Tasklab

A task simulation API
v0.1

Alexandre Medeiros Isadora Sophia

1 Introduction

The tasklab was designed to enable the simulation of task parallelism applications without relying on multiple benchmarks. The API consists in enabling the user:

- 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 *.dot* file;
- communicate directly with the runtime in order to execute the generated DAG file;
- faster and practical performance analysis.

2 Tasklab API

The following tasklab API describes version 0.1. It consists of three files: **common.h**, which consists of the graph data and serialization functions; **gen-dag.h**, implements the DAG generation; **dispatcher.h**, dispatches the graph to the runtime.

2.1 Common.h

```
void Graph::save(const Graph &g, const char* filename);
```

Method responsible for saving a graph as a *.dat* file, by serializing it.

- g: graph to be serialized;
- filename: name of the file.

```
void Graph::restore(Graph **g, const char* filename);
```

Method responsible for retrieving a graph stored on a *.dat* file, by deserializing it.

- g: where the graph should be restored;
- filename: name of the serialized graph file.

```
void Graph::show(const Graph &g, const char* filename);
```

Method responsible for saving a graph as a *.dot* file, allowing to visualize it.

- g: graph to be saved;
- filename: name of the file.

2.2 Gen-dag.h

```
void Gen_dag::generate(Graph** graph, uint n, uint m,
                       uint d = DEFAULT_DEP_RANGE,
                       uint t = DEFAULT_LOAD_TIME,
                       float r = DEFAULT_LOAD_RANGE);
```

Method responsible for generating the DAG that represents the application with task parallelism to be simulated. Each parameter corresponds as follows:

- graph: pointer where the final graph will be stored. It must be freed by the user afterwards;
- n: number of tasks to be generated;
- m: maximum number of IN/INOUT dependencies that has to be created on each task;
- d: how far a predecessor may be from a parent (optional);
- t: standard load time per task (ms) (optional);
- r: maximum range from standard load time (from 0 to 1) (optional).

All the optional parameters have a default value, which can be defined by the user.

2.3 Dispatcher.h

```
void Dispatcher::dispatch(Graph* g);
```

Method responsible for dispatching a graph to the MTSP runtime, displaying on-the-fly information regarding the tasks and if it succeeded the execution. The success is defined by checking if the graph was executed in the appropriate order.

- g: graph to be dispatched.

3 Useful cases

Some cases that can be pointed out as examples in which the tasklab API may be useful:

1. Measure a performance of a given runtime and compare it with another libraries;
2. Check for correctness in the execution of applications in a given plataform;
3. Allow multiple benchmarking without relying in real applications.

4 The design

4.1 How it works

The generation 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.

The dispatcher works by simply communicating 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.

4.2 Graph format

The DAG is saved according to the graph data in **common.h**. The code below corresponds to the public data of the **Graph** class:

```
std::vector<Task> tasks;          // tasks structure
uint  total_tasks;                // total number of tasks
uint  total_deps;                 // total number of dependencies between tasks

uint  dep_range;                  // max range of how far a predecessor may be

uint  load_time;                  // describes the standard load time per task (ms)
float max_range;                  // max. range from standard load time (0 to 1)
```

Public data of **Task** class:

```
std::list<Dep> predecessors;      // predecessors tasks
std::list<Dep> successors;        // by default, all successor tasks are OUT

uint  C_dep_tasks;                // total number of predecessors
float load;                       // how long should the task remain on load
```

Finally, public data of **Dep** class:

```
uint task;                        // task that the dependency is heading towards to
Type type;                        // type of dependency
uint index;                       // the index of the dependency
```

Where **Type** stands for an enum with values IN, INOUT and OUT.

4.3 Graph validation

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.

5 Future

A 0.2 release of the tasklab application is currently being developed. Some features in development are:

- Allow compatibility with other runtimes;
- Enable more customization regarding graph generation.