StarPU Internal Handbook

for StarPU 1.3.2

Generated by Doxygen.

This manual documents the internal usage of StarPU version 1.3.2. Its contents was last updated on 14 June 2019.

Contents

Chapter 1

Introduction

1.1 Motivation

Chapter 2

StarPU Core

2.1 StarPU Core Entities

TODO

2.1.1 Overview

Execution entities:

- **worker**: A worker (see Workers, Workers and Scheduling Contexts) entity is a CPU thread created by StarPU to manage one computing unit. The computing unit can be a local CPU core, an accelerator or GPU device, or — on the master side when running in master-slave distributed mode — a remote slave computing node. It is responsible for querying scheduling policies for tasks to execute.

- **sched_context**: A scheduling context (see Scheduling Contexts, Workers and Scheduling Contexts) is a logical set of workers governed by an instance of a scheduling policy. It defines the computing units to which the scheduling policy instance may assign work entities.

- **driver**: A driver is the set of hardware-dependent routines used by a worker to initialize its associated computing unit, execute work entities on it, and finalize the computing unit usage at the end of the session.

Work entities:

- **task**: A task is a high level work request submitted to StarPU by the application, or internally by StarPU itself.

- **job**: A job is a low level view of a work request. It is not exposed to the application. A job structure may be shared among several task structures in the case of a parallel task.

Data entities:

- **data handle**: A data handle is a high-level, application opaque object designating a piece of data currently registered to the StarPU data management layer. Internally, it is a _starpu_data_state structure.

- **data replicate**: A data replicate is a low-level object designating one copy of a piece of data registered to StarPU as a data handle, residing in one memory node managed by StarPU. It is not exposed to the application.

2.1.2 Workers

A **worker** is a CPU thread created by StarPU. Its role is to manage one computing unit. This computing unit can be a local CPU core, in which case, the worker thread manages the actual CPU core to which it is assigned; or it can be a computing device such as a GPU or an accelerator (or even a remote computing node when StarPU is running in distributed master-slave mode.) When a worker manages a computing device, the CPU core to which the worker's thread is by default exclusively assigned to the device management work and does not participate to computation.

2.1.2.1 States

Scheduling operations related state
While a worker is conducting a scheduling operations, e.g. the worker is in the process of selecting a new task to execute, flag state_sched_op_pending is set to `!0`, otherwise it is set to `0`.
While state_sched_op_pending is !0, the following exhaustive list of operations on that workers are restricted in the stated way:

- adding the worker to a context is not allowed;
- removing the worker from a context is not allowed;
- adding the worker to a parallel task team is not allowed;
- removing the worker from a parallel task team is not allowed;
- querying state information about the worker is only allowed while `state_relax_refcnt > 0`;
  - in particular, querying whether the worker is blocked on a parallel team entry is only allowed while `state_relax_refcnt > 0`.

Entering and leaving the state_sched_op_pending state is done through calls to _starpu_worker_enter_sched_op() and _starpu_worker_leave_sched_op() respectively (see these functions in use in functions _starpu_get_worker↩_task() and _starpu_get_multi_worker_task()). These calls ensure that any pending conflicting operation deferred while the worker was in the state_sched_op_pending state is performed in an orderly manner.

Scheduling contexts related states
Flag `state_changing_ctx_notice` is set to `!0` when a thread is about to add the worker to a scheduling context or remove it from a scheduling context, and is currently waiting for a safe window to do so, until the targeted worker is not in a scheduling operation or parallel task operation anymore. This flag set to `!0` will also prevent the targeted worker to attempt a fresh scheduling operation or parallel task operation to avoid starving conditions. However, a scheduling operation that was already in progress before the notice is allowed to complete.
Flag `state_changing_ctx_waiting` is set to `!0` when a scheduling context worker addition or removal involving the targeted worker is about to occur and the worker is currently performing a scheduling operation to tell the targeted worker that the initiator thread is waiting for the scheduling operation to complete and should be woken up upon completion.

Relaxed synchronization related states
Any StarPU worker may participate to scheduling operations, and in this process, may be forced to observe state information from other workers. A StarPU worker thread may therefore be observed by any thread, even other StarPU workers. Since workers may observe each other in any order, it is not possible to rely exclusively on the `sched_mutex` of each worker to protect the observation of worker state flags by other workers, because worker A observing worker B would involve locking workers in (A B) sequence, while worker B observing worker A would involve locking workers in (B A) sequence, leading to lock inversion deadlocks.
In consequence, no thread must hold more than one worker's sched_mutex at any time. Instead, workers implement a relaxed locking scheme based on the `state_relax_refcnt` counter, itself protected by the worker's sched↩_mutex. When `state_relax_refcnt > 0`, the targeted worker state flags may be observed, otherwise the thread attempting the observation must repeatedly wait on the targeted worker's `sched_cond` condition until `state_relax_refcnt > 0`.
The relaxed mode, while on, can actually be seen as a transactional consistency model, where concurrent accesses are authorized and potential conflicts are resolved after the fact. When the relaxed mode is off, the consistency model becomes a mutual exclusion model, where the sched_mutex of the worker must be held in order to access or change the worker state.

Parallel tasks related states
When a worker is scheduled to participate to the execution of a parallel task, it must wait for the whole team of workers participating to the execution of this task to be ready. While the worker waits for its teammates, it is not available to run other tasks or perform other operations. Such a waiting operation can therefore not start while conflicting operations such as scheduling operations and scheduling context resizing involving the worker are on-going. Conversely these operations and other may query weather the worker is blocked on a parallel task entry with starpu_worker_is_blocked_in_parallel().

The starpu_worker_is_blocked_in_parallel() function is allowed to proceed while and only while state_relax↩ _refcnt > 0. Due to the relaxed worker locking scheme, the state_blocked_in_parallel flag of the targeted worker may change after it has been observed by an observer thread. In consequence, flag state_↩ blocked_in_parallel_observed of the targeted worker is set to 1 by the observer immediately after the observation to "taint" the targeted worker. The targeted worker will clear the state_blocked_in_parallel↩ _observed flag tainting and defer the processing of parallel task related requests until a full scheduling operation shot completes without the state_blocked_in_parallel_observed flag being tainted again. The purpose of this tainting flag is to prevent parallel task operations to be started immediately after the observation of a transient scheduling state.
Worker's management of parallel tasks is governed by the following set of state flags and counters:

- state_blocked_in_parallel: set to !0 while the worker is currently blocked on a parallel task;
- state_blocked_in_parallel_observed: set to !0 to taint the worker when a thread has observed the state_blocked_in_parallel flag of this worker while its state_relax_refcnt state counter was >0. Any pending request to add or remove the worker from a parallel task team will be deferred until a whole scheduling operation shot completes without being tainted again.
- state_block_in_parallel_req: set to !0 when a thread is waiting on a request for the worker to be added to a parallel task team. Must be protected by the worker's sched_mutex.
- state_block_in_parallel_ack: set to !0 by the worker when acknowledging a request for being added to a parallel task team. Must be protected by the worker's sched_mutex.
- state_unblock_in_parallel_req: set to !0 when a thread is waiting on a request for the worker to be removed from a parallel task team. Must be protected by the worker's sched_mutex.
- state_unblock_in_parallel_ack: set to !0 by the worker when acknowledging a request for being removed from a parallel task team. Must be protected by the worker's sched_mutex.
- block_in_parallel_ref_count: counts the number of consecutive pending requests to enter parallel task teams. Only the first of a train of requests for entering parallel task teams triggers the transition of the state_block_in_parallel_req flag from 0 to 1. Only the last of a train of requests to leave a parallel task team triggers the transition of flag state_unblock_in_parallel_req from 0 to 1. Must be protected by the worker's sched_mutex.

2.1.2.2 Operations

Entry point
All the operations of a worker are handled in an iterative fashion, either by the application code on a thread launched by the application, or automatically by StarPU on a device-dependent CPU thread launched by StarPU. Whether a worker's operation cycle is managed automatically or not is controlled per session by the field not_launched↩ _drivers of the starpu_conf struct, and is decided in _starpu_launch_drivers() function.
When managed automatically, cycles of operations for a worker are handled by the corresponding driver specific _starpu_<DRV>_worker() function, where DRV is a driver name such as cpu (_starpu_cpu_worker) or cuda (_starpu_cuda_worker), for instance. Otherwise, the application must supply a thread which will repeatedly call starpu_driver_run_once() for the corresponding worker.
In both cases, control is then transferred to _starpu_cpu_driver_run_once() (or the corresponding driver specific func). The cycle of operations typically includes, at least, the following operations:

- **task scheduling**
- **parallel task team build-up**
- **task input processing**
- **data transfer processing**
- **task execution**

When the worker cycles are handled by StarPU automatically, the iterative operation processing ends when the running field of _starpu_config becomes false. This field should not be read directly, instead it should be read through the _starpu_machine_is_running() function.

Task scheduling
If the worker does not yet have a queued task, it calls _starpu_get_worker_task() to try and obtain a task. This may involve scheduling operations such as stealing a queued but not yet executed task from another worker. The operation may not necessarily succeed if no tasks are ready and/or suitable to run on the worker's computing unit.

Parallel task team build-up
If the worker has a task ready to run and the corresponding job has a size >1, then the task is a parallel job and the worker must synchronize with the other workers participating to the parallel execution of the job to assign a unique rank for each worker. The synchronization is done through the job's `sync_mutex` mutex.

Task input processing
Before the task can be executed, its input data must be made available on a memory node reachable by the worker's computing unit. To do so, the worker calls _starpu_fetch_task_input()

Data transfer processing
The worker makes pending data transfers (involving memory node(s) that it is driving) progress, with a call to __↩ starpu_datawizard_progress(),

Task execution
Once the worker has a pending task assigned and the input data for that task are available in the memory node reachable by the worker's computing unit, the worker calls _starpu_cpu_driver_execute_task() (or the corresponding driver specific function) to proceed to the execution of the task.

2.1.3 Scheduling Contexts

A scheduling context is a logical set of workers governed by an instance of a scheduling policy. Tasks submitted to a given scheduling context are confined to the computing units governed by the workers belonging to this scheduling context at the time they get scheduled.
A scheduling context is identified by an unsigned integer identifier between `0` and `STARPU_NMAX_SCHED_CT↩ XS - 1`. The `STARPU_NMAX_SCHED_CTXS` identifier value is reserved to indicated an unallocated, invalid or deleted scheduling context.
Accesses to the scheduling context structure are governed by a multiple-readers/single-writer lock (`rwlock` field). Changes to the structure contents, additions or removals of workers, statistics updates, all must be done with proper exclusive write access.

2.1.4 Workers and Scheduling Contexts

A worker can be assigned to one or more **scheduling contexts**. It exclusively receives tasks submitted to the scheduling context(s) it is currently assigned at the time such tasks are scheduled. A worker may add itself to or remove itself from a scheduling context.

Locking and synchronization rules between workers and scheduling contexts
A thread currently holding a worker sched_mutex must not attempt to acquire a scheduling context rwlock, neither for writing nor for reading. Such an attempt constitutes a lock inversion and may result in a deadlock.
A worker currently in a scheduling operation must enter the relaxed state before attempting to acquire a scheduling context rwlock, either for reading or for writing.
When the set of workers assigned to a scheduling context is about to be modified, all the workers in the union between the workers belonging to the scheduling context before the change and the workers expected to belong to the scheduling context after the change must be notified using the notify_workers_about_changing_ctx_pending() function prior to the update. After the update, all the workers in that same union must be notified for the update completion with a call to notify_workers_about_changing_ctx_done().
The function notify_workers_about_changing_ctx_pending() places every worker passed in argument in a state compatible with changing the scheduling context assignment of that worker, possibly blocking until that worker leaves incompatible states such as a pending scheduling operation. If the caller of `notify_workers_about↩ _changing_ctx_pending()` is itself a worker included in the set of workers passed in argument, it does not notify itself, with the assumption that the worker is already calling `notify_workers_about_changing↩ _ctx_pending()` from a state compatible with a scheduling context assignment update. Once a worker has

been notified about a scheduling context change pending, it cannot proceed with incompatible operations such as a scheduling operation until it receives a notification that the context update operation is complete.

2.1.5 Drivers

Each driver defines a set of routines depending on some specific hardware. These routines include hardware discovery/initialization, task execution, device memory management and data transfers.
While most hardware dependent routines are in source files located in the `/src/drivers` subdirectory of the StarPU tree, some can be found elsewhere in the tree such as `src/datawizard/malloc.c` for memory allocation routines or the subdirectories of `src/datawizard/interfaces/` for data transfer routines.
The driver ABI defined in the _starpu_driver_ops structure includes the following operations:

- .init: initialize a driver instance for the calling worker managing a hardware computing unit compatible with this driver.
- .run_once: perform a single driver progress cycle for the calling worker (see Operations).
- .deinit: deinitialize the driver instance for the calling worker
- .run: executes the following sequence automatically: call .init, repeatedly call .run_once until the function _starpu_machine_is_running() returns false, call .deinit.

The source code common to all drivers is shared in `src/drivers/driver_common/driver_↩common.[ch]`. This file includes services such as grabbing a new task to execute on a worker, managing statistics accounting on job startup and completion and updating the worker status

2.1.5.1 Master/Slave Drivers

A subset of the drivers corresponds to drivers managing computing units in master/slave mode, that is, drivers involving a local master instance managing one or more remote slave instances on the targeted device(s). This includes devices such as discrete manycore accelerators (e.g. Intel's Knight Corners board, for instance), or pseudo devices such as a cluster of cpu nodes driver through StarPU's MPI master/slave mode. A driver instance on the master side is named the **source**, while a driver instances on the slave side is named the **sink**.
A significant part of the work realized on the source and sink sides of master/slave drivers is identical among all master/slave drivers, due to the similarities in the software pattern. Therefore, many routines are shared among all these drivers in the `src/drivers/mp_common` subdirectory. In particular, a set of default commands to be used between sources and sinks is defined, assuming the availability of some communication channel between them (see enum _starpu_mp_command)
TODO

2.1.6 Tasks and Jobs

TODO

2.1.7 Data

TODO

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Module Documentation

4.1 Workers

Data Structures

- struct _starpu_worker
- struct _starpu_combined_worker
- struct _starpu_worker_set
- struct _starpu_machine_topology
- struct _starpu_machine_config
- struct _starpu_machine_config.bindid_workers

Macros

- #define **STARPU_MAX_PIPELINE**
- #define **starpu_worker_get_count**
- #define **starpu_worker_get_id**
- #define **_starpu_worker_get_id_check**(f, l)
- #define **starpu_worker_relax_on**
- #define **starpu_worker_relax_off**
- #define **starpu_worker_get_relax_state**

Enumerations

- enum **initialization** { **UNINITIALIZED**, **CHANGING**, **INITIALIZED** }

Functions

- void _starpu_set_argc_argv (int *argc, char ***argv)
- int * **_starpu_get_argc** ()
- char *** **_starpu_get_argv** ()
- void _starpu_conf_check_environment (struct starpu_conf *conf)
- void _starpu_may_pause (void)
- static unsigned _starpu_machine_is_running (void)
- void _starpu_worker_init (struct _starpu_worker *workerarg, struct _starpu_machine_config *pconfig)
- uint32_t _starpu_worker_exists (struct starpu_task *)
- uint32_t _starpu_can_submit_cuda_task (void)
- uint32_t _starpu_can_submit_cpu_task (void)
- uint32_t _starpu_can_submit_opencl_task (void)
- unsigned _starpu_worker_can_block (unsigned memnode, struct _starpu_worker *worker)
- void _starpu_block_worker (int workerid, starpu_pthread_cond_t *cond, starpu_pthread_mutex_t *mutex)
- void _starpu_driver_start (struct _starpu_worker *worker, unsigned fut_key, unsigned sync)
- void _starpu_worker_start (struct _starpu_worker *worker, unsigned fut_key, unsigned sync)

- static unsigned **_starpu_worker_get_count** (void)
- static void _starpu_set_local_worker_key (struct _starpu_worker *worker)
- static struct _starpu_worker * _starpu_get_local_worker_key (void)
- static void _starpu_set_local_worker_set_key (struct _starpu_worker_set *worker)
- static struct _starpu_worker_set * _starpu_get_local_worker_set_key (void)
- static struct _starpu_worker * _starpu_get_worker_struct (unsigned id)
- static struct _starpu_sched_ctx * _starpu_get_sched_ctx_struct (unsigned id)
- struct _starpu_combined_worker * **_starpu_get_combined_worker_struct** (unsigned id)
- static struct _starpu_machine_config * _starpu_get_machine_config (void)
- static int _starpu_get_disable_kernels (void)
- static enum _starpu_worker_status _starpu_worker_get_status (int workerid)
- static void _starpu_worker_set_status (int workerid, enum _starpu_worker_status status)
- static struct _starpu_sched_ctx * _starpu_get_initial_sched_ctx (void)
- int **starpu_worker_get_nids_by_type** (enum starpu_worker_archtype type, int *workerids, int maxsize)
- int starpu_worker_get_nids_ctx_free_by_type (enum starpu_worker_archtype type, int *workerids, int maxsize)
- static unsigned **_starpu_worker_mutex_is_sched_mutex** (int workerid, starpu_pthread_mutex_t *mutex)
- static int **_starpu_worker_get_nsched_ctxs** (int workerid)
- static unsigned _starpu_get_nsched_ctxs (void)
- static int _starpu_worker_get_id (void)
- static unsigned __starpu_worker_get_id_check (const char *f, int l)
- enum starpu_node_kind **_starpu_worker_get_node_kind** (enum starpu_worker_archtype type)
- void **_starpu_worker_set_stream_ctx** (unsigned workerid, struct _starpu_sched_ctx *sched_ctx)
- struct _starpu_sched_ctx * **_starpu_worker_get_ctx_stream** (unsigned stream_workerid)
- static void _starpu_worker_request_blocking_in_parallel (struct _starpu_worker *const worker)
- static void _starpu_worker_request_unblocking_in_parallel (struct _starpu_worker *const worker)
- static void _starpu_worker_process_block_in_parallel_requests (struct _starpu_worker *const worker)
- static void _starpu_worker_enter_sched_op (struct _starpu_worker *const worker)
- void _starpu_worker_apply_deferred_ctx_changes (void)
- static void **_starpu_worker_leave_sched_op** (struct _starpu_worker *const worker)
- static int **_starpu_worker_sched_op_pending** (void)
- static void _starpu_worker_enter_changing_ctx_op (struct _starpu_worker *const worker)
- static void _starpu_worker_leave_changing_ctx_op (struct _starpu_worker *const worker)
- static void _starpu_worker_relax_on (void)
- static void _starpu_worker_relax_on_locked (struct _starpu_worker *worker)
- static void **_starpu_worker_relax_off** (void)
- static void **_starpu_worker_relax_off_locked** (void)
- static int **_starpu_worker_get_relax_state** (void)
- static void _starpu_worker_lock (int workerid)
- static int **_starpu_worker_trylock** (int workerid)
- static void **_starpu_worker_unlock** (int workerid)
- static void **_starpu_worker_lock_self** (void)
- static void **_starpu_worker_unlock_self** (void)
- static int **_starpu_wake_worker_relax** (int workerid)
- int **starpu_wake_worker_relax_light** (int workerid)
- void _starpu_worker_refuse_task (struct _starpu_worker *worker, struct starpu_task *task)

Variables

- int **_starpu_worker_parallel_blocks**
- struct _starpu_machine_config _starpu_config **STARPU_ATTRIBUTE_INTERNAL**

4.1.1 Detailed Description

4.1.2 Data Structure Documentation

4.1.2.1 struct _starpu_worker

This is initialized by _starpu_worker_init()

Data Fields

struct _starpu_machine_config *	config	
starpu_pthread_mutex_t	mutex	
enum starpu_worker_archtype	arch	what is the type of worker ?
uint32_t	worker_mask	what is the type of worker ?
struct starpu_perfmodel_arch	perf_arch	in case there are different models of the same arch
starpu_pthread_t	worker_thread	the thread which runs the worker
unsigned	devid	which cpu/gpu/etc is controlled by the worker ?
unsigned	subworkerid	which sub-worker this one is for the cpu/gpu
int	bindid	which cpu is the driver bound to ? (logical index)
int	workerid	uniquely identify the worker among all processing units types
int	combined_workerid	combined worker currently using this worker
int	current_rank	current rank in case the worker is used in a parallel fashion
int	worker_size	size of the worker in case we use a combined worker
starpu_pthread_cond_t	started_cond	indicate when the worker is ready
starpu_pthread_cond_t	ready_cond	indicate when the worker is ready
unsigned	memory_node	which memory node is the worker associated with ?
unsigned	numa_memory_node	which numa memory node is the worker associated with? (logical index)
starpu_pthread_cond_t	sched_cond	condition variable used for passive waiting operations on worker STARPU_PTHREAD_COND_B↩ ROADCAST must be used instead of STARPU_PTHREAD_COND↩ _SIGNAL, since the condition is shared for multiple purpose
starpu_pthread_mutex_t	sched_mutex	mutex protecting sched_cond
unsigned	state_relax_refcnt	mark scheduling sections where other workers can safely access the worker state
unsigned	state_sched_op_pending	a task pop is ongoing even though sched_mutex may temporarily be unlocked
unsigned	state_changing_ctx_waiting	a thread is waiting for operations such as pop to complete before acquiring sched_mutex and modifying the worker ctx
unsigned	state_changing_ctx_notice	the worker ctx is about to change or being changed, wait for flag to be cleared before starting new scheduling operations
unsigned	state_blocked_in_parallel	worker is currently blocked on a parallel section

Data Fields

unsigned	state_blocked_in_parallel_observed	the blocked state of the worker has been observed by another worker during a relaxed section
unsigned	state_block_in_parallel_req	a request for state transition from unblocked to blocked is pending
unsigned	state_block_in_parallel_ack	a block request has been honored
unsigned	state_unblock_in_parallel_req	a request for state transition from blocked to unblocked is pending
unsigned	state_unblock_in_parallel_ack	an unblock request has been honored
unsigned	block_in_parallel_ref_count	cumulative blocking depth • =0 worker unblocked • $>$0 worker blocked • transition from 0 to 1 triggers a block_req • transition from 1 to 0 triggers a unblock_req
starpu_pthread_t	thread_changing_ctx	thread currently changing a sched_ctx containing the worker
struct _starpu_ctx_change_list	ctx_change_list	list of deferred context changes when the current thread is a worker, _and_ this worker is in a scheduling operation, new ctx changes are queued to this list for subsequent processing once worker completes the ongoing scheduling operation
struct starpu_task_list	local_tasks	this queue contains tasks that have been explicitely submitted to that queue
struct starpu_task $**$	local_ordered_tasks	this queue contains tasks that have been explicitely submitted to that queue with an explicit order
unsigned	local_ordered_tasks_size	this records the size of local_ordered_tasks
unsigned	current_ordered_task	this records the index (within local_ordered_tasks) of the next ordered task to be executed
unsigned	current_ordered_task_order	this records the order of the next ordered task to be executed
struct starpu_task $*$	current_task	task currently executed by this worker (non-pipelined version)
struct starpu_task $*$	current_tasks[STARPU_MAX_PIPELINES]	tasks currently executed by this worker (pipelined version)
starpu_pthread_wait_t	wait	
struct timespec	cl_start	Codelet start time of the task currently running
struct timespec	cl_end	Codelet end time of the last task running
unsigned char	first_task	Index of first task in the pipeline

Data Fields

unsigned char	ntasks	number of tasks in the pipeline
unsigned char	pipeline_length	number of tasks to be put in the pipeline
unsigned char	pipeline_stuck	whether a task prevents us from pipelining
struct _starpu_worker_set *	set	in case this worker belongs to a set
unsigned	worker_is_running	
unsigned	worker_is_initialized	
enum _starpu_worker_status	status	what is the worker doing now ? (eg. CALLBACK)
unsigned	state_keep_awake	!0 if a task has been pushed to the worker and the task has not yet been seen by the worker, the worker should no go to sleep before processing this task
char	name[128]	
char	short_name[32]	
unsigned	run_by_starpu	Is this run by StarPU or directly by the application ?
struct _starpu_driver_ops *	driver_ops	
struct _starpu_sched_ctx_list *	sched_ctx_list	
int	tmp_sched_ctx	
unsigned	nsched_ctxs	the no of contexts a worker belongs to
struct _starpu_barrier_counter	tasks_barrier	wait for the tasks submitted
unsigned	has_prev_init	had already been inited in another ctx
unsigned	removed_from_ctx[STARPU_NMAX_SCHED_CTXS+1]	
unsigned	spinning_backoff	number of cycles to pause when spinning
unsigned	nb_buffers_transferred	number of piece of data already send to worker
unsigned	nb_buffers_totransfer	number of piece of data already send to worker
struct starpu_task *	task_transferring	The buffers of this task are being sent
unsigned	shares_tasks_lists[STARPU_NMAX_SCHED_CTXS+1]	boolean whether the workers shares tasks lists with other workers in this case when removing him from a context it disapears instantly
unsigned	poped_in_ctx[STARPU_NMAX_SCHED_CTXS+1]	boolean to chose the next ctx a worker will pop into
unsigned	reverse_phase[2]	boolean indicating at which moment we checked all ctxs and change phase for the booleab poped_in_ctx one for each of the 2 priorities
unsigned	pop_ctx_priority	indicate which priority of ctx is currently active: the values are 0 or 1

Data Fields

unsigned	is_slave_somewhere	bool to indicate if the worker is slave in a ctx
struct _starpu_sched_ctx *	stream_ctx	
hwloc_bitmap_t	hwloc_cpu_set	
hwloc_obj_t	hwloc_obj	

4.1.2.2 struct _starpu_combined_worker

Data Fields

struct starpu_perfmodel_arch	perf_arch	in case there are different models of the same arch
uint32_t	worker_mask	what is the type of workers ?
int	worker_size	
unsigned	memory_node	which memory node is associated that worker to ?
int	combined_workerid[STARPU_NMAXWORKERS]	
hwloc_bitmap_t	hwloc_cpu_set	

4.1.2.3 struct _starpu_worker_set

in case a single CPU worker may control multiple accelerators

Data Fields

starpu_pthread_mutex_t	mutex	
starpu_pthread_t	worker_thread	the thread which runs the worker
unsigned	nworkers	
unsigned	started	Only one thread for the whole set
void *	retval	
struct _starpu_worker *	workers	
starpu_pthread_cond_t	ready_cond	indicate when the set is ready
unsigned	set_is_initialized	

4.1.2.4 struct _starpu_machine_topology

Data Fields

unsigned	nworkers	Total number of workers.
unsigned	ncombinedworkers	Total number of combined workers.
unsigned	nsched_ctxs	
hwloc_topology_t	hwtopology	Topology as detected by hwloc.
struct starpu_tree *	tree	custom hwloc tree
unsigned	nhwcpus	Total number of CPU cores, as detected by the topology code. May be different from the actual number of CPU workers.

Data Fields

unsigned	nhwpus	Total number of PUs (i.e. threads), as detected by the topology code. May be different from the actual number of PU workers.
unsigned	nhwcudagpus	Total number of CUDA devices, as detected. May be different from the actual number of CUDA workers.
unsigned	nhwopenclgpus	Total number of OpenCL devices, as detected. May be different from the actual number of OpenCL workers.
unsigned	nhwmpi	Total number of MPI nodes, as detected. May be different from the actual number of node workers.
unsigned	ncpus	Actual number of CPU workers used by StarPU.
unsigned	ncudagpus	Actual number of CUDA GPUs used by StarPU.
unsigned	nworkerpercuda	
int	cuda_th_per_stream	
int	cuda_th_per_dev	
unsigned	nopenclgpus	Actual number of OpenCL workers used by StarPU.
unsigned	nmpidevices	Actual number of MPI workers used by StarPU.
unsigned	nhwmpidevices	
unsigned	nhwmpicores[STARPU_MAXMPIDEVS]	Each MPI node has its set of cores.
unsigned	nmpicores[STARPU_MAXMPIDEVS]	
unsigned	nhwmicdevices	Topology of MP nodes (MIC) as well as necessary objects to communicate with them.
unsigned	nmicdevices	
unsigned	nhwmiccores[STARPU_MAXMICDEVS]	Each MIC node has its set of cores.
unsigned	nmiccores[STARPU_MAXMICDEVS]	
unsigned	workers_bindid[STARPU_NMAXWORKERS]	Indicates the successive logical PU identifier that should be used to bind the workers. It is either filled according to the user's explicit parameters (from starpu_conf) or according to the STARPU_WORKERS_CPUID env. variable. Otherwise, a round-robin policy is used to distributed the workers over the cores.
unsigned	workers_cuda_gpuid[STARPU_NMAXWORKERS]	Indicates the successive CUDA identifier that should be used by the CUDA driver. It is either filled according to the user's explicit parameters (from starpu_conf) or according to the STARPU_WORKERS_CUDAID env. variable. Otherwise, they are taken in ID order.

Data Fields

unsigned	workers_opencl_gpuid[STARPU_NMAXWORKERS]	Indicates the successive OpenCL identifier that should be used by the OpenCL driver. It is either filled according to the user's explicit parameters (from starpu_conf) or according to the STARPU_WORKERS_OPENCLID env. variable. Otherwise, they are taken in ID order.
unsigned	workers_mpi_ms_deviceid[STARPU_NMAXWORKERS]	Indicates the successive unsigned workers_mic_↩ deviceid[STARPU_NMAXWORKERS];

4.1.2.5 struct _starpu_machine_config

Data Fields

struct _starpu_machine_topology	topology	
int	cpu_depth	
int	pu_depth	
int	current_bindid	Where to bind next worker ?
char	currently_bound[STARPU_NMAXWORKERS]	
char	currently_shared[STARPU_NMAXWORKERS]	
int	current_cuda_gpuid	Which GPU(s) do we use for CUDA ?
int	current_opencl_gpuid	Which GPU(s) do we use for OpenCL ?
int	current_mic_deviceid	Which MIC do we use?
int	current_mpi_deviceid	Which MPI do we use?
int	cpus_nodeid	Memory node for cpus, if only one
int	cuda_nodeid	Memory node for CUDA, if only one
int	opencl_nodeid	Memory node for OpenCL, if only one
int	mic_nodeid	Memory node for MIC, if only one
int	mpi_nodeid	Memory node for MPI, if only one
struct _starpu_worker	workers[STARPU_NMAXWORKERS]	Basic workers : each of this worker is running its own driver and can be combined with other basic workers.
struct _starpu_combined_worker	combined_workers[STARPU_NMAX_COMBINEDWORKERS]	Combined workers : this worker are a combination of basic workers that can run parallel tasks together.
struct _starpu_machine_config	bindid_workers	Translation table from bindid to worker IDs
unsigned	nbindid	size of bindid_workers
uint32_t	worker_mask	This bitmask indicates which kinds of worker are available. For instance it is possible to test if there is a CUDA worker with the result of (worker_mask & STARPU_CUDA).
struct starpu_conf	conf	either the user given configuration passed to starpu_init or a default configuration

Data Fields

unsigned	running	this flag is set until the runtime is stopped
int	disable_kernels	
int	pause_depth	Number of calls to starpu_pause() - calls to starpu_resume(). When >0, StarPU should pause.
struct _starpu_sched_ctx	sched_ctxs[STARPU_NMAX_SCHED_CTXS+1]	The scheduled ctx of the current instance of starpu
unsigned	submitting	this flag is set until the application is finished submitting tasks
int	watchdog_ok	
starpu_pthread_mutex_t	submitted_mutex	

4.1.2.6 struct _starpu_machine_config.bindid_workers

Translation table from bindid to worker IDs

Data Fields

int *	workerids	
unsigned	nworkers	size of workerids

4.1.3 Function Documentation

4.1.3.1 _starpu_set_argc_argv()

```
void _starpu_set_argc_argv (
            int * argc,
            char *** argv )
```

Three functions to manage argv, argc

4.1.3.2 _starpu_conf_check_environment()

```
void _starpu_conf_check_environment (
            struct starpu_conf * conf )
```

Fill conf with environment variables

4.1.3.3 _starpu_may_pause()

```
void _starpu_may_pause (
            void  )
```

Called by the driver when it is ready to pause

4.1.3.4 _starpu_machine_is_running()

```
static unsigned _starpu_machine_is_running (
            void  )  [inline], [static]
```

Has starpu_shutdown already been called ?

4.1.3.5 _starpu_worker_init()

```
void _starpu_worker_init (
            struct _starpu_worker * workerarg,
            struct _starpu_machine_config * pconfig )
```

initialise a worker

4.1.3.6 _starpu_worker_exists()

```
uint32_t _starpu_worker_exists (
            struct starpu_task *  )
```

Check if there is a worker that may execute the task.

4.1.3.7 _starpu_can_submit_cuda_task()

```
uint32_t _starpu_can_submit_cuda_task (
            void  )
```

Is there a worker that can execute CUDA code ?

4.1.3.8 _starpu_can_submit_cpu_task()

```
uint32_t _starpu_can_submit_cpu_task (
            void  )
```

Is there a worker that can execute CPU code ?

4.1.3.9 _starpu_can_submit_opencl_task()

```
uint32_t _starpu_can_submit_opencl_task (
            void  )
```

Is there a worker that can execute OpenCL code ?

4.1.3.10 _starpu_worker_can_block()

```
unsigned _starpu_worker_can_block (
            unsigned memnode,
            struct _starpu_worker * worker )
```

Check whether there is anything that the worker should do instead of sleeping (waiting on something to happen).

4.1.3.11 _starpu_block_worker()

```
void _starpu_block_worker (
            int workerid,
            starpu_pthread_cond_t * cond,
            starpu_pthread_mutex_t * mutex )
```

This function must be called to block a worker. It puts the worker in a sleeping state until there is some event that forces the worker to wake up.

4.1.3.12 _starpu_driver_start()

```
void _starpu_driver_start (
            struct _starpu_worker * worker,
            unsigned fut_key,
            unsigned sync )
```

This function initializes the current driver for the given worker

4.1.3.13 _starpu_worker_start()

```
void _starpu_worker_start (
            struct _starpu_worker * worker,
```

```
            unsigned fut_key,
            unsigned sync )
```
This function initializes the current thread for the given worker

4.1.3.14 _starpu_set_local_worker_key()

```
static void _starpu_set_local_worker_key (
            struct _starpu_worker * worker )  [inline], [static]
```
The _starpu_worker structure describes all the state of a StarPU worker. This function sets the pthread key which stores a pointer to this structure.

4.1.3.15 _starpu_get_local_worker_key()

```
static struct _starpu_worker* _starpu_get_local_worker_key (
            void  )  [static]
```
Returns the _starpu_worker structure that describes the state of the current worker.

4.1.3.16 _starpu_set_local_worker_set_key()

```
static void _starpu_set_local_worker_set_key (
            struct _starpu_worker_set * worker )  [inline], [static]
```
The _starpu_worker_set structure describes all the state of a StarPU worker_set. This function sets the pthread key which stores a pointer to this structure.

4.1.3.17 _starpu_get_local_worker_set_key()

```
static struct _starpu_worker_set* _starpu_get_local_worker_set_key (
            void  )  [static]
```
Returns the _starpu_worker_set structure that describes the state of the current worker_set.

4.1.3.18 _starpu_get_worker_struct()

```
static struct _starpu_worker* _starpu_get_worker_struct (
            unsigned id )  [static]
```
Returns the _starpu_worker structure that describes the state of the specified worker.

4.1.3.19 _starpu_get_sched_ctx_struct()

```
static struct _starpu_sched_ctx* _starpu_get_sched_ctx_struct (
            unsigned id )  [static]
```
Returns the starpu_sched_ctx structure that describes the state of the specified ctx

4.1.3.20 _starpu_get_machine_config()

```
static struct _starpu_machine_config* _starpu_get_machine_config (
            void  )  [static]
```
Returns the structure that describes the overall machine configuration (eg. all workers and topology).

4.1.3.21 _starpu_get_disable_kernels()

```
static int _starpu_get_disable_kernels (
            void  )  [inline], [static]
```
Return whether kernels should be run (<=0) or not (>0)

4.1.3.22 _starpu_worker_get_status()

```
static enum _starpu_worker_status _starpu_worker_get_status (
            int workerid )  [inline], [static]
```
Retrieve the status which indicates what the worker is currently doing.

4.1.3.23 _starpu_worker_set_status()

```
static void _starpu_worker_set_status (
            int workerid,
            enum _starpu_worker_status status )  [inline], [static]
```

Change the status of the worker which indicates what the worker is currently doing (eg. executing a callback).

4.1.3.24 _starpu_get_initial_sched_ctx()

```
static struct _starpu_sched_ctx* _starpu_get_initial_sched_ctx (
            void  )  [static]
```

We keep an initial sched ctx which might be used in case no other ctx is available

4.1.3.25 starpu_worker_get_nids_ctx_free_by_type()

```
int starpu_worker_get_nids_ctx_free_by_type (
            enum starpu_worker_archtype type,
            int * workerids,
            int maxsize )
```

returns workers not belonging to any context, be careful no mutex is used, the list might not be updated

4.1.3.26 _starpu_get_nsched_ctxs()

```
static unsigned _starpu_get_nsched_ctxs (
            void  )  [inline], [static]
```

Get the total number of sched_ctxs created till now

4.1.3.27 _starpu_worker_get_id()

```
static int _starpu_worker_get_id (
            void  )  [inline], [static]
```

Inlined version when building the core.

4.1.3.28 __starpu_worker_get_id_check()

```
static unsigned __starpu_worker_get_id_check (
            const char * f,
            int l )  [inline], [static]
```

Similar behaviour to starpu_worker_get_id() but fails when called from outside a worker This returns an unsigned object on purpose, so that the caller is sure to get a positive value

4.1.3.29 _starpu_worker_request_blocking_in_parallel()

```
static void _starpu_worker_request_blocking_in_parallel (
            struct _starpu_worker *const worker )  [inline], [static]
```

Send a request to the worker to block, before a parallel task is about to begin.
Must be called with worker's sched_mutex held.

4.1.3.30 _starpu_worker_request_unblocking_in_parallel()

```
static void _starpu_worker_request_unblocking_in_parallel (
            struct _starpu_worker *const worker )  [inline], [static]
```

Send a request to the worker to unblock, after a parallel task is complete.
Must be called with worker's sched_mutex held.

4.1.3.31 _starpu_worker_process_block_in_parallel_requests()

```
static void _starpu_worker_process_block_in_parallel_requests (
            struct _starpu_worker *const worker )  [inline], [static]
```

Called by the the worker to process incoming requests to block or unblock on parallel task boundaries.

Must be called with worker's sched_mutex held.

4.1.3.32 _starpu_worker_enter_sched_op()

```
static void _starpu_worker_enter_sched_op (
            struct _starpu_worker *const worker )  [inline], [static]
```

Mark the beginning of a scheduling operation by the worker. No worker blocking operations on parallel tasks and no scheduling context change operations must be performed on contexts containing the worker, on contexts about to add the worker and on contexts about to remove the worker, while the scheduling operation is in process. The sched mutex of the worker may only be acquired permanently by another thread when no scheduling operation is in process, or when a scheduling operation is in process _and_ worker->state_relax_refcnt!=0. If a scheduling operation is in process _and_ worker->state_relax_refcnt==0, a thread other than the worker must wait on condition worker->sched_cond for worker->state_relax_refcnt!=0 to become true, before acquiring the worker sched mutex permanently.
Must be called with worker's sched_mutex held.

4.1.3.33 _starpu_worker_apply_deferred_ctx_changes()

```
void _starpu_worker_apply_deferred_ctx_changes (
            void  )
```

Mark the end of a scheduling operation by the worker.
Must be called with worker's sched_mutex held.

4.1.3.34 _starpu_worker_enter_changing_ctx_op()

```
static void _starpu_worker_enter_changing_ctx_op (
            struct _starpu_worker *const worker )  [inline], [static]
```

Must be called before altering a context related to the worker whether about adding the worker to a context, removing it from a context or modifying the set of workers of a context of which the worker is a member, to mark the beginning of a context change operation. The sched mutex of the worker must be held before calling this function.
Must be called with worker's sched_mutex held.

4.1.3.35 _starpu_worker_leave_changing_ctx_op()

```
static void _starpu_worker_leave_changing_ctx_op (
            struct _starpu_worker *const worker )  [inline], [static]
```

Mark the end of a context change operation.
Must be called with worker's sched_mutex held.

4.1.3.36 _starpu_worker_relax_on()

```
static void _starpu_worker_relax_on (
            void  )  [inline], [static]
```

Temporarily allow other worker to access current worker state, when still scheduling, but the scheduling has not yet been made or is already done

4.1.3.37 _starpu_worker_relax_on_locked()

```
static void _starpu_worker_relax_on_locked (
            struct _starpu_worker * worker )  [inline], [static]
```

Same, but with current worker mutex already held

4.1.3.38 _starpu_worker_lock()

```
static void _starpu_worker_lock (
            int workerid )  [inline], [static]
```

lock a worker for observing contents
notes:

- if the observed worker is not in state_relax_refcnt, the function block until the state is reached

4.1.3.39 _starpu_worker_refuse_task()

```
void _starpu_worker_refuse_task (
            struct _starpu_worker * worker,
            struct starpu_task * task )
```

Allow a worker pulling a task it cannot execute to properly refuse it and send it back to the scheduler.

Chapter 5

Index

Index