GOLD Ultra Large Docking

Introduction

This document will explain how to use the new GOLD Docker images to run large scale virtual screens using GOLD on a Kubernetes (K8s) cluster.

GOLD jobs are generated locally and pushed to a message queue running on a RabbitMQ pod in the K8s cluster. GOLD worker pods will pick up jobs off the job queue and return results to the results queue. Results are retrieved from the results queue locally by running the provided script.

Download the scripts required here and extract locally to the machine that you will be using to manage the cluster. You will have a top level directory 'Gold-HPC' with two sub directories: K8s and Scripts.

The python scripts are examples of how to batch up the GOLD jobs and how to collect results from the queues and can be customised depending on your own workflow.

Docker and Kubernetes

We have created two GOLD Docker images that can be deployed to a standard Kubernetes cluster and easily scaled up to run more than a thousand jobs in parallel. Kubernetes can be run on many common Cloud platforms (e.g. Azure, AWS) or locally on a single machine for testing using Docker Desktop.

We will assume that the user already knows how to create a Kubernetes cluster and that they are able to run the kubectl command line tool to configure that cluster. For additional information, please refer to the Kubernetes Documentation.

Accessing the CCDC Harbor repository

There are two docker images required for deploying GOLD in the cloud; `goldbaseimage` and a `goldqueueimage`. These images are hosted at our container image registry harbor.ccdc.cam.ac.uk. Please sign up for an account and then contact CCDC support by email (support@ccdc.cam.ac.uk) so that you can be granted access to the images in the "gold-release" project. Your cluster will need to be able to access harbor.ccdc.cam.ac.uk to download the GOLD images.

Licensing

GOLD will need access to a floating licence server. The server must be a bare metal install (as opposed to a VM) and must be accessible by GOLD running on the cluster. For more information, please refer to the CCDC Licence Server Installation Notes.

> Please ensure you are requesting licences via your floating licence server. If you are not, it is likely the licence request will fail.

Editing the Kubernetes Cluster configuration scripts

We recommend editing the configuration scripts before deploying your K8s cluster.

secrets

To ensure security on the cluster and to restrict access to the message queue you need to generate several secrets. All secrets in Gold-HPC\K8s\secrets.yml must be base64 encoded. You can easily use python to base64 encode your secrets:

```
In [1]: import base64
In [2]: base64.b64encode(b'hello world')
Out[2]: b'aGVsbG8gd29ybGQ='
```

RabbitMQ Access

We are using the RabbitMQ message broker.

There are two users with access to RabbitMQ: the administrator (username: ccdc_gold_admin) used to access the web admin page and the standard user (username: ccdc_gold_worker) used to push/pull messages from queues. Generate passwords for these two accounts, base64 encode them and replace the text "<base64 encoded password>" in secrets.yml:

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-default-user
  namespace: gold-docking
type: Opaque
data:
  username: Y2NkY19nb2xkX2FkbWlu  # base64 for ccdc_gold_admin
  password: <base64 encoded password>
---
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-gold-user
  namespace: gold-docking
type: Opaque
data:
  username: Y2NkY19nb2xkX3dvcmtlcg==  # base64 for ccdc_gold_worker
  password: <base64 encoded password>
```

You will need the RabbitMQ ccdc_gold_worker credentials when running Gold-HPC\Scripts\submit_tasks.py and Gold-HPC\Scripts\collect_results.yml. Pass the ccdc_gold_worker password to the script on the command line:

```
python submit_tasks.py --password <RabbitMQ password> ...
python collect_results.py --password <RabbitMQ password> ...
```

RabbitMQ Erlang Cookie

RabbitMQ nodes/pods and command line tools use a shared secret for secure communication: the Erlang Cookie. Generate a value (e.g. a password greater than 20 characters), base64 encode it and edit secrets.yml:

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbitmq-erlang-cookie
  namespace: gold-docking
type: Opaque
data:
  erlang-cookie: <base64 encoded Erlang Cookie>
```

CCDC Licence String

The last secret that needs updating is the CCDC licence string. For a floating licence server this must take the form "-s <floating licence server URL>:5000":

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: gold-licence-string
  namespace: gold-docking
type: Opaque
data:
  ccdc-licence: <base64 encoded CCDC licence string>
```

RabbitMQ Considerations

Depending on the size of your virtual screen and the type of nodes (Virtual Machines) that you are using then you may need to increase the amount of disk space available for messages on the RabbitMQ pod. For instance if you push 75 GB of GOLD jobs to the queue we recommend that the RabbitMQ pod has access to at least 300GB of available disk space for persisted messages and results.

For our internal testing, we used the Microsoft Azure Cloud Service. Following the Microsoft Azure Documentation, we created a 512GB volume for our queue. To add the volume to the RabbitMQ pod edit the rabbitmq.yml:

rabbitmq.yml

```
spec:
  containers:
    ...
    volumeMounts:
      - name: <volume name>
        mountPath: /var/lib/rabbitmq
  volumes:
      - name: <give the volume a name>
        azureDisk:
          kind: Managed
          diskName: <insert disk name here>
          diskURI: <insert full disk ID here>
```

Configure the cluster and deploy the GOLD pods

The cluster configuration scripts can be found in the Gold-HPC\K8s directory. Once your cluster is deployed navigate to that directory and follow these steps:

To create the gold-docking namespace and update the context to use that namespace, run the following commands:

```
> kubectl apply -f namespace.yml
namespace/gold-docking created

> kubectl config set-context --current --namespace=gold-docking
Context "<my cluster>" modified.  # all subsequent kubectl commands will now apply to the gold-docking namespace
```

Then create the secret used to download the images from harbor using your harbor.ccdc.cam.ac.uk account credentials. Edit and run this command:

```
> kubectl create secret docker-registry ccdcharbor --docker-server=harbor.ccdc.cam.ac.uk --docker-
username=<username> --docker-password=<pswd> --docker-email=<email>
secret/ccdcharbor created
```

Apply the edited secrets.yml:

```
> kubectl apply -f secrets.yml
secret/rabbitmq-default-user created
secret/rabbitmq-gold-user created
secret/rabbitmq-erlang-cookie created
secret/gold-licence-string created
```

Deploy the RabbitMQ service.

We have chosen to run RabbitMQ on its own node and we have achieved this by setting a resource request and limit for the number of CPUs that the RabbitMQ service will use. It was observed that connection to the queue can fail when the host node is under computational stress. Adjust the CPU resource requests/limits in the yml file to be one less than the number of CPUs available on your chosen node type. e.g. for an 8 core node set the requests and limits to 7. This will stop the cluster horizontal scaler from adding any GOLD worker pods to the node running RabbitMQ.

rabbitmq.yml

```
    resources:
      requests:
        cpu: 7
      limits:
        cpu: 7
```

Now actually create the service:

```
> kubectl apply -f rabbitmq.yml
service/rabbitmq created
pod/rabbitmq created
```

And finally deploy the GOLD pods.

You need to consider what CPU load is acceptable - you can run smaller clusters with the CPU load close to 100% but we have found that with larger clusters it is better to restrict the number of GOLD pods to less than the number of available CPUs on the node. Again, this is achieved using CPU resource requests and limits e.g. we found that setting the limit to 1.07 will deploy 14 GOLD pods to a 16 core node and the average CPU usage will approach 90%. Decide how many GOLD pods you require per node and edit the limit in the yml file if required.

Next edit the number of 'replicas' (instances of the GOLD pod) to deploy. This is the number of GOLD pods per node times the number of nodes. e.g. 14 x 50 = 700

goldqueue.yml

```
spec:
  replicas: 700
```

And then actually deploy them:

```
> kubectl apply -f goldqueue.yml
deployment.apps/gold-docking-deployment created
```

Depending on the size of your cluster, you can expect this step to take some time. For example, for a cluster with 1000 GOLD worker pods, it could take up to an hour for all pods to reach a running and ready state. It is normal to see some pods fail and restart.

Viewing your Kubernetes Cluster

You can use the Kubernetes WebUI to view your K8s cluster or a GUI like Kontena Lens. Follow the links for information on setup and installation (where required).

If you use Lens you can easily add metrics to your cluster so that you can monitor resource usage.

RabbitMQ admin UI

You need the IP address of your RabbitMQ server. You can access this using kubectl:

```
kubectl get all

...
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                          AGE
service/rabbitmq   LoadBalancer   10.0.156.218   52.146.49.166   15672:31941/TCP,5672:31517/TCP   90m
...
```

Then using the IP address log in to the RabbitMQ admin page at http://<cluster external IP>:15672/#/

The Overview and Queues tabs allow easy access to the number of messages in the queues and the system resources in use, e.g. memory and disk space. These metrics are configurable. For more information, please visit the RabbitMQ Documentation.

Running a GOLD Virtual Screen

How to structure your input data

Your input ligand files should be in `.sdf` or `.mol2` format. The structure of input data depends on the kind of docking you wish to run. There are two options, which our reference scripts refer to as "gold_files" and "gold_path".

gold_files - files are stored in separate locations

The "gold_files" setup should be used when the input files necessary for a docking are stored in **separate locations** and you want to use the provided gold.conf template. For this you need:

- The absolute path to your protein file
- The absolute path to a file defining a reference ligand/cavity
- The absolute path to a collection of ligands to dock - this can be a `.tar.gz` archive of many structure files or a directory of ligand files or a collection of both.

When calling the `submit_task.py` script with the `gold_files` command, you will pass these all in individually (see below in "How to use the scripts to submit docking jobs").

gold_path - files are stored in a single location

The "gold_path" setup should be used when the input files necessary for a docking are stored in **the same location** and you are providing a gold.conf file. Note that the ligands to be docked must be in a different location than the gold configuration files because the submission script packages up everything in the 'gold_path' directory and adds it to each batch.

This option is more flexible than the 'gold_files' option and allows the user to use a gold configuration other then the one defined in the supplied template gold.conf e.g. define the binding site from a point rather then a reference ligand or include water(s) in the binding site. You can use the template gold.conf file as a starting point but the submission and worker scripts make certain assumptions about how the configuration is structured so you need to make sure those assumptions are adhered to:

- The gold configuration file must be named 'gold.conf'.
- Keep the directory structure flat (no sub directories) and delete any file paths. e.g. "protein_datafile = protein.mol2"
- Edit the ligand_data_file line so that the batched ligand filename can be inserted by the submission script. It should read "ligand_data_file {ligand_data_file} 10"
- Bear in mind how much information you want to get back from each job. We recommend using the MIN_OUT option so that you just get the bestranking.lst and gold.log files back - see the template gold.conf file.

When calling the `submit_task.py` script with the `gold_path` command, you simply pass in the absolute path to the `gold_path` directory and the path to the ligands to be docked (see below in "How to use the scripts to submit docking jobs").

Customising your GOLD settings

If you are using `submit_tasks.py` with the gold_files option you will use the template `gold.conf` file. To customise these settings simply edit this file.

> Do not edit the following template file names as these are used by `submit_tasks.py`: `{cavity_data_file}`, `{ligand_data_file}` and `{protein_data_file}`. These will be replaced with the correct file names before tasks are submitted to the queue, and editing them is likely to cause failures.

> For ultra large docking with GOLD, it is highly recommended you use the write_options = MIN_OUT configuration file option. This will reduce the output produced by GOLD to just the bestranking.lst, a gold.log and a target.txt. All errors/warning messages will be lost. If you wish to include the gold.err file, please include all possible write_options **except for** NO_BESTRANKING_LST_FILE, NO_GOLD_LOG_FILE and NO_GOLD_ERR_FILE. Please be aware that this may dramatically increase the overall size of the output produced.
>
> More information on output options can be found in the GOLD Configuration File User Guide.
>
> The target.txt file is generated by the GOLD worker python script running on the pod and contains the protein filename so that the results can be matched to the target if you have jobs against different targets in the queue. Remember to make the protein filename unique.

System Requirements for the template scripts

To run the scripts, you will need an installation of Python (we recommend Python 3.7) and the `pika` module installed. A stable internet connection is also a necessity.

> If connection is lost to the queue, submit_tasks.py will exit after five seconds. If batching exits you can restart after the last successful batch using the --ligand_file and --ligand_index command line options with the values that were last printed to the console. See 'python submit_tasks.py --help' for more information.

How to use the scripts to submit GOLD tasks

Once your data is structured correctly and your K8s cluster is running, submit_tasks.py will divide your data into batches and submit these to the queue as separate tasks for the GOLD workers. The script will output information about its progress in batching and submitting tasks, and exit once it completes the work.

GOLD Files

Run the script as follows:

```
python submit_tasks.py --password <RabbitMQ password>
                       -s [the external IP address of the queue server]
                       -b [batch size]
                       -l [A path to a file or directory containing ligands that are to be docked]
                       gold_files
                       -p [protein file]
                       -c [reference ligand file]
```

GOLD Path

Run the script as follows:

```
python submit_tasks.py --password <RabbitMQ password>
                       -s [the external IP address of the queue server.]
                       -b [batch size]
                       -l [A path to a file or directory containing ligands that are to be docked]
                       gold_path
                       -g [ A directory of required gold files to include with the batches (e.g. gold.conf,
protein, cavity, water(s) etc)]
```

How to use the scripts to get your results back

In order to obtain your results, navigate to the directory where you want your results to be saved and run the collect_results.py script as follows:

```
python <path to scripts>\Gold-HPC\Scripts\collect_results.py -s [the external IP address of the queue server.]
                         -t [the number of top ranking results to keep in results.lst. Default is 1000.]
```

The script will then watch the queue and wait for results to be returned. The collect_results.py script will merge and sort the bestranking.lst files into a single results.lst file which will keep only the top results, ordered by score. The number of results that are kept in results.lst is specified by -t/--top_number, as highlighted above. Note that sorting the list takes time and may cause results to be queued up if the number of results to keep is too large. In that case use the default setting and post-process the results to filter out the top N ligands that you require.

RabbitMQ

The state of the queue and use of system resources can be easily monitored in the RabbitMQ admin Web UI at http://<cluster external IP>:15672/#/ . Use the admin credentials that you added to secrets.yml.

Cluster and Node Health

Your chosen Cloud provider should have tools available for you to monitor the cluster and nodes. Make sure CPU, memory and disk use are within acceptable bounds.

Pod health

GOLD worker pods should be in the 'Running' state. You can get more information by looking at the pod log in the K8s Web UI or Lens. GOLD output is written to the log file and can be checked for errors. Check that a GOLD job matching the submitted tasks runs to completion locally without any errors. A failing GOLD job will result in the job being returned to the queue and the pod restarting and eventually entering the 'CrashLoopBackoff' state.

You can also log into the pod directly using the 'docker exec' command.

For large scale GOLD runs we recommend using the MIN_OUT option in the gold.conf file but for smaller scale jobs or for troubleshooting you can change the output options in the conf file so that more GOLD files are returned. You can play with the options in the Hermes GOLD GUI and create a gold.conf file that you can use as a template.

Known Limitations

Large Nodes

Internal testing suggested that the individual tasks are slower on larger nodes. For example, tasks run on a 32-CPU node would be approximately 1.2x slower than on a 16-CPU node.

Disk Space

The nodes used for internal testing had only 100GB of internal storage available. This was more than sufficient for the GOLD worker pods, but the queue required significantly more for some large jobs. As mentioned earlier, we added a 512GB volume for the RabbitMQ pod in order to remedy this issue.

Task submission speed

For large clusters with many GOLD worker pods, the task submission process can be slower than the processing of the jobs. This can result in starved pods. One workaround is to divide your data into separate sets and run a number of instances of submit_tasks.py in parallel.