Data Intensive Computing – Handout 1

Sun Grid Engine and others

- Since 2001, open-source.
- In 2009 Sum bought by Oracle – Oracle Grid Engine, no longer open-source.
- Two major open-source forks, one of them (Son of Grid Engine) still active.

`qsub`: Submits a job for execution

- `-b [y|n]` binary or a script; always use `-b y`
- `-cwd` keep current working directory
- `-v variable[=value]` defines or redefines environment variable
- `-V` export all environment variables
- `-N name` set name of the job
- `-o outpath` set file with the standard output of the job; default `$JOB_NAME.o$JOB_ID`
- `-e outpath` set file with the standard error of the job; default `$JOB_NAME.e$JOB_ID`
- `-j [y|n]` merge standard output and error
- `-sync [y|n]` wait till the job finishes
- `-l mem_free=1G` set required amount of memory
- `-l h_vmem=1G` set maximum amount of memory; stop job if exceeded
- `-hold_jid comma_separated_job_list` jobs that must finish before this job starts
- environmental variable `JOB_ID`
- environmental variable `JOB_NAME`

Array jobs:

- `-t 1-n` start array job with n jobs numbered $1 \ldots n$
- environmental variable `SGE_TASK_ID`
- output and error files `$JOB_NAME.[eo]$JOB_ID.$TASK_ID`
- `-t m-n[:s]` start array job with jobs $m, m+s, ..., n$
- environmental variables `SGE_TASK_FIRST`, `SGE_TASK_LAST`, `SGE_TASK_STEPSIZE`
- `-tc j` run at most j jobs simultaneously
- `-hold_jid_ad comma_separated_job_list` array jobs that must finish before this job starts; task i depends only on task i of specified jobs

`qstat`: List of running jobs

Detailed information about a job can be obtained using `qstat -j job_id`.

`qdel`: Stops jobs with given ids

`qrsh`: Starts interactive shell – think `ssh`

Training Cluster

You can login to the training cluster *dlrc* via machine `ufallab.ms.mff.cuni.cz` and port 11422, i.e., using `ssh -p 11422 ufallab.ms.mff.cuni.cz`.

All machines in the cluster use same architecture and share `/dlrc_share` directory.

The cluster consists of master `dlrc` and 5 nodes `dlrc-node1` to `dlrc-node5`. Each node has 2 cores and 4GB ram, but four SGE jobs are allowed to run simultaneously (only because otherwise there would be too little slots; the slot number may be even increased later).

Splitting Large Files on Lines

To help you with splitting large files on lines, you can use supplied `split_file.py` script from `/dlrc_share/data/wiki/split/` directory. When executed as

```
split_file.py input_file start_offset length
```

only lines in the file part starting at offset greater than `start_offset` and less than `start_offset + length` are printed (allowing the last line to end outside the file part).

The `split_file.py` and also `split_file.pm` can be used also as modules in your Python and Perl scripts. In that case, you can use the method

```
next_line(file, start_offset, length)
```

which sequentially returns lines in the file part (including newline characters), returning empty string at the end.

Exercises for using Sun Grid Engine

Wikipedia Data

In the `/dlrc_share/data/wiki/` there are following data:

- `cs-text/cswiki.txt`: Czech Wikipedia data (Sep 2009), file size 195MB, 124k articles.
- `en-text/enwiki.txt`: English Wikipedia data (Sep 2009), File size 4.9BG, 1.7M articles.

Both files are encoded in UTF-8 and contain one particle per line. Article name is separated by `\t` character from the article content.

For testing, medium and small variants of the Czech Wikipedia data is also available:

- `/dlrc_share/data/wiki/cs-text-medium/cswiki-medium.txt`: 16MB, 5307 articles
- `/dlrc_share/data/wiki/cs-text-small/cswiki-small.txt`: 72kB, 13 articles

Example Solution of a Simple Distributed Task using Bash

Consider simple example of producing sorted list of article names. The example solution consists of two sources. Both are available in the `/dlrc_share/data/wiki/example` directory.

The source `articles.sh` is the main driver for distributing work and collecting results:

```
#!/bin/bash

set -e

# Parse arguments
[ "$#" -ge 3 ] || { echo Usage: "$0 input outdir tasks [conc_tasks]" >&2;exit 1;}
input_file="$1"
output_dir="$2"
tasks="$3"
conc_tasks="$4"

# Check that input file exists and get its size
[ -f "$input_file" ] || { echo File $input_file does not exist >&2; exit 1; }
input_size=`stat -c%s "$input_file"`

# Check that output dir does not exist and create it
[ -d "$output_dir" ] && { echo Directory $output_dir already exists >&2; exit 1;}
mkdir -p "$output_dir"

# Compute file split sizes
split_size=$((1 + ($input_size / $tasks)))

# Run distributed computations
qsub -cwd -b y -sync y -o "$output_dir" -e "$output_dir" -V \
  -t 1-"$tasks" ${conc_tasks:+-tc $concurrent_tasks} \
 ./articles_distributed.sh "$input_file" "$split_size" "$output_dir"/articles.txt

# Merge all results
sort -m `seq -f "$output_dir/articles.txt.%g" 1 "$tasks"` \
  > "$output_dir"/articles.txt
rm `seq -f "$output_dir/articles.txt.%g" 1 "$tasks"`
```

The source `articles_distributed.sh` is the helper script executed distributively on the nodes:

```
#!/bin/bash

set -e

# Parse arguments
[ "$#" -ge 3 ] || { echo Usage: $0 input split_length output_file >&2; exit 1;}
input_file="$1"
split_length="$2"
output_file="$3"

# Parse SGE_TASK_ID and compute file offset
[ -n "$SGE_TASK_ID" ] || { echo Variable SGE_TASK_ID is not set >&2; exit 1; }
task="$SGE_TASK_ID"
input_offset=$(( ($task - 1) * $split_length ))
output_file="$output_file.$task"

# Run computation outputting to temporary file
tmp_file="`mktemp`"
trap "rm -f \"$tmp_file\"" EXIT

/dlrc_share/data/wiki/split/split_file.py "$input_file" "$input_offset" \
  "$split_length" | cut -f1 | sort > "$tmp_file"

# On success move temporary file to output
mv "$tmp_file" "$output_file"
```

Tasks

Solve the following tasks. Solution for each task is a source code processing the Wikipedia source data and producing required results, while utilizing distributed computation. The solution does not need to recover when one of the computation fails, but it should fail as a whole.

Task	*Points*	*Description*
`unique_words`	3	Create a list of unique words used in the articles. Convert them to lowercase to ignore case. Because the article data is not tokenized, use provided `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer`, which reads untokenized UTF-8 text from standard input and produces tokenized UTF-8 text on standard output. It preserves line breaks and separates tokens on each line by exactly one space.
`inverted_index`	4	Compute inverted index – for every lowercased word from the articles, compute ascending *(article id, ascending positions of occurrences as word indices)* pairs. In order to do so, number the articles using consecutive integers and produce also a list of articles representing this mapping (the article on line i is the article with id i; you can use the example `articles.sh`). The output should be a file with list of articles ordered by article id, and a file with one word on a line in this format: `word \t article_id \t space separated occurrences \t article_id \t space separated occurrences ...` Once again use provided tokenizer.
`wordsim_index`	4	In order to implement word similarity search, compute for each lemma with at least three occurrences all *contexts* in which it occurs, including their number of occurrences. Given N (either 1, 2 or 3) on the command line, the *context* of a lemma occurrence is N lemmas preceding it and N lemmas following it (ignore sentence boundaries). To compute the lemmas for a given article, use provided `/dlrc_share/data/wiki/lemmatizer/{cs,en}_lemmatizer`, which works just like the tokenizer – it reads untokenized UTF-8 text from standard input and produces tokenized and lemmatized UTF-8 text on standard output, each lemma separated by exactly one space. The output should be a file with one lemma on a line in the following format: `lemma \t context \t counts \t context \t counts ...`
`wordsim_find`	3	Let S be natural number given on the command line. Using the index created in `wordsim_index`, find for each lemma S most similar lemmas. The similarity of two lemmas is computed using *cosine similarity* as $\frac{C_A \cdot C_B}{\lvert C_A \rvert \cdot \lvert C_B \rvert}$, where C_L is a vector of occurrences of lemma L contexts. The output should be a file with one lemma on a line in the following format: `lemma \t most similar lemma \t cosine similarity \t 2. most similar lemma \t cosine similarity ...`