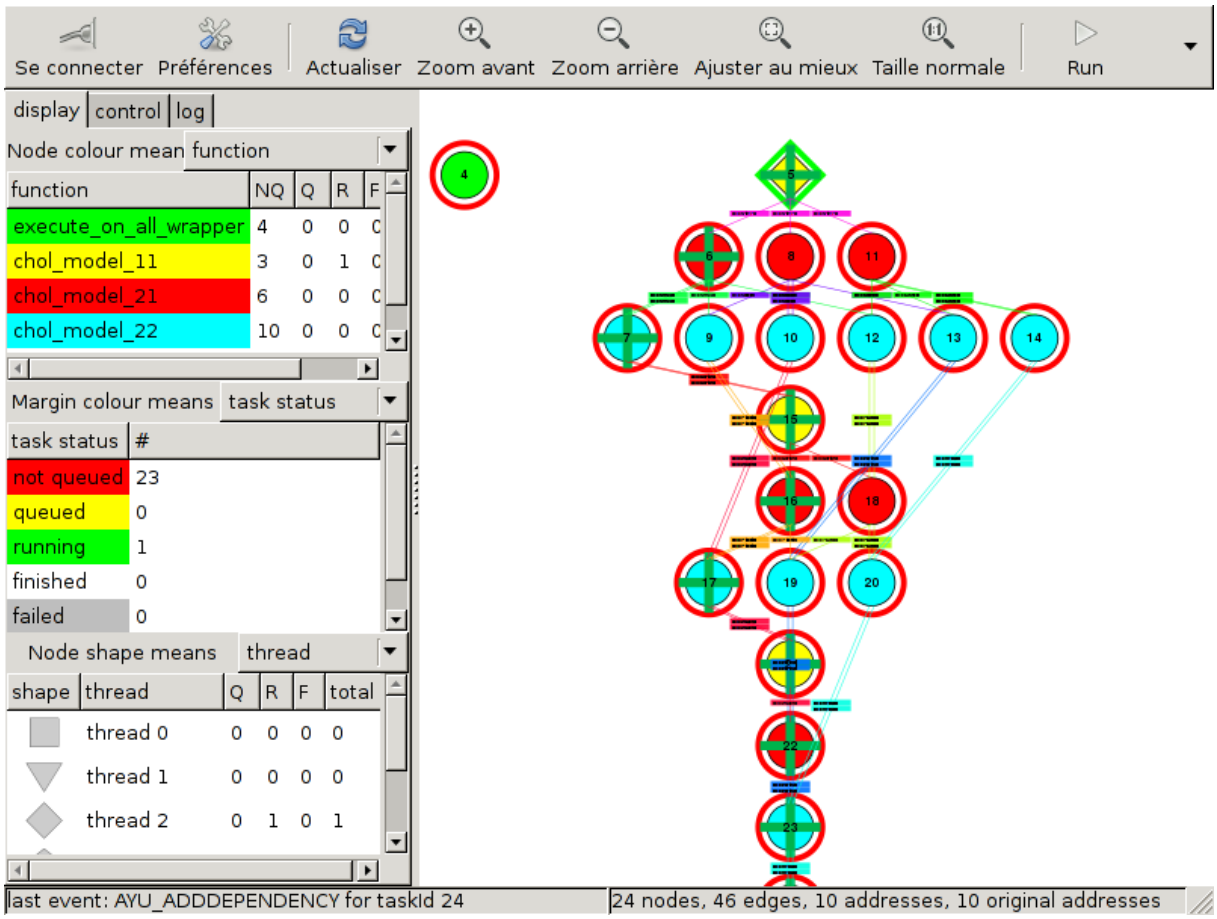
Valgrind can be used on StarPU: valgrind.h just needs to be found at `configure` time, to tell valgrind about some known false positives and disable host memory pinning. Other known false positives can be suppressed by giving the suppression files in `tools/valgrind/*.suppr` to valgrind's `-suppressions` option.
The environment variable STARPU_DISABLE_KERNELS can also be set to `1` to make StarPU does 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.

22.4 Watchdog Support

starpu_task_watchdog_set_hook() is used to set a callback function "watchdog hook" that will be called when there is no task completed during an expected time. The purpose of the watchdog hook is to allow the application to get the state for debugging.

22.5 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`.
It permits to visualize the task graph, add breakpoints, continue execution task-by-task, and run `gdb` on a given task, etc.



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 (to distinguish them from tasks).

Chapter 23

Helpers

StarPU provides several utilities functions to help programmers:

- starpu_conf_noworker() sets configuration fields so that no worker is enabled, i.e. it sets starpu_conf::ncpus to 0, starpu_conf::ncuda to 0, etc.
- starpu_is_initialized() returns a value indicating whether StarPU is already initialized, starpu_wait_initialized() only returns when the initialization is finished.
- starpu_topology_print() prints the current topology of the system, and is therefore useful for debugging purposes or for understanding the underlying architecture of the system.
- starpu_get_version() returns the version of StarPU used when running the application.
- starpu_sleep() and starpu_usleep() allow the application to pause the execution of the current thread for a specified amount of time. starpu_sleep() pauses the thread for a specified number of seconds and starpu_usleep() for a specified number of microseconds.

Part I

Appendix

Chapter 24

The GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright

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 EntitledAcknowledgements", and any sections Entitled `Dedications''. You must delete all sections EntitledEndorsements."

7. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ``aggregate'' if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled `Acknowledgements'', Dedications", or ``History'', the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

    You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

    However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

    Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

    Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

11. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License `‘or any later version’’ applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

12. RELICENSING

    `Massive Multiauthor Collaboration Site'' (orMMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A `Massive Multiauthor Collaboration'' (orMMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    `‘CC-BY-SA’’ means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    `‘Incorporate’’ means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is `‘eligible for relicensing’’ if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

24.1 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.