CSC 2012 – Virtualization Exercises

Part II

Note: These exercises are available online under
`http://cvmrepo.web.cern.ch/cvmrepo/csc12/cernvm-exercises.pdf`

Today's Synopsis

- Run a distributed analysis job using a single *master node* and multiple *worker nodes*
- Split a workload into small pieces by specifying data dependencies
- Problems of scaling parallel applications in the master-worker style

Exercise 3 – Monte Carlo π

In this exercise, we will use a Monte Carlo method (i. e. we roll dices) in order to compute an approximation of π. In order to do so, we randomly shoot points into a square with side length 2 that encloses a circle with radius $r = 1$ (Figure 1).

This computation can be very well parallelized. You'll find a program that shoots randomly in the square and keeps track of the number of hits inside the circle (`mcpi-worker`). You'll also find a program that takes the

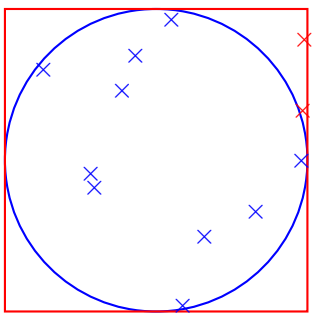

Figure 1: Random points shot inside and outside the unit circle.

overall number of shots and the number of hits inside the circle to compute π (`mcpi-merge`).

The `mcpi-worker` instances can run independently on many of your virtual machines in parallel. Their result is sent back to an instance of `mcpi-merge` for final calculation. We will use a system called *Makeflow* that automatically steers the workflow and distributes the computation jobs to all available virtual machines. In order to make Makeflow work, you have to create a dependency file that consists of *rules* in the following form:

```
<output file>:  <input files and processing program>
          <command line to produce output file>
```

Every rule can potentially run on another virtual machine. A rule will only be executed if all its dependencies are present, either as initial input or as a result from the execution of other rules. You can see how it looks like for this particular problem in the file `mcpi.makeflow`. The workflow is started by `makeflow -T wq -a -C 130.238.25.21:9097 -N <CSC-GROUP> mcpi.makeflow`. Don't worry about the extra options; they are required to distribute the work on the infrastructure of these exercises.

In fact, a Makeflow file looks rather similar to an ordinay Makefile. One of the differences is that you have to additionally specify the program required for processing as a dependency. The prefix "LOCAL" tells makeflow to execute this specific rule on the invoking machine, and not on one of your workers.

1. You know already how to boot a Desktop CernVM as well as a zoo of batch CernVMs. Now we need it. Boot your Desktop CernVM and the CernVM zoo of 2 batch nodes and check that all the machines are up and running.

2. Log into your Desktop CernVM. Download and unpack the exercises by

   ```
   wget http://cvmrepo.web.cern.ch/cvmrepo/csc12/cernvm.tar.gz
   tar xvfz cernvm.tar.gz
   ```

   Go to the directory cernvm/03mcpi and run `make` to compile the sample programs.

3. Run the distributed makeflow from a terminal in your Desktop CernVM with

   ```
   makeflow -T wq -a -C 130.238.25.21:9097 -N <CSC-GROUP>  \
     mcpi.makeflow
   ```

   The result of your calculation is in the file PI.

4. Ooops. Someone forgot to insert the formula to get π from the Monte Carlo results.
   Can you help and fix `mcpi-merge.cc`? **Hint:** The area of a circle is πr^2. You can cleanup the makeflow results with `makeflow -c mcpi.makeflow`.

5. Use the Makeflow file to calculate π up to the first 5 digits ($\pi = 3.1415\cdots$). Play with the number of trials (shots) to improve the precision.
   How many trials did you eventually use?
   Bonus task: Can you figure out analytically how many trials you would need so that the error of your computation is with a probability of 95 % not more than 10^{-5}?

Exercise 4 – Who's a Hub?

This exercise will use Makeflow to analyze graphs (more on graphs in the appendix). A common analysis on graphs is to find out which of the vertices are important (for a suitable definition of "important"). Such a function $f : V \to \mathbb{R}$ that rates the vertices of a graph is called a *centrality*. In the following, we will deal with the *betweenness centrality*. Let $\sigma(s,t)$ be the number of shortest paths between the vertices s and t and $\sigma_v(s,t)$ the number of shortest paths between the vertices s and t via v. Betweenness is then defined as

$$\text{Betweenness}(v) = \sum_{\substack{s,t \in V \\ s \neq t \neq v \neq s}} \frac{\sigma_v(s,t)}{\sigma(s,t)}$$

That means: a vertex v that is on many shortest s-t-paths is rated high. Such vertices are usually hubs or router vertices such as motorway junctions. See Figure 2 for an example.

The computation of betweenness is nicely parallelizable. To do so, we compute independently for every vertex v the *betweenness dependency* $\delta_v(w)$

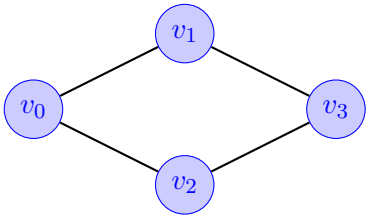

Figure 2: Betweenness example. Vertex v_1 is a middle node on the shortest path $v_0 - v_1 - v_3$, but not on the shortest path $v_0 - v_2 - v_3$. Hence its betweenness is 1/2.

of v to all other vertices w. For the moment, we do not care how exactly these dependencies are computed. We just note the relationship between dependency values and betweenness, which is:

$$\text{Betweenness}(w) = \sum_{v \in V} \delta_v(w).$$

1. Go to the directory cernvm/04hubs and run `make` to compile the sample programs. Have a look at the programs `betweenness`, `dependency`, and `dsum`. They can be used to compute the betweenness of graphs, either in one go (`betweenness`), or by computing the dependencies of small chunks of vertices (`dependency`) and then summing up the partial results (`dsum`). Compute the betweenness of the toy graphs `star.graph`, `grid.graph`, and `complete.graph`.
   What do you get?

2. The `routes.graph` file represents 3219 major airports and the airline routes between them[1] (Figure 3). Write a Makeflow file to distribute the computation using the `dependency` and the `dsum` programs.
   According to this analysis, which are the 3 most important airports?
   Hints: The exact number of vertices has been mentioned because it is required to parallelize the work. If you have problems with your Makeflow file, try using the Makeflow option `-d all` for more detailed output. You might want to use the UNIX utility `sort -n -k2,2` on the final result. You'll find a table that maps the airport vertex number to an airport name in the file `airport.dat`. The UNIX `grep` utility might become handy.

3. Measure the speed-up you get using all virtual machines (Makeflow) compared to just 1 virtual machine. In order to run the computation only on a single machine, you can either use the `betweenness` program or use makeflow without the extra options, which will run it locally.
   Hint: the UNIX `time` utility can be helpful to measure running times.

4. Suppose you wouldn't know beforehand how many virtual machines you have.
   Would that change the way your Makeflow file look?
   Would that change the way you created the file?
   Can you think of a potential new problem that might arise?

Bonus exercise. Calculate the betweenness of the graph `smallworld.graph` (the graph has 4096 vertices). So-called *small-world graphs* are characterized by only few edges per vertex on average and very short connections from every vertex to every other vertex. They were built after the phenomenon

[1] See `http://openflights.org/data.html`

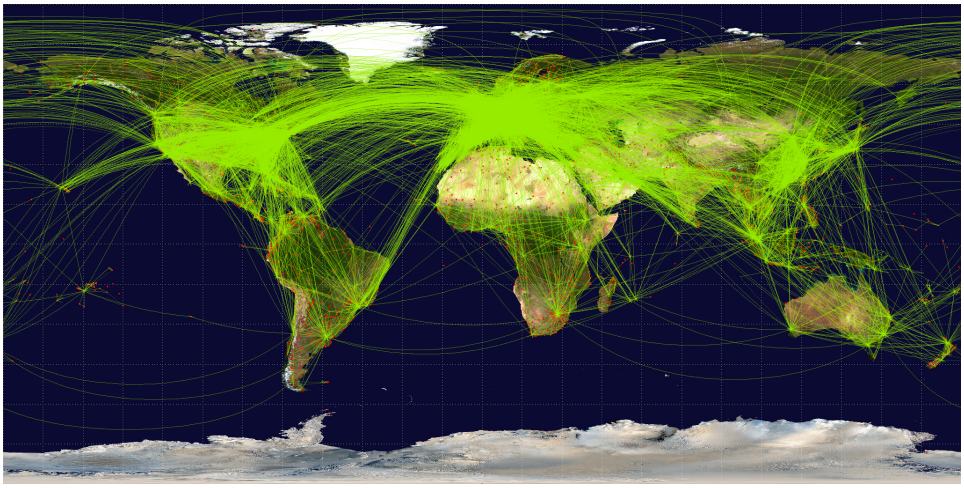

Figure 3: Major airports and airline routes between them. Data and visualization comes from OpenFlights.org. With the naked eye, it's difficult to retrieve information from larger graphs.

that every person on earth knows every other person via only few intermediate hops. You'll find a ROOT macro `plot-betweenness.C` that you can use to plot the betweenness distribution of your result. You might have to change the name of the input file in the macro. Setup ROOT by

```
source setup_root.sh
```

Run the macro by `root -l plot-betweenness.C`
Can you infer from the distribution anything about hubs?
Do you have an assumption which distribution this is?
Which impact does it have on the reliability of small-world networks?

A Brief Recap on Graphs

A graph consists of a set of *vertices* that are connected by *edges*. Figure 4 shows some graphs. Many relationships can be modeled as graphs, for example the hosts on the Internet and their network connections, the street network, or the `#include` dependencies of a C++ source file.

There are many different incarnations of graphs, directed and undirected, weighted and unweighted, simple graphs and multigraphs, finite and infinite graphs. In this exercise, we deal with *finite, simple, undirected, unweighted*

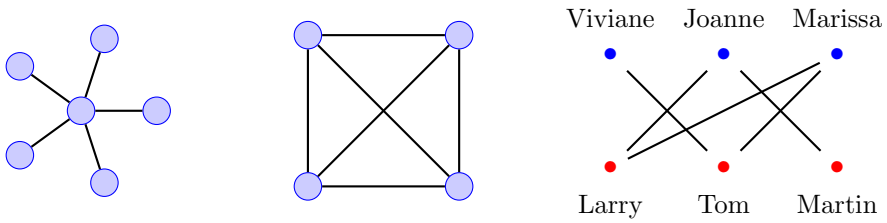

Figure 4: A few graphs. Left side: a star (6 vertices, 5 edges). Middle: a complete graph (4 vertices, 6 edges). Right side: a *bipartite* graph that models who wants to marry whom (can you help them make a match?)

graphs. In this form, a graph $G = (V, E)$ is a tuple consisting of a finite set of vertices V and a finite set of edges $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$. For simplicity, we denote the vertices with numbers $0, \ldots, n-1$. A *path* from vertex v_0 to vertex v_k is a sequence $(v_0, \ldots, v_k)$ of vertices in which successive vertices have to be connected by an edge. In this exercise, we only deal with paths without loops, i. e. the vertices of the path have to be pairwise distinct.

Further Reading

- Albrecht, Donnelly, Bui and Thain, *Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids*, SWEET'12, 2012
- Buncic et al., *A practical approach to virtualiztion in HEP*, The European Physical Journal Plus **126**:1, 2011
- Brandes, *A Faster Algorithm for Betweeness Centrality*, Journal of Mathematical Sociology **25**:2 pp. 163-177, 2001
- Koschutzki et al., *Centrality Indizes*, in "Network Analysis", LCNS 3418 pp. 16-104, Springer, 2005
- *Stanford Network Analysis Platform*, `http://snap.stanford.edu`