Job, parameter and data management using a database

Emyr James

Outline

- Why ?
- How ?
- Advantages
- Can I use it?

Why ?

- MSc Project on Genetic Programming (Fancy Genetic Algorithms)
- 14 algorithm variants
- 7 test problems
- 30 for each for stats
- ~3000 runs
- How to keep track of all these??

How ?

- Store details about the runs in a database
- Used Postgresql – free & open source
- Feature rich - stored procedures, foreign key constraints etc.
- Design Schema : Tables, data types and stored procedures
- Interact with database from C++ using libpqxx

The Run Table

```
create table t_run (
        run_id serial primary key,
        expt_id int references t_experimet.expt_id,
       job_id int,
        started bool default FALSE,
        finished bool default FALSE,
        start_time timestamp,
        finish_time timestamp,
        rng_x0 bigint,
        rng_x1 bigint,
        rng_x2 bigint,
        rng_x3 bigint,
        rng_x4 bigint,
        host varchar(255)
);
```

The Experiment Table

```
create table t_experiment (
   expt_id serial,
   algo_id int,
   problem_id int,
   run_expiry_seconds int,
   pop_size int,
   init_depth int,
   prob_crossover float,
   prob_mutation float,
   max_tree_size int,
   spatial_radius int
);
```

Start and Finish Stored Procedures

- start_next_run called at start
  - Passes in RNG state & Hostname to DB
  - Get next run to process I.e started=FALSE
  - Mark as started and set time stamp
  - Return custom datatype populated with run parameters
- mark_run_as_finished called when program finished
  - Pass in run_id
  - Mark as finished and set finish time

Custom Datatype

```
create type tp_run_params AS (
   run_id int,
   problem_name char(8),
   algo_name char(10),
   ep_order int,
   poly_coefficients varchar(255),
   poly_min_x float,
   poly_max_x float,
   poly_points int,
   pop_size int,
   init_depth int,
   run_expiry_seconds int,
   max_tree_size int,
   spatial_radius int
);
```

Process

- Populate Experiment table with data for experiments
- Populate Run Table based on above (e.g. 30 runs per experiment)
- Use qsub to queue up jobs (no parameters needed)
  - Jobs start on nodes..
  - start_run to get info for next run & mark as started
  - Does work
  - Call mark_as_finished at end

The Code

One Simple Run Script

```
#!/bin/bash
#$ -S /bin/bash
#$ -o /path/to/output/logs
#$ -e /path/to/error/logs
PATH=/first/path:/second/path
LD_LIBRARY_PATH=/firs/lib:/second/lib:/third/lib
my_job $JOB_ID
```

Easy to Queue up Jobs

```
#!/bin/bash
for {x in 1..3000}
do
   qsub -my.q run_script.sh
done
```

Other Advantages

- Web Portal
  - Upload a text file / spreadsheet to populate tables
  - Button to press which can kick off runs
  - Can report on what's done, what's in progress
  - Failed jobs – started but not finished
  - Easy to redo only jobs that failed, see what parameters that job had
- Coherent history of what you've done
  - Can go back and redo experiment – just mark all runs as not started

Can I Use It ?

- Easy if you write your own programs
  - Design problem specific experiment & parameter tables
  - If not using C++, easy to port the database stuff to python, java, C, Fortran etc.
- If Not ?
  - How do you get parameters into your 'off the shelf' program?
  - Usually command line or config file
  - Wrap it all up in a python script – call python wrapper instead of your job

Python Wrapper

- Do same start_run procedure as C++
- Generate a config file called <run_id>.cfg using a template file – use Cheetah python module
- Or just generate a command line string
- Call your actual job from within python using the subprocess module
- Call the mark_run_as_finished at end of python wrapper