GLT (Generic Lightweight Thread)
Adrián Castelló (updated 02/18/2016)

This library joins several lightweight thread libraries into a single common API.

In order to achieve a good balance between the functionality offered by all the implementations, several approaches can be adopted:

Including just the functionality that matches in all the libraries.

- Pros:
  - Easy implementation (just coding)
  - Perfect for the library with the minimum functionality
- Cons:
  - Lot of functionality can be lost
  - It is unfair for the more complete libraries

Including all the functionality from all the libraries

- Pros:
  - All the functionality is offered to the programmer
  - All the libraries are fully represented
- Cons:
  - A high number of code lines
  - Functionality that is not available in one Library
    - Add the functionality to the libraries
    - Error message indicating the correct runtime compilation option

Including just the common functions and those which can be replaced with current features

- Pros:
  - Most of the overall functionality
  - Maintainable code
- Cons:
  - Specific library functions are discarded
  - The programmer can ask for a functionality and can receive one similar. (e.g. requesting a GLT_tasklet without using the Argobots implementation and receiving a GLT_ult)

A basic implementation can be found in https://github.com/adcastel/GLT.git with the third approach and composed by Argobots, MassiveThreads and Qthreads libraries. By now, ConverseThreads and Go are discarded because the their programming model does not match with the first three libraries ones.

In addition, two different implementations can be done. On the one hand, the common dynamic library approach that is linked and loaded in execution time. On the other hand a static inline based implementation that is expecte to be faster than the former but it should be compiled with the user application.

Own Structures

// ULT object

GLT_ult

Represents the ABT_Thread, myth_thread_t and aligned_t in Argobots, MassiveThreads and Qthreads respectively

// Tasklet Object

GLT_tasklet

Represents the ABT_Task, myth_thread_t and aligned_t in Argobots, MassiveThreads and Qthreads respectively

//Thread Object

GLT_thread

Represents the ABT_xstream, myth_thread_t and aligned_t in Argobots, MassiveThreads and Qthreads respectively

//Mutex object

GLT_mutex

Represents the ABT_mutex, myth_mutex_t and aligned_t in Argobots, MassiveThreads and Qthreads respectively

//Barrier object

GLT_barrier

Represents the ABT_barrier, myth_barrier_t and qt_barrier_t in Argobots, MassiveThreads and Qthreads respectively

//Condition Object

GLT_cond

Represents the ABT_cond, myth_cond_t and aligned_t in Argobots, MassiveThreads and Qthreads respectively

//Team of threads' object

glt_team_t

Useful to keep the necessary information depending on the implementation.

GLT-Core

```
// ARGOBOTS

#define GLT_ult ABT_thread
#define GLT_tasklet ABT_task
#define GLT_thread ABT_xstream

typedef struct glt_team {
    ABT_xstream master;
    ABT_xstream *team;
    int num_xstreams;
    int num_pools;
    ABT_pool *pools;
} glt_team_t;


// MASSIVETHREADS

#define GLT_ult myth_thread_t
#define GLT_tasklet myth_thread_t
#define GLT_thread myth_thread_t

typedef struct glt_team {
    int num_workers;
} glt_team_t;


// QTHREADS

#define GLT_ult aligned_t
#define GLT_tasklet aligned_t
#define GLT_thread aligned_t

typedef struct glt_team {
    int num_shepherds;
    int num_workers_per_shepherd;
} glt_team_t;
```

// Library Enter and Exit points

void __attribute__((constructor)) **glt_start(**void);
void __attribute__((destructor)) **glt_end**(void);

// Initialization function.

void **glt_init**(int argc, char * argv[]);

In Argobots, by default, only one thread and one pool are created and it can be modifyed by using GLT_NUM_THREADS and GLT_NUM_POOLS environment variables.
In MassiveThreads and Qthreads, its own environment variables can be used but the GLT_NUM_THREADS overlaps MYTH_WORKER_NUM and QTHREAD_NUM_SHEPHERDS.

//Finalize function

void **glt_finalize**();

In Argobots, the Execution Stremas are joined and freed while in MassiveThreads and Qthreads execute myth_fini and qthread_finalize functions respectively.

//Work Units allocation

GLT_ult * **glt_ult_malloc**(int number_of_ult);
GLT_tasklet * **glt_tasklet_malloc**(int number_of_tasklets);

All libraries allow the use of ULT but just Argobots also allows the use of Tasklets. So, if MassiveThreads or Qthreads uses the glt_tasklet_malloc, an intern call to the glt_ult_malloc is done. The handle of the ults in Qthreads are done by using word memory status sot, one aligned_t pointer is returned.

*Possible function join (to be decided)
GLT_wu * glt_work_unit_malloc(int number, bool tasklet);

So the programmer can ask for a tasklet but if it is not possible, a ult is returned.

//Work Units creation

void **glt_ult_creation**(void(*thread_func)(void *), void *arg, GLT_ult *new_ult);
void **glt_ult_creation_to**(void(*thread_func)(void *), void *arg, GLT_ult *new_ult, int dest);

Two options are implemented, the former creates the work unit into its own wor unit structure, in the later, the destination can be selected. As MassiveThreads does not allow the second option. It just create the work units into the current thread's queue.

If Tasklets are allowed next two functions puts one task in the destination queue. If they are not allowed, a ult is used.

void **glt_tasklet_creation**(void(*thread_func)(void *), void *arg, GLT_tasklet *new_ult);
void **glt_tasklet_creation_to**(void(*thread_func)(void *), void *arg, GLT_tasklet *new_ult, int dest);

*Possible function join (to be decided)

void glt_work_unit_craetion(void(*thread_func)(void *), void *arg, GLT_wu *new_wu, bool tasklet);
void glt_work_unit_craetion_to(void(*thread_func)(void *), void *arg, GLT_wu *new_wu, int dest, bool tasklet);

So the programmer can ask for a tasklet but if it is not possible, a ult is created.

// Yield functions

void **glt_yield**();

This function gives the control to the next ult decided by the scheduler

void **glt_yield_to**(GLT_ult ult);

Argobots also allows to give the control directly to another given ult. MassiveThreads and Qthreads do not allow it so a call to glt_yield() is done.

*Possible function join (to be decided)

void glt_yield(GLT_wu wu);

If the parameter is NULL, yield function is called. If it is a valid pointer, yield_to (if possible) is called.

//Join functions

void **glt_ult_join**(GLT_ult *ult);
void **glt_tasklet_join**(GLT_tasklet *tasklet);

These functions waits until a given ult is fnished. If tasks are supported, the second function waits for a task, if not waits for a ult as in the previous function.

*Possible function join (to be decided)

void glt_wu_join(GLT_wu *wu);

Depending on the pointer, a ult or tasklet join is executed

// Mutex functions

void **glt_mutex_create**(GLT_mutex * mutex);
void **glt_mutex_lock**(GLT_mutex mutex);
void **glt_mutex_unlock**(GLT_mutex mutex);
void **glt_mutex_free**(GLT_mutex * mutex);

Common actions over a mutex object

// Barrier functions

void **glt_barrier_create**(int num_waiters, GLT_barrier *barrier);
void **glt_barrier_free**(GLT_barrier *barrier);
void **glt_barrier_wait**(GLT_barrier *barrier);

Common actions over a barrier object

//Condition functions

void **glt_cond_create**(GLT_cond *cond);
void **glt_cond_free**(GLT_cond *cond);
void **glt_cond_signal**(GLT_cond cond);
void **glt_cond_wait**(GLT_cond cond, GLT_mutex mutex);
void **glt_cond_broadcast**(GLT_cond cond);

//Common actions over condition objects

//Helpful functions

int **glt_get_thread_num**();

Return the number of the current thread

int **glt_get_num_threads**();

Return the total number of threads
// Environment variables

GLT_NUM_THREADS controls the number of threads that are created in the init function (ES, Shpeherds or Workers for Argobots, Qthreads and MassiveThreads respectively)

// ARGOBOTS
By default the number of ES and Pools is 1.
GLT_NUM_POOLS controls the number of pools in Argobots

//MASSIVETHREADS

MYTH_WORKER_NUM can be used in order to control the number of Workers in MassiveThreads (if GLT_NUM_THREADS is not set)

//QTHREADS

QTHREAD_NUM_SHEPHERDS can be used in order to control the number of Shepherds in Qthreads (if GLT_NUM_THREADS is not set)

GLT_NUM_WORKERS_PER_THREAD or QTHREAD_NUM_WORKERS_PER_SHEPHERD is used to set the number of workers per thread in Qthreads

If just one shepherd and more than a worker are created, they are bounded to node and core respectively. On the other hand, if more than a Shepherd is created, they are bounded to a core as the workers do. The Affinity is always set to true