Homework 5

Assignments: All parts are to be solved individually (turned in electronically, no ZIP files or tarballs, written parts in ASCII text, NO Word, postscript, PDF etc. permitted unless explicitly stated). All filenames must be exactly what is requested, capitalization included.

Please use the ARC cluster for this assignment. All programs have to be written in JAVA/C, translated with javac/mpicc/gcc and turned in with a corresponding Makefile.

1. (50 points, individual problem) Write a program that computes a TF-IDF value for each word using the same TFIDF calculations as HW4 in a corpus of documents using Spark. The Spark API is here. The skeleton code provides a starter point for the implementation. You will see that the organization is slightly different from the MapReduce implementation but the (key, value) pairs are similar. Use the DEBUG variable in the skelton code to print intermediate output.

   Since Spark uses HDFS as well, we need to set it up similar to HW4, except some extra variables for Spark:

   ```
   srun -N4 -popteron --pty /bin/bash
   tar xvf TFIDF.tar
   source spark-hadoop-setup.sh &> setup_output.txt
   hdfs dfs -put input /user/UNITYID/input


   javac TFIDF.java
   jar cf TFIDF.jar TFIDF*.class
   spark-submit --class TFIDF TFIDF.jar input &> spark_output.txt
   grep -v '^2018\|^(\|^-' spark_output.txt > output.txt
   diff -s solution.txt output.txt
   ```

   Hints:

   - During compilation, you may ignore any "unchecked or unsafe operations warnings"
   - Be sure to check `spark_output.txt` for any errors
   - The `grep` commmand filters out the `INFO` statements from the output and also filters out all of the `DEBUG` print statements. You can use `-v '^2018'` to filter out only the `INFO` statements and keep the `DEBUG` print statements.
   - An example **input** directory and **solution.txt** are provided which are the same as from HW4. You can use these to check your work, but I will be grading you on a different input set.
   - You should not need to change any of the provided skeleton code (you can if you want). You should only need to add to it.

   In your **p1.README**, explain how you implemented each of the steps of the TFIDF algorithm.

   Turn in **p1.README** and **TFIDF.java**.

2. (10 points, group problem) Assess the CoMD application (a molecular dynamics reference application by LANL) in terms of its checkpointing overhead under DMTCP. Familiarize yourself with DMTCP checkpoint/restart using quickstart, and full documentation.

- Download and make the DMTCP code and the CoMD code:

```
srun -popteron --pty /bin/bash
git clone https://github.com/dmtcp/dmtcp.git
cd dmtcp
./configure
make
export PATH=$PATH:$PWD/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/lib/dmtcp
cd ..
tar xvzf CoMD.tar.gz
cd CoMD/src-mpi
make
cd ../bin
```

- We will run 3 cases:
  A: CoMD
  B: CoMD+checkpoint
  C: CoMD+restart

- Run the CoMD code without/with checkpointing:

```
time ./CoMD --nx 40 --ny 40 --nz 40                     #Case A
time dmtcp_launch -i 40 ./CoMD --nx 40 --ny 40 --nz 40  #Case B
```

  Case A should take ~53 seconds. The `-i 40` flag indicates that DMTCP will automatically checkpoint every 40 seconds. Since our code only runs for ~53 seconds, this means that only one checkpoint will be written. After 40 seconds, DMTCP will write a checkpoint file and will also write a restart script to restart the CoMD code from the checkpoint. Compare the "`real`" times of case A and B in order calculate the overhead of one checkpoint.

- Restart the CoMD code from the checkpoint:

```
time ./dmtcp_restart_script.sh  #Case C
```

  Since the checkpoint was written after 40s of execution, we can calculate the total execution time of case C by adding 40s to the "`real`" time of case C. Note that this time does not take into account the time it took to write the checkpoint. Compare this time with the "`real`" time for case A in order to calculate the overhead of one restart.

- Discuss your results in the **p2.README** file. Specifically:
  - What is the overhead of performing one checkpoint?
  - What is the overhead of one restart?
  - How to the "`real`" times for each case compare to the execution time output by the CoMD code?
  - You should see a very large execution time reported by the CoMD code in case C. Why?

Turn in: **p2.README**

3. (40 points, group problem) Familiarize yourself with "persistent memory allocation" (PERM) using this page.
   - Download/configure/build PERM:

```
srun -n16 -popteron --pty /bin/bash
wget --no-check-certificate https://computation.llnl.gov/projects/memory-centric-architectures/download/perm-je-0.9.7.tgz
tar xzvf perm-je-0.9.7.tgz
cd perm-je-0.9.7/
./configure
make
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/lib
cd ..
```

     The `make` step will throw a lot of warnings. It is okay to disregard these warnings.

   - Download this example program which uses PERM. This example program runs for 20 iterations and checkpoints itself every iteration. It has an array called **values** which is initially set to all zeros. Each iteration, the element at that iteration's index is set to 1. Thus the sum over the entire array should always equal the iteration number.

```
tar xvf example-perm.tar
make
./example-perm
# kill it after it passes a few iterations
[Ctrl+C]
# restart it
./example-perm -r
```

     You should see that the program picks up where it left off when you killed it!

   - Extend the Lake application from V1 of HW2 (serial, CPU version) to use persistent memory allocation with the PERM library.
     - **Use default valus of npoints = 128, npebs = 8, and end_time = 1.0, instead of taking inputs from the command line**. Due to the extra overhead of backing up every iteration, these changes should make the execution time stay around 40s.
     - You can copy-paste from the provided example program to build **lake-perm.c** around that.
     - The example **Makefile** has a target named "`lake`" which you should use to compile your code.
     - You should be able to kill/restart your code as many times as you want
     - The "Timestep" should always pick up where it left off!
     - The output should be the same no matter if you kill/restart your code or not
     - There may be some precision errors, but the images created by **heatmap.gnu** should look exactly the same
     - For grading, I will be running your lake code once through without killing it. Then I will run it and kill/restart it one or more times. Finally I will compare the results from both runs.
     - I will run something similar to the following:

```
make lake
./lake-perm
mv lake_f.dat lake_1.out
./lake-perm
[Ctrl+C]
./lake-perm -r
[Ctrl+C]
./lake-perm -r
mv lake_f.dat lake_2.out
gnuplot heatmap.gnu
```

- Discuss your solution in the **p3.README** file. Specifically:
  - What data structures did you have to place in persistent memory?
  - What is the overhead of checkpointing?

Turn in **lake-perm.c** and **p3.README**

Peer evaluation: Each group member has to submit a peer evaluation form.

What to turn in for programming assignments:

- commented program(s) as source code, comments count 15% of the points (see class policy on guidelines on comments)
- Makefiles (if required)
- test programs as source (and input files, if required)
- README (documentation to outline solution and list commands to install/execute)
- in each file, include the following information as a comment at the top of the file, where "username" is your unity login name and the single author is the person who wrote this file:

```
Single Author info:

username FirstName MiddleInitial LastName
```

How to turn in:

Use the "Submit Homework" link on the course web page. Please upload all files individually (no zip/tar balls).

Remember: If you submit a file for the second time, it will overwrite the original file.