### **IGM OVERVIEW: INPUT, OUTPUT, STRUCTURE**

### 

### 

### Expected Input Files:

* Pipeline configuration: **igm-config.json (**This json config file should include all input parameters and data files like hic input matrix etc);
* Data files (.hcs, .txt, etc.) should also be available in the appropriate folder(s);
* Lammps executable file path needs also to be correct.

All the parameters of the model (e.g., how many and which kinds of experimental data are being incorporated, the optimization parameters, the number of structures, the name of relevant files) are listed in a **json** file (a fancy python dictionary, which can be viewed and edited using any text editor such as VIM). Such a file can be created manually (by editing an existing template) or by using the igm graphic interface.n

Most of the parameters do not need fine-tuning, but others do. Specifically, some entries require particular care: **segmentation** (i.e., degree of coarsening of the model), **population size** (i.e., number of structures to be generated), **optimization/kernel\_opts/lammps\_executable** (this path needs to point to the executable version of LAMMPS which was installed together with IGM in the first place, this you generally need to adjust only the first time running the code), **parameters** (change names to taste), **parallel/controller** (change this to ipyparallel or serial if simple test on a single core).

A comprehensive list of all the configuration file options (and their brief description) can be found on the GitHub @ i[**gm**](https://github.com/alberlab/igm)**/**[**igm**](https://github.com/alberlab/igm/tree/master/igm)**/**[**core**](https://github.com/alberlab/igm/tree/master/igm/core)**/**[**defaults**](https://github.com/alberlab/igm/tree/master/igm/core/defaults)**/**config\_schema.json.

### Expected Output Files:

Final output structures: **igm-model.hss** (default), or specified in **optimization/structure\_output** in json configure file. This file can be manipulated using alabtools.HssFile(“igm-model.hss”,’r’). The coordinates are stored in 3D array in dimensions:[nbeads, nstruct, 3]. igm-model.hss.T is transposed, with dimensions: [nstruct, nbeads, 3].

If **optimization/keep\_intermediate\_structures** is set to true, there will be .hss files for each step named like **igm-model.hss.sigma\_#\_iter\_#.#######hash\_value####.hss**

Runtime log: **igm.log** file embedded with colorcode. Use cat to see colors.

Evaluation folder: if **restraints/Hi-C/run\_evaluation\_step** is set to true, Hi-C evaluation step is executed to plot matrix comparison plots for each chromosome. If this step is run, it is recommended that multiple cores are requested on the same node (multi-thread),

Temp files (regularly removed, at the end of each step): in folder **tmp**/

**tmp/actdist/**: default for Hi-C restraints temp files (activation distance file)

**tmp/opt/**: default for optimization restraints files and lammps trajectories

tmp/damid\_actdist/: default for damid distance files.

### Running scheme

The main script is igm\_run.sh whose structure reads:

---------------------------------------------------------------------------

import igm (among others)

def run\_pipeline(config):

…….

# start the run as a subprocess

print(‘[CONTROL] Starting IGM:’)

P = multiprocessing.Process(target = run\_pipeline, args=(cfgfile,))

p.start()

p.join()

# exiting calculation smoothly

print(‘[CONTROL] Exiting’)

---------------------------------------------------------------------------

The run\_pipeline function consists of the following steps:

1. igm-run creates hidden file .igm-pid.txt to create the process id. This file indicates that the igm pipeline is running.
2. **igm.Config()**: Configuration is read from igm-config.json[[1]](#footnote-0)
3. **igm.Preprocess()**: Preprocess genome, index and allocate disk space for genome structures (basically calls utilities from alabtools)
4. Generate starting coordinates with random coordinates or from another hss file.
5. If start with random coordinates, **igm.RandomInit()** is run and simulation with chain restraints **igm.RelaxInit()** is run.

**Loop** (over decreasing values of threshold values):

# ----

1. Assignment steps:

This step reads in structures (.hss file) optimized in the previous step, and calculates the new assignment criteria for the current step. For example, HiC restraint is assigned using distance threshold called activation distance, which needs to be calculated. Following steps are run in sequence if corresponding data is provided.

* 1. **igm.ActivationDistanceStep()** for HiC restraints
  2. **igm.FishAssignmentStep()** for FISH restraints
  3. **igm.SpriteAssignmentStep()** for Sprite restraints
  4. **igm.DamidActivationDistanceStep()** for DamID restraints

1. Modeling step:

**igm.ModelingStep()**: Runs the script generation for lammps. Build model and effecitvely apply the restraints computed from the assignment steps. Run lammps simulation and collect the structure files, violation score etc.

1. Evaluation step (optional):

Contact matrices are generated (using multiple cores). Plot matrix comparison plots for each chromosome (**igm.HicEvaluationStep()**).

1. Violations of restraints for each category in Assingment step (a, b, c or d) are assessed to see if the expectations are met. Decide if the loop continues (using parameters in cfg[“runtime”]).

# ---

**Done**

### Remark: checkpointing and restarting

A production run also creates a database file with extension **.sqlite** (filename can be edited in the configuration file) on-the-fly, which keeps track of the progress status of the calculation (checkpointing). If such a file from a previous run is in the main folder, a new calculation looks into it (first thing first) and picks up where the other calculation left off. For instance, if the previous run exceeded walltime during a mapping step, this is recorded and written to the .sqlite file, and the new run starts from that point, writing to the log file that “step XXX is already completed, moving on”. The .sqlite file should be removed ONLY if a new fresh calculation is to be submitted. The job tracking is carrried out in the **igm/igm/core/job\_tracking.py.**

Things were cleverly designed in such a way that if code stopped during a mapping step, the indices of the optimized configurations are tracked, so that a restarting run wouln’t try to optimize them again. No wasting of computing resources.

### Post-IGM Conformational Analysis

A standard zero-th order analysis procedure (achronym QC) has been coded and added to the IGM software in such a way that the output structure population (.hss) can be processed and the essential information is extracted and saved to a folder named **QC\_igm-model**.

The analysis code can be found in **igm/bin/igm-report[[2]](#footnote-1)** and takes as an input the hss filename. This code uses all the routines that are defined in **img/igm/report**, and heavily relies on the alabtools python module.

A number of subfolders are created (**violations, radius\_of\_gyration, shells, radials, damid, matrix\_comparison, images**), each populated by the corresponding files. A report.html file is also output. As a comparison term, 5000 configurations generate approximately 200MB of analysis report.

***DETAILS OF THE INPUT DATA***

Each kind of experimental data to be incorporated in the population modeling needs to be preprocessed and written to a file, whose format and content is compatible with the way IGM was designed. Here we cover a few details.

### -Hi-C ---------------

The input data (input\_matrix) consists of a (haploid) contact probability matrix, which is saved into a .hcs file, and is read in by IGM using the alabtools python package (see ActivationDistanceStep.py). The (i,j) entry indicates the probability that a contact is formed between i and j locii; clearly, this depends on how a contact cutoff between locii is defined.

A (i,j) contact is enforced in a structure, by adding a proper harmonic restraint between i and j. An iterative procedure is employed, where multiple decreasing threshold probability values are swept, as detailed in the pioneering papers.

The relevant part in the configuration file looks as follows:

================

"Hi-C": {

"input\_matrix": "path\_to\_input/hff\_200kb\_probability.hcs",

"intra\_sigma\_list": [

1.0, 1.0,1.0,1.0, 0.2,0.1,0.05, 0.02, 0.01, 0.008

], list of threshold values for intra-chromosome contacts (to be iteratively enforced)

"inter\_sigma\_list": [

1.0,1.0,1.0,1.0,0.2,0.1, 0.05,0.02,0.01,0.008,0.005,0.002,0.001

], list of threshold values for inter-chromosome contacts (to be iteratively enforced)

"contact\_range": 2.0,

"contact\_kspring": 1.0,

"actdist\_file": "actdist.h5", file generated from ActivationStep.py

"tmp\_dir": "actdist",

"keep\_temporary\_files": false,

"run\_evaluation\_step": false

},

================

The output file actdist\_file is then used in order to effectively assign the restraints to the structures. Such a file has a compressed format and essentially looks like:

I j d p

E.g.; for (diploid) contact between locii i and j, the activation distance d\_ij is given and the probability for that contact to be formed.

### Damid ---------------------------

The input (input\_profile) consists of a (haploid) lamina contact probability array, which is saved into a .txt file, and is read in by IGM in the DamidActivationStep.py part. The (i) entry gives the probability that locus i is close to the nuclear envelope.

A lamina contact is enforced in a structure by adding an energy term which is either spherical or ellipsoidal, depending on the envelope shape. Such an energy term equals one (in arbitrary units) when the bead is in contact with the envelope, and gets closer to zero, the innerer the position of the bead.

The relevant part in the configuration file looks as follows:

================

"DamID": {

"Input\_profile": "path\_to\_input/damid\_hff\_B.16.4pc.c1.00.txt",

"sigma\_list": [

0.7, 0.5, 0.4, 0.2 list of threshold values of lamina contacts (to be iteratively enforced)

],

"contact\_range": 0.05,

"contact\_kspring": 1.0,

"damid\_actdist\_file": "actdist.h5", file generated from DamidActivationStep.py

"tmp\_dir": "damid",

"keep\_temporary\_files": false

},

================

The output file damid\_actdist\_file is then used in order to effectively assign the restraints to the structures. Such a file has a compressed format and essentially looks like:

I d p

E.g.; for (diploid) nuclear envelope contact i, the activation distance d\_i is given and the probability for that contact to be formed.

### Sprite -------------------------

Sprite experimental data provides information about multi-body contacts between locii, which is expressed in terms of “clusters”. Clearly, more information is needed as to how clusters are defined and which distance thresholds are selected to define locus proximity.

The input data consists of a (haploid) cluster characterization, which is saved into a .h5 file (clusters), and is read in by IGM in the SpriteAssignment.py step.

* h5[‘data’] the indexes of all (haploid) locii making up the clusters are listed for each clusters, and all clusters are concatenated.
* h5[‘indptr’] contains the information about cleaving points identifying each cluster in ‘data’. An example may be beneficial to understanding this. Assume a chunk reads [0,3,13,21]: then, such a sequence of locus indexes indicating there are 3 clusters, [0,2], [3, 12] and [13,21][[3]](#footnote-2). Please note that the length of the list equals the number of clusters (plus one), and the sum of all the cluster sizes equals the length of the ‘data’ list. This is a good consistency check to make sure the input file is not broken.

A ‘cluster’ contact is enforced by introducing a dummy particle located at the geometric center of the cluster, and then by applying harmonic restraints between it and each locus assigned to that cluster (think about this as many springs originating from the cluster centroid and reaching out to each locus).

The relevant part in the configuration file looks as follows:

================

"sprite" : {

"clusters": "path\_to\_input/sprite\_clusters.h5",

"volume\_fraction": 0.05, volume fraction used to define the cluster

"radius\_kt": 50.0, Assignment is a stochastic process based on a Gibbs distribution, this is a parameter which helps define the Boltzmann factor.

"assignment\_file": "sprite\_assignment.h5",

"tmp\_dir": "/staging/fa/polles/tmpsprite",

"keep\_temporary\_files": false,

"batch\_size": 10,

"kspring": 1.0

}

================

The output file assignment\_file (produced by the assignment step) is then used to effectively assign the restraints to the structures in the population. Such a file is still h5 compressed, and the dictionary breaks down as follows:

* ‘Assignment’: array of length N (n\_cluster), entry i-th given the index of structure in which cluster i-th is to activated
* ‘Selected’: same as ‘data’ in input file, but locus indexing is now diploid
* ‘Indptr’: same as ‘indptr’ in input file

Sprite assignment is not iterative and does not require looping over different values of some threshold values. Even if assignment is performed multiple times, the result does not change.

***DETAILS OF THE DIFFERENT ALGORITHMIC STEPS***

A controller is a more general, higher hierarchical layer, which maps a serial function (task) to the workers: the actual way this is performed is conditional on the speicific choice of controllers which is made in the JSON config file. Syntax will be different (check IGM/parallel)

### Random Initialization Step

Configurations are randomly initialized within a sphere during the mapping step (which is managed by the controller onto the engines), each structure coordinates are stored in a .hms file in $TMP\_DIR. A master file .hss (n\_beads x n\_struct x 3) is created during the reduce step and is then flipped in the “repacking” step.

### Relaxation Step

Mapping step: each configuration is LAMMPS optimized. While progress status bar progresses, a bunch of different files are created.

In TMP\_DIR/OPT files ‘FILENAME\_%d’ + str(struct\_id) + **.log, .lam, .lammpstrj, .data.** When the calculation is over, the corresponding **relax\_#.hms** and **relax\_#.hms.ready[[4]](#footnote-3)** files are created one level ahead, in TMP\_DIR, and the temporary files in OPT are removed.

The .hms files contain the last frame from the relaxation step for a given configuration AND the set of violations that are later also merged in the reduce step and redirected to the log file to keep track. That information the software needs to figure out if an extra iteration is needed, for the same set of parameters.

Reduce step: the master file .hss is opened, and the “relaxed” coordinates from each .hms files are copied into the corresponding structure slice in the .hss file. The master file is closed and flipped (repacking).

Multiprocesses: parallel to igm\_run there is a “poller process” going on which constantly checks if the optimization is over and the ‘.ready’ is created, so that the .hss file can be updated on the fly, without having to wait for ALL the relaxation processes to be over.

Please note that this occurs at the controller level, the engines are sleeping as of now...each of them communicates back the information by creating the corresponding .hms file.

It also seems like the tqmd progress bar stalls if configuration A has not been optimized, even though A+1, A+2, …. have been. This may need to be adapted. As of now, the POLL\_INTERVAL parameter was increased to 10, this way the actual calculations are not constantly interrupted by the polling function continuously checking .ready files.

One thing to be particularly careful about is when the modeling step is interrupted, for whatever reason. If temporary files .log, .lam, etc are in the TMP\_DIR/OPT folder, but no calculation went to completion (such that the corresponding .hms files are not in TMP\_DIR), then restarting would automatically skip those configurations and move on to the next ones….please check this

### Activation Distance Step

In TMP\_DIR, a ACTDIST folder is created and inside it files of type **#.in.npy** are created. Each file contains a numpy array (N x 4), N = number of structures. Then, they are all merged together into a **actdist.hdf5** file. There is a mapping-reducing sequence here too, but it seems very fast (look into igm.log).

### Damid Activation Distance Step

In TMP\_DIR a DAMIDACTDIST folder is created and almost instantaneously a file **damid\_actdist.hdf5** is created. There is a mapping-reducing sequence here too, but it seems very fast (look into igm.log for details).

Now, the actual optimization step starts, and things are equivalent to the mapping step detailed in the Relaxation Step paragraph. The files created in TMP\_DIR are now called **mstep\_#.hms** and **mstep\_#.hms.ready**, and contain the optimized coordinates upon relaxing the hss configurations with restrained computed in the (Damid)Activationstep. What is interesting here is that the file **igm-model.hss.T** is being edited on the fly while the mapping step proceeds. That is probably due to mapping reading in initial configurations from the master hss file.

A temporary hss file summarizing the results for this setup (in terms of activation and damid activation thresholds) is then produced after reducing and repacking. Then the next iteration starts. Before repacking step, the code prints a summary of the thermodynamic properties and violations during the optimization.

The next iteration sees the actdist.hdf5 file touched and then the new #.in.npy(s) files are produced in the TMP\_DIR/ACT\_DIST directory. Apparentlh, the number of .npy files changes from iteration to iteration: try to figure out why is that.

The the files #.out.npy(s) are generated, everything is purged and a file ‘actdist.hdf5.sigma\_0.2000.iter\_0’ is created. Now, only this file and the original ‘actdist.hdf5’ are in the folder.

Then the file in DAMIDACTDIST is updated/touched. Then, the minimization of all the structures can start.

The logger info (‘Read 860 probabilities from last step‘) appears in the log file, right after DamidActivationDistanceStep (cut=50.00%, iter=0) - starting

And immediately before

DamidActivationDistanceStep (cut=50.00%, iter=0) - mapping

NB: the igm/igm/steps/ModelingStep.py poller was adapted in such a way that attribute hss.set\_coordinates() is only called once (the code works by replacing the i-th slice in the master matrix referring to the coordinates of the i-th configuration). Now the reducing step is lightning fast :) Look into ‘Relaxation’ for the details of the mapping step.

### Modeling Step

The chromatin model is defined by introducing degrees of freedom (particles) and interactions (mainly structural integrity restraints, such as connectivity and excluded volume, and IGM restraints from Hi-C or DAMID). The model is them optimized the same way as in RelaxInit step,

The main difference is that the output .hms file also contains the optimization statistics, especially the number of restraint violations. All the statistics are collectively summarized and saved into the hss population file, recapitulated in the logger file and are processed by the igm-report script to produce a preliminary analysis.

### **Codes and Wrappers**

**------------------------ igm/igm/tasks/modeling\_step.py**

Serial function to be mapped in parallel, the controller is general and can be specified in the configuration file. This file supersedes everything else, defines the most general structure for managing task distributions between controller and workers, which can be ipyparallel or dask or serial (see Controller\_class dictionary opening the file, each entry is defined, especially the map attribute, in the corresponding files...see below).

Main structure:

Controller\_class = {

u’serial’ : SerialController,

u’ipyparallel’ : AdvancedIppController,

u’ipyparallel\_basic’ : BasicIppController,}

cfg = Config(cfg\_file)

def modeling\_task(struct\_id, cfg\_file)

….

def modeling\_step(model, cfg):

serial\_function = partial(modeling\_task, cfg\_file)

pctype = cfg[‘parallel\_controller’]

popts = cfg[‘parallel\_controller\_options’]

controller = controller\_class[pctype](\*\*pcopts)

argument\_list = list(range(n\_struct))

controller.map(serial\_function, argument\_list)

**------------------------ igm/igm/parallel/ipyparallel\_controller.py**

This is the part of the code where the main code “connects” to the ipyparallel environment. The key is the map function (lines 66 and following), where the load\_balanced\_view() and map\_async(IppFunctionWrapper(parallel\_task, ...)) are called, where the parallel\_task (usually the lammps relaxation) is submitted to the different workers. If the ipyparallel environment is not run, then a “remote failure” message appears.

**------------------------ igm/igm/parallel/async\_file\_operations.py**

This is the file where the polling function is defined: when one calculation is completed in any engine (the polling checks that actively on the fly), then one operation is carried out, usually it is updating the coordinates of the corresponding structure index in the hss matrix. The key ingredient is the callback parameter, which indicates the operation to be performed within the polling function. The enumerate function is then used together with the progress bar module to estimate the remaining time to completion.

**------------------------ igm/igm/core/step.py**

This is the unified template for any step included in the overall IGM pipeline. Any computation is a IGM pipeline should be a subclass of this class. The .run() attribute executes, in order:

setup()

before\_map()

map()

before\_reduce()

reduce()

cleanup()

If the full step already went to completion, then the skip() member function is called and the code moves on to the next step.

The name() function returns the name of the class. It is mainly used for logging and visualization purposes. Overloading it to add information helps detail the output.

**---------- setup()**:

The role of setup is to prepare folders, files, data for the step. In particular, it needs to set the `self.argument\_list` special variable, which contains the argument to be mapped in parallel. It may also modify the self.tmp\_dir or modify the "runtime" section of the configuration object, stored in `self.cfg`. Also, automatic temporary files deletion can be specified, see the `Special member variables` section below. Note that this function will be called also on restart runs, if the whole step was not completed.

**---------- before\_map()**:

This function is executed just before mapping. It is intended o setup resources which are needed ONLY in case of mapping. in a restart run, it is skipped if mapping has already been completed.

**---------- task(arg, cfg, tmp\_dir)**:

The static `task` method is executed on parallel workers. Each run of the task method gets as first parameter one of the arguments in `argument\_list`. The second and third parameters are the same for each run and are the configuration object and the temporary directory. Note that any value returned by task is ignored because of a design choice. Any data processed and further needed should be stored on a accessible location, like a shared file system or server.

[ In general, large amounts of data or complex objects can be difficult to save properly. This means additional complexity in the restart design if something fails, which is better handled by delegating the data transport/storage to each specific case. ]

**---------- before\_reduce()**:

This function is executed just before reducing. It is intended to setup resources which are needed only for the reduce step, for example initializing in-memory resources if the mapping step is skipped. In a restart run, it is skipped if reduce() has already been completed.

**---------- reduce()**:

This is executed only on the master node after mapping, and it is intended for either serial steps which do not require a mapping, or to collect, reduce, and atomically write the results.

**---------- cleanup()**:

This is the last step, intended to cleanup temporary files and resources, and update the runtime environment. In general, overriding this method is unnecessary and not suggested.

**---------- skip()**:

When the step is skipped in a restart run, this function is called. It is intended for possibly setting runtime variables

Special Member Variables

------------------------

cfg : Config

configuration object

db : StepDB

database object

argument\_list : list

arguments to be mapped in parallel

tmp\_dir : str

path of the directory where temporary files will be saved

tmp\_extensions : list

list of extensions of temporary files. All files with the

listed extensions in tmp\_dir are removed at the cleanup()

call if `keep temporary files` is True

keep\_temporary\_files: bool

if True, it deletes temporary files during cleanup

uid : str

a unique string identifying the current step

The **run()** attribute actually runs the calculation and contains the part where the database file is checked (line 240) and the completed parts are skipped.

Also, this file contains the class StructGenStep(step) which is called in the Initialization Step.

**------------------------ igm/igm/model/kernel/lammps.py**

This is the reference script where data is collected, scripts are prepared and submitted, and the actual optimization of the structures is performed (Modeling Step).

def create\_lammps\_data(model, user\_args)

def create\_lammps\_script(model, user\_args)

def optimize(model, cfg):

m = LammpsModel(model)

# prepare data and scripts)

create\_lammps\_data(m, run\_opts)

create\_lammps\_script(m , run\_opts)

# run the lammps minimization

with open(script\_fname, ‘r’) as lamfile:

proc = Popen([lammps\_executable, ‘-log’, log\_fname],

stdin = lamfile,

stdout = PIPE, stderr = PIPE)

output, error = proc.communicate()

...

The key quantity is the igm.model object, which is the abstract model object for simulation optimization (define particels, define interactions, define restraints…)

### **Add new restraints in the pipeline**

Adding a new type of restraint from experimental data requires particular care, since many files making up the code require editing accordingly. In particular:

* Configuration file, since new parameters need to be added (this requires editing of the GUI to allow implementation)
* A new assignment (A) step is needed. A new class inherited from Step needs to be created, overloading setup, task, reduce, clean-up functions accordingly
* A new restraint class (inherited from Restraint) needs to be created, to add restraint to the structures. Please keep in mind that if novel types of interaction forces are needed, once has to create a new Force class and the code for the kernel to create it.
* The modeling step (M) needs to be updated to include the new restraint (if specified in the configuration JSON file)
* Finally, the igm-run script needs to include the assignment step, update checks on satisfactory completion, eventually update the runtime data.

To test an Assignment step, one can run the following:

from igm import Config

from igm.steps import NewAssignmentStep

cfg = Config(x)

t = NewAssignmentStep(jsonfile.json)

t.run()

And make sure that the controller is turned into ‘serial’ in the JSON file.

### **RUN IGM ON UCLA HOFFMAN2 HCP CLUSTER**

### 

### UCLA Hoffman2 cluster uses a Univa Sun Grid Engine job scheduler, whose syntax is very close in spirit to that of PBS schedulers, albeit with crucial differences. A good starting point to submit jobs onto the cluster would be the following .sh templates.

Running IGM is a multi-step submission procedure:

* (If applicable) Remove temporary or old files (from previous productions) from folder(s)
* Double check configuration .json file
* Create ipyparallel environment using **ipcluster**, controller + N engines: (it is recommended that the controller and the engines be started separately, this way the status of the connection/environment can be checked on-the-fly by looking into the output/error files)
* Submit serial **igm\_run.sh** job (actual calculation)

It is crucial that the jobs start serially:

Controller ---> Engines ---> Igm

In the following, template SGE scripts are provided to submit the different steps to the cluster.

**------------------ TEMPLATE: INITIALIZE ipyparallel CONTROLLER --------------------**

#!/bin/bash

#--- memory of the controller -----#

SMEM = 4G

#--- walltime (not to exceed 24hrs) ------- #

WTIME = 20:00:00

# --- email and flags for communication ----#

#$ -M XXXX@g.ucla.edu

#$ -m ea

#$ -N ipycontroller

#--- allocate resources for controller ---#

#$ -l h\_data=${SMEM}

#$ -l h\_rt=${WTIME}

#$ -cwd

# --- output files ----#

#$ -o out\_controller

#$ -e err\_controller

#--- copy full environment (modules) onto computing node --- #

#$ -V

export PATH="$PATH"

ulimit -s 8192

cd $SGE\_O\_WORKDIR

#--- extract ip address to start controller ---#

myip=\$(getent hosts \$(hostname) | awk '{print \$1}')

# --- launch ipcontroller which is being monitored ----#

MONITOR=$(command -v monitor\_process)

if [[ ! -z "$MONITOR" ]]; then

monitor\_process --wtype S ipcontroller --nodb --ip=\$myip

else

ipcontroller --nodb --ip=\$myip

fi

Echo “Controller submitted!”

**----------------------------------------------------------------------**

**----------------- TEMPLATE: ALLOCATE ipyparallel ENGINES ----------------**

#!/bin/bash

#$ -M XXXX@g.ucla.edu

#$ -m ea

#$ -N ipycluster

#$ -l h\_data=1G

#$ -l h\_rt=20:00:00

#$ -cwd

#$ -o out\_engines

#$ -e err\_engines

#$ -V

#--- create parallel environment (pe): allow for processors to be allocated on different physical nodes (dc\*)...this is mandatory, since no node physically contain more than ~30 cores

#$ -pe dc\* ${NTASKS}

export PATH="$PATH"

ulimit -s 8192

cd $SGE\_O\_WORKDIR

# --- start engines (aka SRUN --n=$TASKS, in SLURM syntax) simultaneously ---- #

MONITOR=$(command -v monitor\_process)

if [[ ! -z "$MONITOR" ]]; then

mpirun --n=${NTASKS} monitor\_process --wtype W ipengine

else

mpirun --n=${NTASKS} ipengine

fi

echo "should have submitted $NTASKS engines on the top of the controller!"

**----------------------------------------------------------------------**

(NB: It is straightforward to enforce the second script to start upon the first starting, by using the **hold\_jid** flag.

qsub -**hold\_jid** $SCHEDJOB $TMPFILE )

**------------------------ TEMPLATE: SUBMIT IGM COMPUTATION ----------------------**

#!/bin/bash

#$ -M XXX[@g.ucla.edu](mailto:bonimba@g.ucla.edu)

#$ -m ea

#$ -N run\_igm

#$ -o out\_igm

#$ -e err\_igm

#$ -V

#$ -cwd

# --- resources ---- #

#$ -l h\_data=20G

#$ -l h\_rt=23:59:00

#$ -pe shared 2

# --- define parallel processes ---- #

Export NUM\_OMP\_THREADS = 2

Echo “submit actual IGM calculation.”

# --- execute job and redirect output ---- #

igm-run igm-config.json > igm\_output.txt

**----------------------------------------------------------------------**

**NB**: igm\_run is strictly serial, it is controller responsibility to efficiently parallelize the calculation (the main idea is that each structure in the population can be optimized separately from all the others, which is a trivially parallel trask). However, there is multiprocess step (polling step), so using 2 processors which share memory (open\_omp style) may improve communication and not have competing processes on the same CPU.

Please note that file ‘**err\_igm**’ is updated on the fly and details the steps that are eventually summarized in the igm.log file. For example, the progress status bar for the mapping and reducing steps with the completion percentage, can be found there.

### Remark: the ipyparallel environment

1. The controller and engines can be started at once using the integrated command

**ipcluster**  start --n=${TASKS} --ip=$(hostname -i | awk '{print $0}’)

However, keeping track of the status of the controller/engines is more involved.

1. Make sure the different output/error files are named differently, in order to keep track of the different steps.
2. Upon creating the ipcluster environment, one can check the status from terminal by opening iPython and interactively typing:

**----------------------------------------------------------------------**

# ---- import ipyparallel

**from ipyparallel import Client**

# ----- call the Client

**r = Client()**

# ---- extract indexes for different engines (this is a good point to double check the number of engines activated)

**r.ids**

# ----- import socket on ALL engines

**with r[:].sync\_imports(): import socket**

# ----- print hostname of ALL engines (this is a good point to check the physical allocation of the cores on the supercomputer)

**%px print(socket.gethostname())**

**----------------------------------------------------------------------**

1. **Igm-run** uses a similar protocol to connect to the ipcluster environment and distribute the tasks. This happens in the task file in parallel folder.

**----------------------------------------------------------------------**

Import ipyparallel as ip

Rc = ip.Client( -- profile --) Use the local ip address

V = rc.load\_balanced\_view() Let the controller manage the distribution of tasks among the different engines

Res = v.map\_sync(f, \*args) CORE: a function is run on all engines

rcl.queue\_status() check status of the tasks on each core of the network

**----------------------------------------------------------------------**

### 

### 

### Freely experimenting with IGM

Assume that a user wants to experiment with the IGM code, without having to push/pull changes from the main repository. Please follow:

1. Fork the IGM repository onto one’s own GitHub account
2. Enter target folder from the terminal
3. Type **pip install -e Forked-address** ./
4. (Just for the sake of doing that, maybe reinstall the serial version of lammps)

The “editable” version will be compiled automatically upon each edit, locally. If the ALBERLAB version was previously installed, it will also be purged. This way, import igm will automatically call the editable version.

Also, if you want something to be printed into the log file, please follow the syntax

from ..utils.log import logger

logger.info(‘------stuff to be printed ----’)

Missing how to merge/pus/pull from/to forked repository

1. An additional entry in the .json dictionary is appended, “runtime”. This entry is required and is home to all generated parameters and those needed to run the different steps; it is updated for each list of threshold values (thetas and laminaDAMID thetas) in the preamble to “ActivationDistanceStep.py’ and ‘DamidActivationDistanceStep.py’ files. Once a value of sigma is run, it is systematically removed from all the lists in the “runtime” entry (.pop attribute), and once all those lists are empty, then the algorithm has gone to completion. [↑](#footnote-ref-0)
2. It is recommended that the script be submitted as a regular job to Hoffman2, since memory requirements can be quite intensive. [↑](#footnote-ref-1)
3. The locii assigned to the first cluster are then data[0], data[1], data[2]; locii assigned to second cluster are then [data[3], …, data[12]], and so on and so forth. [↑](#footnote-ref-2)
4. .ready files are empty and are used as signals: if those are there, then coordinates are available for updating the .hss master file. [↑](#footnote-ref-3)