High Throughput Docking on Helium Cluster At University of Iowa Ashley Spies Group

Kenneth Tussey

July 23, 2012

1 Getting Started on Helium

Helium is the University of Iowa scientific computing cluster. There are a total of 3748 functional computing cores as of July 23, 2012. Groups are assigned a guarunteed number of cores based on their contribution to the Helium project. Ashley Spies group has 108 computing cores as of July 23, 2012 as part of the *AS* queue. Those 108 are ours to use whenever we please - however, when not in use, they become available as part of the *all.q*, where anyone can use them. If a job happens to be running on one of our processors, and we submit a job to our queue, the offending process will be terminated, allowing us to run our job.

To get started with Helium, you need a Helium account. You can request one at `http://hpc.uiowa.edu/helium-access-agreement`.

Once you have an account, you can login to Helium via ssh.

```
$ ssh -CX username@helium.hpc.uiowa.edu
```

The -CX option allows you to modify text files using gedit. Now that you are logged in to Helium, there are a few important things to know about the environment, backups, etc.

1.1 Modules

Modules are used to change environment variable settings. Modules are loaded into the environment by

```
$ module load modulename
```

where modulename is swapped with whatever module you want to load.
So, if you want to load the Intel libraries into the environment, you would type

```
$ module load intel
```

Common modules in Helium	
openmpi_gnu	Sets MPI variables to point to GNU MPI implementation
openmpi_intel	Sets MPI variables to Intel MPI implementation
intel	Loads variables specific to Intel libraries
python26	Uses Python 2.6.4 interpreter and libraries
python27	Uses Python 2.7 interpreter and libraries

Table 1: Some common helium modules.

If you want to see what modules are loaded, you would use

```
$ module list
```

and if you want to see what modules are available, you would use

```
$ module avail
```

More information is located in the Helium documentation. The good news is that you shouldn't have to worry about the modules to perform the high-throughput docking, but it is good to know how to use modules in case you ever want to tweak something.

1.2 Backups

All home directories are 'backed up' in a sense. Snapshots are taken at specific intervals and saved in a hidden directory in the user's home directory. To get acess to your snapshots, you must type

```
$ cd ~/.zfs
```

This will take you into the .zfs directory. Note that if you list the home directory, it will not appear. From the .zfs directory, you will be able to look at old snapshots to recover your data if need be.

1.3 Shared Group Directory

Our group (AS) has a shared directory that all of us can access and read/write files to. It is located in /Groups/AS/. The first thing you want to do when you login to your helium account is create a link in your home folder to this directory.

```
$ ln -s  /Groups/AS   ~/shared
```

You may call it whatever else you like, but take care that you will need to modify some rc files in order make everything work. After you link the shared directory to your home folder, copy over some rc files into your home directory.

```
$ cp  ~/shared/env/.*   ~
```

This copies a .bashrc and a .chemenv.bashrc file into your home directory, which sets up your user environment variables specific to high-throughput docking. Check to make sure that these are copied over by

```
$ ls -a
```

You should see both files in your home directory. Go ahead and source the .bashrc file.

```
$ source  ~/.bashrc
```

2 Setting Up Docking Directory Structure

Once you have the Helium environment setup, we can prepare the docking directory structure. Please not that I have gone to great lengths to minimize the amount of work needed to setup a docking. This will become evident as you read on.

2.1 Ligand Directory

First, start out by making a directory which will contain all of your projects. I like 'work', but you may use what you like. For this tutorial, I will use work.

```
$ mkdir  ~/work && cd ~/work
```

Now, make a tutorial directory.

```
$ mkdir tutorial && cd tutorial
```

Now, create a directory for the ligand archives. The ligand archives are gzipped archives containing all the individual ligand pdbqt files for a given database.

```
$ mkdir ligand-archives
```

Please name this directory verbatim, or else you will get problems with the docking scripts. The script directly references the 'ligand-archives' directory. Now, you will pick a database to screen. The databases that are available are located in ~/shared/ligands/by-vendor. For this tutorial, we will use the analyticon database, because it is small and shouldn't take very long to perform the entire run. So link or copy (link is prefereble for large databases) the archives into the ligand-archives directory.

```
$ ln -s  ~/shared/ligands/by-vendor/analyticon/*.tar.gz    ligand-archives/
```

Check the ligand-archives folder to make sure you succesfully linked the ligand archives.

```
$ ls ligand-archives
```

The links will show up in red if they are broken, and blue if they are competent. This is all that you have to do to setup the ligands part of the directory.

2.2 Receptor Directory

Later, I will write into the tutorial how to create the necessary receptor files using Autogrid, but for the purpose of this tutorial, I will use a pregenerated set of receptor files. In this case, it will correspond to the active site of B. Subtilus Glutamate Racemase (PDB: 1zuw).
Make a directory to contain all of the receptor files.

```
$ mkdir receptor
```

Now, link all of the sample receptor files into the receptor directory.

```
$ ln -s  ~/shared/receptor/*    receptor/
```

3 Running the Docking

There are three main scripts that run all of the docking. They are called 'run_parallel_dpf4.sh', 'run_parallel_autodock4.sh', and 'convert_docking_results.sh', and are run in that order. The appendix contains the script code for reference. These scripts are located in ~/shared/usr/bin.

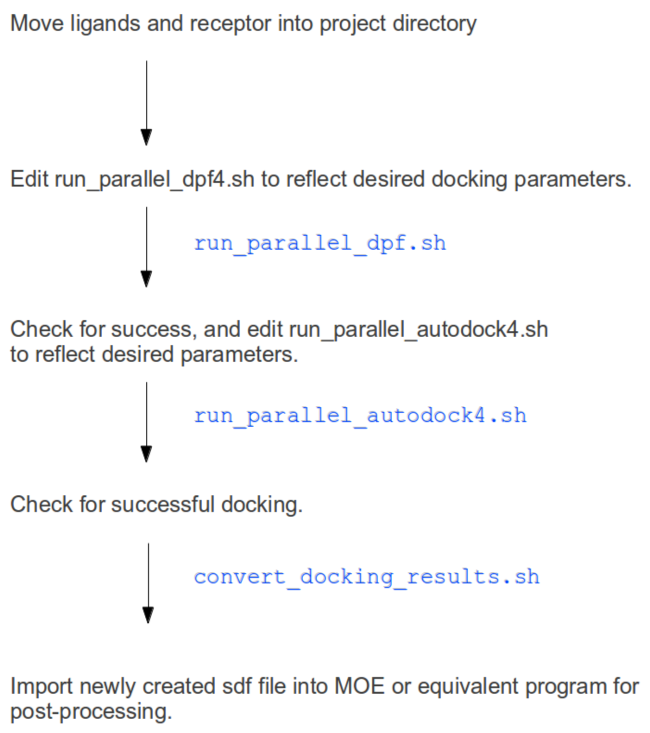

Move ligands and receptor into project directory
Edit run_parallel_dpf4.sh to reflect desired docking parameters.
run_parallel_dpf.sh
Check for success, and edit run_parallel_autodock4.sh to reflect desired parameters.
run_parallel_autodock4.sh
Check for successful docking.
convert_docking_results.sh
Import newly created sdf file into MOE or equivalent program for post-processing.

Figure 1: Flowchart for docking

3.1 run_parallel_dpf4.sh

This script unpacks the pdbqt archives that you linked to the ligand-archives directory, creates a special folder for each ligand, link the receptor and copy the ligand into the folder, and then create a dpf file that autodock will use perform the docking. Once each folder is created, it is then packed back up into a .tar.gz archive consisting of multiple ligand folders. This may seem counterproductive, since we have to unpack it again to perform the docking, but it is critical to compartmentalize the ligands like this to avoid directory overflow (when we start getting 1000s of folders containing 30 files each, the read/write access becomes painfully slow). We take the hit in overhead, but we will recover it because this script will do this entire procedure in parallel.

In the beginning of this file, you will see about 9 variables. These variables can be changed before running the script. It is important to think about what numbers you want for ‘ga_num_evals‘, ‘ga_pop_size‘, and ‘ga_runs‘. These are the genetic algorithm controls, and they will greatly influence the accuracy and speed of the docking process. For the purpose of this tutorial, changing it is not necessary. However, for a production screen, you may want to decrease the ‘ga_num_evals‘ to around 100K, and ‘ga_num_runs‘ from 10 to less than 5. They can be changed by editing the file run_parallel_dpf4.sh. For the purposes of this tutorial, we should change these settings to as low as possible for the sake of brevity. Change the variables in run_parallel_dpf4.sh (this file is in ~/shared/usr/bin - use a text editor to change these) to match Table 2.
Altogether, this script should take around 10 minutes to run for the 22,000 compounds in the analyticon

Settings for docking tutorial	
ga_num_evals	50,000
ga_runs	1

Table 2: Docking parameters for tutorial.

library. For every 108,000 compounds, this script takes around 10 minutes to run (on 108 processors). The script will divide the 22K compounds into 1000 compound segments, and perform the dpf4 creation and archiving ion parallel on 23 processors. This creates 23 folders in the 'ready-for-docking' folder that is newly created, with around 1K compounds in each.
So for every 1M compounds to process, expect about 1.7 hr to run this phase.

```
$ vim  ~/shared/usr/bin/run_parallel_dpf4.sh
```

You may use whatever editor you want (nano is a good choice). Also, take care of anything else that is running on our queue at the time. Check the queue with

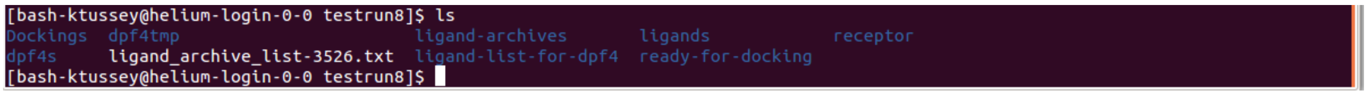
```
$ qstat -g c
```

If all 108 cores are not available, then change the variable 'number_of_processors_anticipated' to how ever many are available. Please refer to the Appendix Section 1 to see an example of this script.

Now, run the script.

```
$ run_parallel_dpf4.sh
```

When you are finished, you're directory structure should now look something like this.



```
[bash-ktussey@helium-login-0-0 testrun8]$ ls
Dockings  dpf4tmp                    ligand-archives      ligands            receptor
dpf4s     ligand_archive_list-3526.txt  ligand-list-for-dpf4  ready-for-docking
[bash-ktussey@helium-login-0-0 testrun8]$
```

Figure 2: Snapshot of directory

You should also check the ready-for-docking directory, and make sure that there are 23 files in there.

3.2 run_parallel_autodock4.sh

The next step is to run the script that unpacks the archives in the 'ready-for-docking' directory into another directory, and calls the parallel autodock program to dock those compounds. This script also takes the docking results, and puts the dlg file (the file containing the docked poses and docking specifics) into an archive.
You can see this script in the appendix. There are 3 parameters that can be changed in the beginning of this script, the most important one being how many processors you would like to use. Again, check the AS queue to see how many you have, and run on all 108 if it is possible. If all 108 are available, than you do not need to modify this script.
Now, run the parallel docking.

```
$ run_parallel_autodock4.sh
```

Now, this script will take a while to finish. This time is very sensitive to the nature of the library being docked. Libraries with compounds containing many rotatable bonds will take noticeably longer to run than libraries with a low number of rotatable bonds. Increasing the number of evaluations and runs will cause the job to run longer. Balancing all of these variables is important to running a succesful, yet timely screen. When this is finished, go into the 'docking_results' directory, and check that there are results .tar.gz files. There should be 23 archives. You can check them with the following script.

```
#!/bin/bash
num_results=0
for i in docking_results
do
  num_results=$((num_results+$(zcat $i | tar -tf - | wc -l)))
done
echo $num_results
```

It should be around 22K, depending on how many ligands autodock choked on. In this tutorial, I successfully docked 22,446 compounds in 72 minutes. 27 minutes were spend in overhead, and 45 minutes in docking. You can check these timings in the file timings.txt. The overhead time seems high for this tutorial, but when we do production runs using more intensive docking parameters, this will be much smaller proportionally.

3.3 convert_docking_results.sh

Now that autodock has finished, you have a nice set of docked structures. Unfortunately, they are locked up in a dlg file, which no commercial or academic program (to my knowlege) has taken the time to include in their supported formats. Luckily, with a few scripting tricks, we can extract the relevant pdbqt files from the dlg file, and then convert them to sdf using a custom program that relies on OpenBabel C++ bindings. To convert the docking results into an SDF format, run

```
$ convert_docking_results.sh
```

This script currently processes the results serially. However, it is trivial to parallelize. If this becomes too cumbersome to run, it can be parallelized to 100x speedup. For 22K compounds, it takes around 10 minutes to run.
When this script is finished, you can find the docking results in docking_results/final_results.sdf.
You can take this file, and import into a MOE database, or use OpenEye's vina to view the results. The receptor is found in receptor/.

4 Appendix

4.1 run_parallel_dpf4.sh

```
#!/bin/bash


# Some control numbers for the script, does not need to be changed. Can be if
# wanted, for optimization purposes

sge_name_for_script=prepdpf4
number_of_processors_anticipated=108
max_ligand_list_size=2000
min_ligand_list_size=1000
limit_num_ligands=100000
ga_num_evals=500000
ga_pop_size=150
ga_runs=10
sge_queue_custom=AS


T="$(date +%s)"
# Need to know how many ligands we will be dealing with

let global_num_ligands=0
let global_total_num_ligands=0

if [ $(($(find ligand-archives -name "*.tar.gz" | wc -l))) -gt 0 ]; then sleep 0
else
  echo "No ligand archives found. Exiting"
  exit
fi



echo "Getting total ligand count..."
for i in ligand-archives/*.tar.gz
do

  global_num_ligands=$(zcat $i | tar -tf - | wc -l)
  global_total_num_ligands=$((global_total_num_ligands+global_num_ligands))
done
echo "Total ligand count: $global_total_num_ligands"




# Function that creates a list of ligand files, divied up by how many processors and compounds
    there are.

function prepare_for_dpf4 () {

  if [ $((total_num_ligands/number_of_processors_anticipated)) -gt $max_ligand_list_size ]; then
    let ligand_list_size=$max_ligand_list_size
  elif [ $((total_num_ligands/number_of_processors_anticipated)) -lt $min_ligand_list_size ]; then
    let ligand_list_size=$min_ligand_list_size
  else
    let ligand_list_size=$((total_num_ligands/$number_of_processors_anticipated+1))
  fi

  cd ligands
  let counter_l=0
  let counter_j=1

  if [ -d ../ligand-list-for-dpf4 ]; then rm ../ligand-list-for-dpf4/*
  else
    mkdir ../ligand-list-for-dpf4
  fi

  # put each ligand file's name into a list of ~2K compounds,
  # depending on number of procs and number of ligs
  for i in *.pdbqt
  do
    echo $i >> ../ligand-list-for-dpf4/ligand-list-for-dpf4-${counter_l}
    if [ $(($counter_j % $ligand_list_size)) -eq 0 ]; then
      counter_l=$((counter_l+1))
    fi
    counter_j=$((counter_j+1))
  done

  cd ../

}


# This function creates a shell script that will be submitted to SGE to process the ligands into
# folders which are meaningful to the parallel autodock. For each list of ~2K ligands produced in
# the function prepare_for_dpf4, a shell script is spawned that will iterate through the list. Each
# of these shell scripts is submitted to whichever queue is specified in the control portion of
    this script.

function run_dpf4 () {



  if [ -d dpf4tmp ]; then rm dpf4tmp/*
  else
    mkdir dpf4tmp
  fi

  basename=$PWD

  if [ ! -d ready-for-docking ]; then mkdir ready-for-docking; fi
  if [ ! -d Dockings ]; then mkdir Dockings; fi
  if [ ! -d dpf4s ]; then mkdir dpf4s; fi

  cd ligand-list-for-dpf4
  for i in ligand-list-for-dpf4-*
  do
    touch ../dpf4tmp/$i.sh
    echo "#!/bin/bash" >> ../dpf4tmp/$i.sh
    echo "module load intel" >> ../dpf4tmp/$i.sh
    echo "module load python26" >> ../dpf4tmp/$i.sh
    echo 'random_folder=$RANDOM-$RANDOM' >> ../dpf4tmp/$i.sh
    echo 'mkdir Dockings/docking-${random_folder}' >> ../dpf4tmp/$i.sh
    echo -ne 'for j in $(cat ' >> ../dpf4tmp/$i.sh
    echo -ne "ligand-list-for-dpf4/$i" >> ../dpf4tmp/$i.sh
    echo ');' >> ../dpf4tmp/$i.sh
    echo 'do ' >> ../dpf4tmp/$i.sh
    echo '  ligand_name=$(basename $j .pdbqt)' >> ../dpf4tmp/$i.sh
    echo '  receptor_name=$(basename receptor/*.pdbqt .pdbqt)' >> ../dpf4tmp/$i.sh
    echo -ne '  prepare_dpf42.py -l ligands/${j} -r receptor/1zuw.pdbqt -o dpf4s/${ligand_name}_${
        receptor_name}.dpf' >> ../dpf4tmp/$i.sh
    echo " -p ga_num_evals=$ga_num_evals -p ga_pop_size=$ga_pop_size -p ga_run=$ga_runs -p rmstol
        =2.0" >> ../dpf4tmp/$i.sh
    echo '  mkdir Dockings/docking-${random_folder}/${ligand_name}_${receptor_name}' >> ../dpf4tmp/
        $i.sh
    echo '  mv dpf4s/${ligand_name}_${receptor_name}.dpf Dockings/docking-${random_folder}/${
        ligand_name}_${receptor_name}'  >> ../dpf4tmp/$i.sh
    echo -ne "  ln -s $basename/" >> ../dpf4tmp/$i.sh
    echo 'receptor/* Dockings/docking-${random_folder}/${ligand_name}_${receptor_name}'  >> ../
        dpf4tmp/$i.sh
    echo '  mv ligands/${ligand_name}.pdbqt Dockings/docking-${random_folder}/${ligand_name}_${
        receptor_name}'  >> ../dpf4tmp/$i.sh
    echo 'done' >> ../dpf4tmp/$i.sh
    echo 'cd Dockings/docking-${random_folder}' >> ../dpf4tmp/$i.sh
    echo 'tar -czf ../../ready-for-docking/docking_folder-${random_folder}.tar.gz * --remove-files'
        >> ../dpf4tmp/$i.sh
    echo 'cd ../../' >> ../dpf4tmp/$i.sh
    echo 'rm -rf Dockings/docking-${random_folder}' >> ../dpf4tmp/$i.sh

  done

  cd ../


  # Submit each shell script to the cluster as an independent, serial job. When the lustre file
      system becomes
  # available again, we will change the above script to output to lustre (or create a symlink from
      the folder
  # dpf4s to some folder on lustre.
  for i in dpf4tmp/*.sh
  do

    qsub -V -N $sge_name_for_script -cwd -j y -o /nfsscratch/$sge_name_for_script.log -q
        $sge_queue_custom $i
  done


  # This line queries the queue, looks for the job named $sge_name_for_script,
  # and looks every 20 seconds until there are no
  # more jobs with then name $sge_name_for_script. The function then exits.
  echo "Running jobs on cluster..."
  sleep 10
  job_waiter=$(qstat | grep $sge_name_for_script)
  while [ "$job_waiter" ]; do sleep 20; job_waiter=$(qstat | grep $sge_name_for_script); done

  echo "Finished!"

}





let number_of_ligands=0
let total_number_of_ligands=0
let total_docking_archives=1
let ligand_counter_for_ouput=0

ligand_archive_list_notifier=$RANDOM


if [ -d ligands ]; then
  rm -rf ligands/*
else
  mkdir ligands
fi


for i in ligand-archives/*.tar.gz
do
  echo $i >> ligand_archive_list-${ligand_archive_list_notifier}.txt
done

# Typically, each archive will house 100K compounds in pdbqt format. Typically, we have around 50
    processors
# free at any given time. This means that the best way to divy up each archive is to split it into
    subsets
# containing ~2K ligands apiece. The time it takes to process 2K compounds is around 10 minutes.

exec 3<&0
exec 0<ligand_archive_list-${ligand_archive_list_notifier}.txt

while read line
do


  # We will fill our ligand directory up to 100K ligands, but no more so that file access isn't
      slowed down
  if [ $total_number_of_ligands -lt $limit_num_ligands ]; then
    echo "Extracting $line..."
    tar -xzf $line -C ligands
    number_of_ligands=$(zcat $line  | tar -tf - | wc -l)
    ligand_counter_for_output=$((ligand_counter_for_output+number_of_ligands))
    total_number_of_ligands=$((total_number_of_ligands+number_of_ligands))
    #sleep 0
  else
    echo "Preparing dpf4s for $ligand_counter_for_output out of $global_total_num_ligands ligands"
    prepare_for_dpf4
    run_dpf4
    echo "Extracting $line..."
    tar -xzf $line -C ligands
    number_of_ligands=$(zcat $line  | tar -tf - | wc -l)
    ligand_counter_for_output=$((ligand_counter_for_output+number_of_ligands))
    total_number_of_ligands=$number_of_ligands
    total_docking_archives=$((total_docking_archives+1))
  fi


done


# Finally, perform the last run through of ligands for processing.
prepare_for_dpf4
run_dpf4


exec 0<&3

#echo "ligands/${LIGAND_FILENAME}"
#echo "dpf4s/$(basename $LIGAND_FILENAME .pdbqt)"


if [ ! -d logs ]; then mkdir logs; fi
rm -rf dpf4tmp
rm -rf dpf4s
rm -rf ligand_archive_list-${ligand_archive_list_notifier}.txt
rm -rf ligand-list-for-dpf4

T="$(($(date +%s)-T))"

echo "Processed $global_total_num_ligands ligands" > timings.txt
echo "Timing in minutes to process: $((T/60))" >> timings.txt
```

4.2 run_parallel_autodock4.sh

```
#!/bin/bash

num_processors_for_docking=108
sge_name_for_script=ad42_mpi
sge_name_for_queue=AS

let overheard_time=0
let docking_time=0
let total_ligands=0
let log_indicator=$RANDOM
let counter=0
ls ready-for-docking > undocked_list.txt
let total_files=$(cat undocked_list.txt | wc -l)

for i in $(cat undocked_list.txt)
do

  total_ligands=$(($total_ligands + $(zcat ready-for-docking/$i | tar -tf - | grep dpf | wc -l)))
  echo "Processing ligands up to $total_ligands compounds"

  echo "Processing $counter of $total_files"
  counter=$((counter+1))
  T="$(date +%s)"
  basename=$(basename $i .tar.gz)
  if [ ! -d Dockings ]; then mkdir Dockings; fi
  if [ ! -d docking_results ]; then mkdir docking_results; fi
  tar -xzf ready-for-docking/$i -C Dockings

  ls Dockings/  > ligand_list.txt

  echo '#!/bin/bash' > cluster_sub_script.sh
  echo "module load intel" >> cluster_sub_script.sh
  echo "module load openmpi_intel" >> cluster_sub_script.sh
  echo 'mpirun autodock-intel-serial-mpi $PWD/ligand_list.txt $PWD/Dockings/ $PWD/logs/
      default_seed reuse_maps' >> cluster_sub_script.sh

  T="$(($(date +%s)-T))"
  overhead_time=$((overhead_time+T))

  T="$(date +%s)"
  qsub -V -N $sge_name_for_script -pe orte $num_processors_for_docking -cwd -j y -o /nfsscratch/
      autodocklogs-${log_indicator}.log -q $sge_name_for_queue cluster_sub_script.sh


  echo "Running jobs on cluster..."
  job_waiter=$(qstat | grep $sge_name_for_script)
  while [ "$job_waiter" ]; do sleep 20; job_waiter=$(qstat | grep $sge_name_for_script); done

  echo "Finished!"
  T="$(($(date +%s)-T))"
  docking_time="$((docking_time+T))"


  T="$(date +%s)"
  echo "Packing docking logs..."
  mkdir tmp_dlglinks

  for j in $(find $PWD/Dockings -name "*.dlg")
  do
    ln -s $j tmp_dlglinks
  done

  cd tmp_dlglinks
  tar -czhf ${basename}-results.tar.gz * --remove-files
  mv ${basename}-results.tar.gz ../docking_results
  cd ../
  rm -rf tmp_dlglinks

  echo "Finished!"

  echo "Cleaning up docking directory..."

  rm -rf Dockings/*

  echo "Finished!"

  T="$(($(date +%s)-T))"
  overhead_time="$((overhead_time+T))"

  echo $i >> finished_docking_list.txt
  sed -i '1d' undocked_list.txt

done

total_time="$((docking_time+overhead_time))"

echo "Docked $total_ligands ligands in a total of $((total_time/60)) minutes" >> timings.txt

echo "Overhead time = $((overhead_time/60)) minutes" >> timings.txt
echo "Docking time = $((docking_time/60)) minutes" >> timings.txt
```

4.3 convert_docking_results.sh

```
#!/bin/bash


counter=1;
total=$(ls docking_results | wc -l)
for i in docking_results/*.tar.gz
do


  if [ ! -d docking_results/tmp ]
   then mkdir docking_results/tmp
   else
    rm docking_results/tmp/*
    echo "Cleaning up..."
  fi




  echo "Converting $counter of $total docking results..."

  echo "Untarring docking results..."
  tar -xzf $i -C docking_results/tmp

  echo "Converting dlgs to sdfs..."
  dlgtosdf -i docking_results/tmp/*.dlg >> docking_results/final_results.sdf

  mv $i processed_results

  counter=$((counter+1))
done
```