TaskLab Library

API Specification
Release v0.1

Unicamp Team

1 Introduction

TaskLab was designed to enable the simulation of task parallelism applications. It is connected to the task runtime (e.g. MTSP in Figure 1) through a set of functions calls which are described in the sections below.

TaskLab can be useful to perform a number of jobs related to task parallelism like: (a) Measure the performance of a given runtime and compare it with other libraries; (b) Check for correctness in the execution of applications in a given platform; and (c) Allow configurable benchmarking without relying in real applications. To enable such jobs TaskLab offers the following functions to the runtime designer:

- Generate, by the user parameters, a directed acyclic graph (DAG) that would represent an arbitrary application with task parallelism;
- Serialize the generated DAG, allowing it to be reproduced repeatedly;
- Visualize the simulated application by a by generating and plotting its corresponding *.dot* file;
- Communicate directly with the runtime in order to execute the generated DAG file;
- Faster and practical performance analysis of both the application and the runtime.

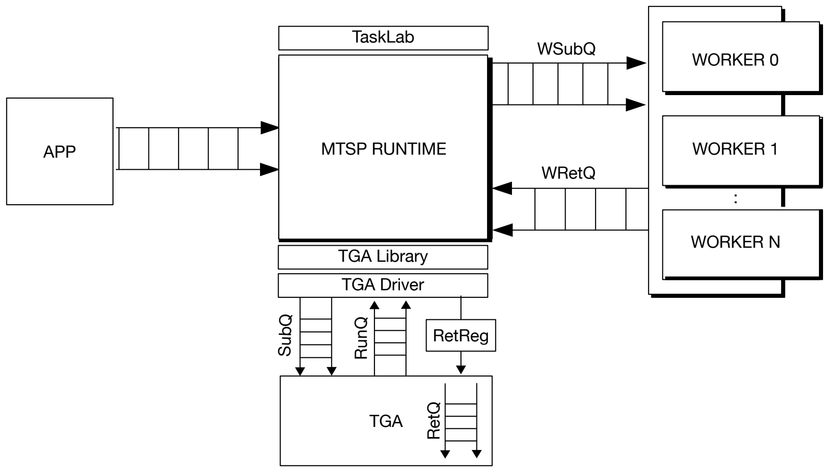

Figure 1: Architecture of the MTSP runtime and its interface to the TGA driver [1] and hardware module. The runtime establish a communication between the runtime, driver and TGA by a Runtime-Library interface and TaskLab is able to talk to the runtime throught the upper level.

2 TaskLab API

The following TaskLab API describes version 0.1. It consists of a single file, **TaskLab.h**, which contains the DAG generation and the dispatcher to the runtime, along with methods regarding visualization and serialization.

2.1 TaskLab data-structures

Isadora: Still needs to define data regarding task trace

```
class TaskLab {
  TaskGraph* tg;
  private std::unique_ptr<TaskTrace> tt; (TBD) // to be defined
}

typedef struct task_s {
  std::list<dep> predecessors;  // predecessors tasks
  std::list<dep> successors;    // by default, all successor tasks are OUT

  uint32_t tID;                 // index of task

  uint32_t npred;               // total number of predecessors
  float    load;                // how long should the task remain on load
} task;


typedef struct dep_s {
  uint32_t task;  // task that the dependency is heading towards to
  uint8_t  type;  // type of dependency
  uint32_t dID;   // index of dependency
} dep;

class TaskGraph {
  private:
    std::vector<task> tasks;        // tasks structure
    uint32_t  ntasks;               // total number of tasks
    uint32_t  ndeps;                // total number of dependencies between tasks
    uint32_t  dep_r;                // max range of how far a predecessor may be
    uint32_t  exec_t;               // standard execution time per task (ms)
    float     max_r;                // max. range from standard load time (0 to 1)
}
```

2.2 TaskLab methods

Below is the list of public and private functions available in TaskLab.

- `void TaskLab::save(const char* filename);`
  This method is responsible for saving a graph as a *.dat* file, by serializing it. It takes as parameter the `filename` of the serialized graph file.

- `void TaskLab::restore(const char* filename)`
  This method is responsible for retrieving a graph stored as a *.dat* file, by deserializing it. It takes as parameter the `filename` of the serialized graph file.

- `void TaskLab::plot(const char* filename)`
  Method responsible for saving a graph as a *.dot* file, allowing to visualize it. It takes as parameter the corresponding `filename` of the graph.

- `void TaskLab::generate`
  `(const uint32_t n, const uint32_t m, const uint32_t d,`
  `const uint32_t t, const float r)`
  Method responsible for generating the DAG that represents the application with task parallelism to be simulated. The parameters requires by the graph generation method are described below; notice that all the optional parameters have a default value, which can be defined by the user.

  - `n` is the number of tasks to be generated;
  - `m` is the maximum number of IN/INOUT dependencies that has to be created on each task, minimum is 1 by default;
  - `d` sets how far a predecessor may be from a parent (optional);
  - `t` is is the standard execution time per task (ms) (optional);
  - `r` is the maximum range from standard load time (from 0 to 1) (optional).

- `void TaskLab::dispatch(const uint8_t rt)`
  Method responsible for dispatching a graph to the `rt` runtime, displaying on-the-fly information about tasks, and whether they simulation runs correctly. The success is defined by checking if the graph was executed in the appropriate order. The dispatcher communicates with the runtime by calling its functions in order to describe each of the graph tasks and execute them. Each task triggers an arbitrary function that simply awaits for the given load time.

- `void TaskLab::burnin(const uint32_t nruns, const uint32_t max_t,`
  `const uint8_t rt)`
  This method is responsible for generating and dispatching to the `rt` runtime a total of `nruns` random task graphs, within a limit of `max_t` tasks for each graph. It displays the output for each task graph executed and if the simulation ran correctly.

  Isadora: Still to be defined. Is it necessary to keep a TaskTrace structure in order to talk with the runtime? Another option is to save it as TaskTrace and then translates to TaskGraph.

- `void TaskLab::recordTrace(task t)` (TBD)
  This method records a task and save it into the graph.

2.3 Task graph generation procedure

The generation of the task graph works as follows: first, the graph is created with `n` tasks, specified by the user. For each created task: a random load time is assigned according to the `default load time` and its `maximum range` - for example, if the default load time is 1000 ms and the load range is 0.25, it may vary from 750ms to 1250ms; a random number of dependencies is created, ranging from 1 to `maximum`, also specified by the user. Then, each dependency in the task is described as follows: a unique id is assigned, for later on purposes; a predecessor that haven't been picked yet is selected, which its `maximum distance` is defined according to the parameter of dependency range; the predecessor task receives an OUT dependency, with the same id as the successor dependency, relating both tasks; finally, the dependency is assigned as IN or INOUT, randomly.

2.4 Task graph validation procedure

The execution of the graph is validated by creating a boolean list with all the dependencies of the graph based on its unique id, assigned initially to false. It consists of: when a task executes, it assigns all of its successors' dependency as true. Then, if a task executes and all of its predecessors' dependency is defined as true, it means that is safe to execute, i.e. all of its father tasks have already executed; otherwise, the execution is not correct. If every task executed correctly, it means that the graph is validated and successfully executed.

3 Next Steps

The v0.2 release of the TaskLab is currently user development. Some features that it will provide are:

- Compatibility with other runtimes;
- More customization regarding graph generation;
- Trace capture, storage and replay of task execution.

References

[1] U. Team, "Tioga user's manual," 2016.