## The Task Scheduler

Andrew Dalia

### Overview

The program generates a process schedule for a set of tasks running against

a heterogeneous cluster of computers.

This program will take a file containing tasks to run and a file containing

computers that can process these tasks as input. The output will be the

schedule that tasks are processed in.

### Execution and Usage

The program is written in Python under version 2.7. It should be called **uberatc\_assignment.py** (for reference in this documentation). It can be invoked directly from any command line as below (-h will give a help menu):

>>> python uberatc\_assignment.py -t tasks.yaml -c computers.yaml

There is one third party module used in program. This is the sortedcontainers module. From here the SortedSet class is used. Information on this package can be found at <http://www.grantjenks.com/docs/sortedcontainers/>

It can be installed from the command line using:

$ pip install sortedcontainers

The inputs to the program are the tasks and computers input files. Both are required and are specified by the -t and -c command line parameters respectively.

A **tasks.yaml** file with the tasks data should be included with a format similar to the below where each task must have a cores\_required and execution\_time field while the parent\_tasks field is optional

task1:

cores\_required: 2

execution\_time: 100

task2:

cores\_required: 1

execution\_time: 200

parent\_tasks: task1

task3:

cores\_required: 4

execution\_time: 50

parent\_tasks: "task1, task2"

A **computers.yaml** file should be included with a format similar to the below where each row is a computer name to number of cores pair.

compute0: 14

compute1: 12

compute2: 15

Also, 2 other files are included to help generate the computers and tasks files. These are **random\_task\_file\_generator.py** and **random\_computer\_file\_generator.py**. They will generate a random data set for these 2 input files within certain limits which can be modified in the scripts.

### Design and Implementation

#### Classes

There are 3 different classes defined: Task, Computer, and Scheduler.

**Task:** Contains all fields specific to an individual task

**Computer:** Contains all fields specific to an individual computer

**Scheduler:** Handles processing of both Tasks and Computers to come up with an overall schedule for the tasks to run

#### States

The program defines 4 states a task can be in: waiting, ready, running, and done.

**Waiting:** In the waiting state, a task is not ready to be run since it has parent tasks that have not completed. The task is to be marked as ready when all its parents are completed based on the newReadyTasks set

**Ready:** In the ready state, all of the tasks’ dependencies (parents) have completed, so it is available to run as soon as one of the computers has enough available cores to process the task. The readyTasks SortedSet has the tasks ready to run.

**Running:** In the running state, the task is being executed as it has been picked up by a computer to process. It will run for its given execution time and then complete. The runningTasks set has the tasks in this state.

**Done:** In the done state, the task has finished executing and the processing cores have been freed and made available to the computer. The count of the completed tasks is kept in numCompletedTasks

#### Data Structures

Most of the top level data structures used were either dictionaries or sets (or a combination). This is because they were used primarily for lookups, insertions, and deletions, which is handled most efficiently using these (average case constant time). Also, in all these cases we do not want duplicate entries. In order to store the tasks, we use the taskMap which is a lookup table with the key being the task name and the value being the task. The computers are kept similarly in the computerMap, with the key being the computer name and the value being the computer object. The task and computer objects are only stored in this container and so lookup by name which is the most common operation can be done in O(1) time. The dependentsMap is also crucial because it maps a parent task to a set of all its dependents. This is so if a task completes, we can quickly find the dependent tasks and indicate that they are no longer waiting on the parent. The readyTasks container is the only SortedSet. The SortedSet is a tree based set data structure that provides inherent ordering to allow most operations in O(logN) time. It is very useful to have this for the ready tasks since we want to be able to iterate through them in order of the number of dependents a task has from highest to lowest. Therefore, readyTasks uses a custom key that compares the amount of dependents a task has. After a few tests, it looks like this improved the performance of the overall processing of the ready tasks by about 40% compared to using a standard python set. The standard set is hash based and does not have inherent ordering which required sorting the container prior to iteration. The schedule itself is a list containing tuples with information on the tasks/computers in the order in which the tasks being executing.

#### Dependencies

A task will continue to be waiting as long as any of its parents are still incomplete. One of the goals of the design was to be able to handle dependencies as optimally as possible. To accomplish this, we process each task in order of number of dependents using the dependentsMap from highest to lowest. Therefore, the tasks that have the most children waiting for them to finish can be completed as soon as possible as long as they have the appropriate number of cores available to them. Tasks that have no or very few dependents can wait to be executed later since they are not preventing other tasks from being ready to execute and hence affecting the parallelism.

#### Task to Computer Matching

When a task is ready to be executed, it must be picked up by a computer that has enough available cores to process it. To accomplish this, we take a best effort approach that looks at how many cores a task needs to execute and looks for the computer with the same or as to close to but greater number of cores available. This is to try to utilize each computer to its best capacity. If there is an exact match, then it will execute on that computer to not leave any unnecessary open cores since it makes more sense to keep other computers open with more cores available to be able to pick up other or larger tasks. This is likely not the most optimized heuristic but seems simple and effective in practice.

#### Scheduling

When a task begins executing, the scheduler will add the task to the overall process schedule with information such as start time of execution, end time, task name, computer name, and number of cores available out of the total for that computer. This is to help looking into issues, like if the cores on each computer seem to be utilized well or even debugging if something looks off. The status of the incomplete tasks are checked at each timestep to see if they need to be promoted. One sanity check that is performed at the beginning is to see if any of the tasks requires more cores than are available on any computer. Scheduling will be impossible in this case. The same check is applied when determining if a ready task can possibly be executed on any computer at a specific point in time.

### Testing

#### Data set generator

In order to test, random task and computer yaml files were generated using random\_task\_file\_generator.py and random\_computer\_file\_generator.py. There are limits to randomness of data they can generate and these can be tweaked within the script. An example of the files generated using the scripts are tasks.yaml and computers.yaml. I have tested against several files generated using these.

### Output

There are two different types of output defined: the timed schedule, and the format schedule.

#### Timed schedule

The timed schedule outputs the schedule in order of when the tasks started to execute with all the information described in the ‘Scheduling’ section above: start time of execution, end time, task name, computer name, and number of cores available out of the total for that computer. This is useful to get a full picture of the execution. The function to display it is printTimedSchedule.

#### Format schedule

This schedule contains pairs of task name to the computer it ran on ordered by the time they started to execute. This can be thought of as an abbreviated version of the timed schedule. The function to display it is printFormatSchedule.