TaskLab User's Manual

API Specification
Release v0.4

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;
- Trace real applications and obtain its task graph structure;
- Faster and practical performance analysis of both the application and the runtime.

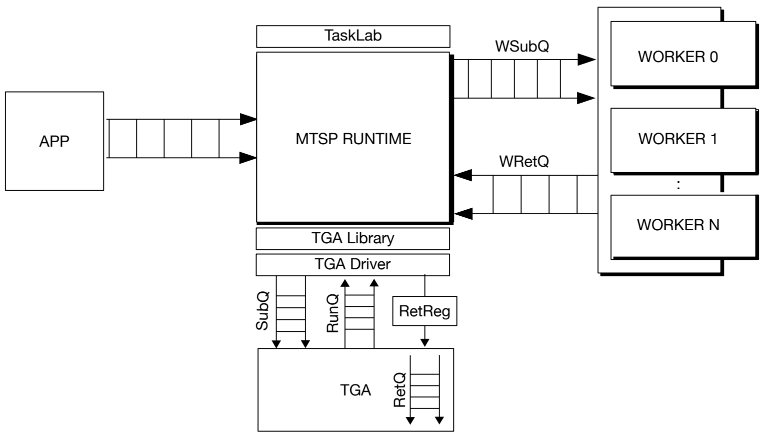

Figure 1: Architecture of the MTSP runtime and its interface to the TGA driver [1] and hardware module. The runtime establish a communication between the runtime, driver and TGA by a Runtime-Library interface and TaskLab is able to talk to the runtime throught the upper level.

2 User's level TaskLab API

The following TaskLab API describes version 0.4, by a user's level point of view. It consists of a header file, **TaskLab.h**, and its implementation, **TaskLab.cpp**, which contains functionality regarding: DAG generation; run task graphs in the runtime; visualization and (de)serialization of task graphs; trace from real applications.

2.1 Data structures

```
typedef struct task_s {
public:
    uint16_t    tID;          // hardware internal task ID
    uint64_t    WDPtr;        // address pointing to the task function
    int         ndeps;        // number of dependencies of the task
    dep*        deparr;       // list of dependencies of the task
} task;


typedef struct dep_s {
    uint64_t     varptr;      // address of the dependency variable
    uint8_t      mode;        // mode of the variable
} dep;
```

2.2 Classes

Above is the class used by the API, **TaskLab**, which is Object-Oriented. Notice that, since this is a part from the User's level API, private methods were ommited for a clearer understanding.

```
class TaskLab {
public:
    void generate(const uint32_t n, const uint32_t m,
                  const uint32_t d, const uint32_t t,
                  const float    r);
    bool run(const uint8_t rt);

    void burnin(const uint32_t nruns, const uint32_t max_t, const uint8_t rt);

    bool hasEvent(uint8_t event);
    void watchEvent(uint8_t event);
    void eventOccurred(uint8_t event, void* t);

    bool save(const char* filename);
    bool restore(const char* filename);
    bool plot(const char* filename);
    bool empty(uint8_t evt);
}
```

2.3 TaskLab method descriptions

Below are the description of **public** functions available in TaskLab.

- `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 a DAG, which represents an application with task parallelism to be simulated. The parameters required for the generation 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 per task, i.e. amount of iterations (optional);
  - `r` is the maximum range from standard execution per task (from 0 to 1) (optional).
- `void TaskLab::run(const uint8_t rt)`
  Dispatches a current loaded graph to the `rt` runtime, displaying on-the-fly information about tasks, and whether if the simulation ran correctly. Its success is defined by checking if the graph was executed in the appropriate order, i.e. it respected the given dependencies. TaskLab communicates with the runtime by calling functions that receives tasks and execute them. Each task calls an arbitrary function which 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 by graph. It displays output for each task graph and if the simulation ran correctly.
- `bool TaskLab::hasEvent(const uint8_t event)`
  Checks if an `event` is currently being watched by the API - if so, returns true, otherwise false.
- `void TaskLab::watchEvent(const uint8_t event)`
  Notifies API to watch an `event` from a running application, i.e. record task dispatching.

- `void TaskLab::eventOccurred(const uint8_t event, void* t)`
  Method used by the runtime in order to notify API that an event occurred, if necessary. Event is of type `event` and `t` is its content. Currently supported types are described below.
  - Event of type `Evt::TASK`; it receives an parameter `t` of type `task`, which is added to the task graph.
- `bool TaskLab::save(const char* filename)`
  Save a graph as a *.dat* file, by serializing it. It takes as parameter the `filename` of the serialized graph file. Returns if serialization was successful.
- `bool 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. Returns if deserialization was successful.
- `bool TaskLab::plot(const char* filename)`
  Save a graph as a *.dot* file, allowing to visualize it. It takes as parameter the corresponding `filename` of the graph. Returns if the plot went successful.
- `bool TaskLab::empty(uint8_t evt)`
  Simply checks if TaskLab is empty, i.e. has a valid task graph structure of type `evt`.

2.4 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 amount of execution is assigned according to the `standard execution` and its `maximum range` - for example, if the default execution size is 1000 and the load range is 0.25, it may vary from 750 to 1250; 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.

2.5 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.

2.6 Task tracing procedure

The task tracing procedure consists in a runtime communication with the TaskLab API. Basically, an environment variable EVT_VAR must be set with the event to be traced: high level task or low level task. When the MTSP is initialized, it verifies if the environment variable is enabled. If so, it uses a TaskLab object to enable the following event (calling `watchEvent()` method). The communication proceeds to be established relying on the TaskLab API.

If the user wants to trace high level tasks, the runtime must send a structure relative to `task` and `dep` structure for each dispatched task; if user wants to trace low level tasks, the runtime must send packets with TIOGA [1] intermediate format. When the execution is finished, it must serialize the TaskLab object as a temporary object, saved (by default) at */tmp/taskgraph.dat*. The Ferret, which is a TaskLab client, simply deserializes this object and successfully completes the tracing.

3 Internal TaskLab API

The following TaskLab API describes version 0.4 under its internal point of view.

3.1 Data structures

Notice that the API use different structures in order to handle its internal data (such as `_task` and `_dep`), which would be closer to a graph structure.

```
typedef struct dep_i {
public:
    uint32_t task;  // task that the dependency is heading towards to
    uint8_t  type;  // type of dependency
    uint32_t dID;   // index of dependency
    uint32_t var;   // var that relies on
} _dep;

typedef struct task_i {
public:
    std::list<_dep> predecessors;  // predecessors tasks
    std::list<_dep> successors;    // successors tasks
    uint32_t tID;                  // index of task
    uint32_t npred;                // total number of predecessors
    float    exec;                 // how long should the task remain executing
} _task;
```

3.2 Classes

Below is the TaskGraph class structure, used for task graph management. Notice some private methods were still omitted since they are not relevant to understand the API under its internal point of view.

```
class TaskGraph {
public:
    // feed the graph with generated tasks
    void create_tasks(const uint32_t max_dep);

    // describe dependency between tasks of the graph
    void describe_deps(const uint32_t tID, uint32_t* dep_id,
                       const uint32_t min, const uint32_t max);

    // add a given task to the graph, solving dependencies
    void add_task(task t);

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  nvar;            // total number of variables shared 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 execution time (0 to 1)

    std::map< uint64_t, std::vector<_dep> > in_map;
    std::map< uint64_t, _dep >              out_map;
    std::vector<uint64_t> ll;  // low level task graph structure
}
```

4 Ferret

Ferret is a TaskLab client created to enable a smoother communication between the user and the TaskLab API. It plays a role of a debugger for task parallelized applications and simulations in general. The usage is described as follows:

- ‘Generate’ in order to generate a random task graph;
- ‘Run’ in order to run a current loaded task graph;
- ‘Burnin’ in order to generate multiple task graphs and run them;
- ‘Trace’ in order to trace a program;
- ‘Save’ to save a current loaded task graph;
- ‘Restore’ to restore and load a saved task graph;
- ‘Plot’ to plot a current loaded task graph.

5 Next Steps

The v0.5 release of the TaskLab is currently under development. Some features that it will provide are:

- Compatibility with other runtimes;
- More customization regarding graph generation;
- More trace options;
- Enable to trace multiple options at the same time;
- Enable debugging/trace for generated task graphs, besides real applications.

References

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