Algorithmique du Parallèlisme, Ordonnancement de Tâches

Lab Session 1

M2 CISD, ENSEIRB-MATMECA

Francieli Zanon-Boito, Laércio Pilla

November 2019

Part 1 - Load Balancing

In this first exercise, we will use a high-level simulator to test load-aware scheduling algorithms.

To start, download it and decompress it from:

http://francielizanon.github.io/teaching/TP/load_balancing_sim.tgz

```
tar xzf load_balancing_sim.tgz
cd load_balancing_sim/
```

In this simulator, we have a list of tasks that are to be processed. Each task is represented by its load, or amount of computation required. There are also a number of identical processing elements (PEs), where tasks can be executed. The whole simulation consists of applying an algorithm (a scheduler) to decide where (in which PE) each task should be processed. Then the tool reports some metrics on the load imbalance of the achieved solution.

To run a simulation, you will need to write a short code in Python. We describe below the required steps to run a simulation. You can also look at the **example_*.py** files in the provided folder for examples.

A simulation starts by defining a list of tasks, which you can do in **one of the following ways**, for different distributions:

- `myTasks = Tasks.range(minimum, maximum, step)`
- `myTasks = Tasks.normal(mean, standard deviation, number of tasks, random seed)`
- `myTasks = Tasks.lognormal(mean, sigma, number of tasks, scale, random seed)`
  Each number sampled from the `lognormal(mean, sigma, number of tasks)` distribution will be multiplied by the *scale* parameter
- `myTasks = Tasks.uniform(minimum, maximum, number of tasks, random seed)`

Or you can directly provide a list of loads to `Tasks.from_loads()`.

Then you'll need a list of PEs, created simply with:

```
myPEs = PEs(number of PEs)
```

Finally, to run the simulation and report the results:

```
simulation = Scheduler(myTasks, myPEs, "myalgorithm")
simulation.work()
simulation.stats()
simulation.plot_all()
```

In the code above, "*myalgorithm*" is to be replaced by the name of one of the algorithms available in **sched/algorithms.py**. Initially, only "*roundrobin*" is available.

1. Observe the implementation of the *round-robin* algorithm, in the **algorithms.py** file. Generate and execute some simulations with different numbers of tasks and PEs, and with different load distributions for the tasks. What is the impact of these parameters on the results from this scheduler? What would be the adversary case for this scheduler?
2. The provided round-robin algorithm is **not** load-aware, i.e. it does **not** consider the load of the tasks when distributing them across the PEs. Add a *basic list scheduling* option in the **algorithms.py** file. Your algorithm is to go over the list of tasks, assigning to each one of them the PE with the lowest current load (you'll have to keep track of the total load given to each PE, which is the sum of the loads of the tasks given to that PE).
3. Repeat your simulations using *listscheduling* instead of *roundrobin*. Are results better? What would be the adversary case for this algorithm?
4. **(To go further)** Try to propose new scheduling algorithms to further improve results. How do your algorithms compare to *listscheduling*? What about their complexity?

Part 2 - Online job scheduling

For this exercise we will use another high-level simulator to implement and evaluate online job schedulers. Download it and decompress it from:
http://francielizanon.github.io/teaching/TP/job_scheduling.tgz

```
tar xzf job_scheduling.tgz
cd job_scheduling/
```

Inside the folder, you will find the **ANL-Intrepid-2009-1.swf** file, obtained from the ANL Intrepid Log (https://www.cse.huji.ac.il/labs/parallel/workload/l_anl_int/). It contains information about 68,936 jobs submitted to the Intrepid supercomputer (Argonne National Laboratory, USA) over 240 days of 2009. At the

time, the machine of 40,960 quad-core nodes was among the world's 10 fastest supercomputers (https://www.top500.org/lists/2009/06/).

About each job, the dataset contains (among other things) its submission time (when an user has asked the system to run it), number of requested nodes, amount of requested time, and its actual execution time. The replay.py script uses this information to simulate the job scheduler operation: jobs are added to a queue when submitted, and a scheduling algorithm is used to decide when to remove jobs from the queue to run them.

At the end of the simulation, the makespan is reported, together with statistics on the wait times (time spent in the queue by the jobs while waiting to be scheduled) and on the usage of the machine.

To run the simulation, you'll use the following command:

```
python replay.py "scheduling algorithm" "number of  nodes" "number of jobs"
```

The first argument is the scheduling algorithm being used, among the ones implemented in the **algorithms.py** file, similarly to what was done in the simulator from the part 1 of this lab session. The simple *fifo* (first-in, first-out) scheduler is provided with the simulator. The second argument allows for trying the same dataset over a machine of different sizes.

A third **optional** argument allows for limiting the number of jobs taken from the input dataset and simulated. It might be useful if your simulations are too long.

1. Run the simulation for three different machine sizes (numbers of nodes) and observe the impact on the metrics reported at the end. How are they affected? Why?
2. Observe the implementation of *fifo* in the **algorithms.py** file. Try to propose other scheduling algorithms to improve the wait times. Repeat the simulations to evaluate your proposal.