KUBE Documentation

Introduction:

This program intends to be a bencharking and testing framework rather than a 'suite', as it is not our objective to gather and define a bunch of applications or software tests along with the proper datasets. This latter is the user's responsibility. As a benchmarking and testing platform (or framework), our goal is to create a mean to drive, centralize, control and organize the benchmarking and testing process in a consistent, flexible and easy to use way. In this manner any benchmark can be added and managed using this software.

The whole process is driven by a configuration file that contains all the rules to be applied in the different stages of the benchmarking/testing process. The configuration file to use can be specified as an argument, otherwise the default "etc/kube.yaml" file will be used. The user may also have different config files (for example if you have different machines and environments) and use one or another as needed.

Directory structure:

Let's start showing the directory structure of this framework since it is very related to the configuration file which is used to drive the whole process.

The directory structure is as follows:

```
|--- bench  ======> Contains all the sections in which all
|       |           the different kind of benchmarks have been categorized
|       |--- apps
|       |       |--- ....
|       |       |--- ....
|       |
|       |--- filesystems
|       |       |--- ....
|       |       |--- ....
|       |
|       |--- networks
|       |       |--- ....
|       |       |--- ....
|       |
|       |--- synthetics
|               |--- ....
|               |--- ....
|
|--- bin   ======> The main scripts lives here
|--- doc
|--- etc   ======> Contains the configuration file(s) and the templates used by the framework
|       |--- kube.yaml -> noor.yaml
|       |--- noor.yaml
|       |--- run.lsf.in
|
```

```
|--- lib
|       |--- PyYAML-3.10
|       |--- yaml -> PyYAML-3.10/lib/yaml/
|
|--- results  ======> Contains the analysis outputs for every run
       |             following the same structure as the 'runs' dir below
       |--- analysis
       |       |--- apps
       |       |        |--- ....
       |       |        |--- ....
       |       |
       |       |--- filesystems
       |       |        |--- ....
       |       |        |--- ....
       |       |
       |       |--- networks
       |       |        |--- ....
       |       |        |--- ....
       |       |
       |       |--- synthetics
       |                |--- ....
       |                |--- ....
       |
       |---  runs  ======> Contains the actual execution directories for every run
               |--- apps
               |        |--- ....
               |        |--- ....
               |
               |--- filesystems
               |        |--- ....
               |        |--- ....
               |
               |--- networks
               |        |--- ....
               |        |--- ....
               |
               |--- synthetics
                        |--- ....
                        |--- ....
```

Inside *bench/apps* there is the list of application names exactly as they appear in the configuration file. Each application should have at least one dataset, but may contain more than one. Inside an app dir, the system should be able to find the files **<dataset_name>.tgz** being **<dataset_name>** the name of the dataset as it appears in the configuration file.

The rest of items: filesystems, networks and synthetic are very similar but different to the apps because while each app expects to find one or several self contained datasets as a .tgz files; the rest of items expect to find one or several directory entries to each "dataset" . Since filesystems, networks and synthetic can be understood as "synthesized" benchmark (or "artificial", "ad-hoc", ...) they are not likely to contain a dataset in an application sense, but rather a full dir structure with binaries and dependencies that need to be compiled and called with certain arguments. For these reasons, these items have an additional tag in the configuration file: "dependencies". This can be a comma separated list of things that is needed to be copied to the run directory in order to be able to create a self contained dir to run it. Of course it is allowed to use wildcards here, so you can write something like:

```
dependencies: bin/* , lib/*
```

if you need to copy all the contents of the bin dir along with the libs

But you will find more details of the configuration file below.

Usage:

The main script is named 'kube.py'. It's been written in python and must be called with one argument (Option) which basically defines an action to be taken. 'Selector' is used then to select a specific item from any of the benchmark items: applications, networks, synthetic or filesystem. In this latter case the argument is needed to tell the system which specific item you are refering to. For example: if one selects -a (means application) it is necessary to tell exactly which app; so, the correct syntax would be '-a <appname>'. So, for example, the full line to run a specific application would be: 'kube.py -ra <appname>'

The different actions that can be performed are:

1.- Run (-r)

2.- Show configuration (-c)

3.- Postprocess (-p)

4.- Compare results using a kiviat diagram (-k)

Every acction is driven by a configuration file that constitutes the core component of this framework. Every action can be performed globaly (means that all the items marked as 'active' in the config file will be taken into account) or on a specific basis (just take into account specific items). To this end the 'Selectors' should be used. Every option can be combined with one selector at a time and then, a selector argument is needed.

Below is the syntax and options as show by the '-h' option:

```
Usage: kube.py <Option> [ <Selector> <arg> ]

Options:
 --version            show program's version number and exit
 -h, --help           show this help message and exit
 --clean              Remove all stored results from previous runs and
                      analysis
 -r, --run            Run the whole benchmark or a specific item according to
                      the 'Selectors' below
 -c, --configuration  Show the benchmark global configuration or a specific
                      item configuration according to the 'Selectors' below
 -p, --postprocess    Perform the Postprocess/Analysis stage for the whole
                      benchmark or for a specific item according to the
                      'Selectors' below .
 -y Y, --yaml=Y       Use the specified yaml configuration file rather than
                      the default one
 -k K                 Displays the kiviat diagram for the specified analysis
                      dir. The first argument is the path to the reference
                      directory and the second argument is a path to another
                      directory which might contain multiple dirs. If any of
                      these dirs contain a valid analysis data, they will be
                      used to compared against the reference dir specified in
                      the first argument

 Selectors:
   -a A               Select a specific application
   -n N               Select a specific network benchmark
   -s S               Select a specific synthetic benchmark.
   -f F               Select a specific filesystem benchmark
```

The workflow when using the benchmark framework would be something like:

1.- Prepare the configuration file:

Manually edit the file and decides which benchmark to run, with which dataset , number of cpus, batch system, etc, etc...

2.- check configuration (-c option) :

Just to see if everything is Ok, for example it could be use to see the submission scripts before actually submitting them to check everything is correct.

3.- run whole benchmark or specific items (-r option):

Here you can run the whole benchmark or just a specific item according to the selector used

4.- post-process (-p option):

Again here you can post-process the whole benchmark or a specific item depending on the selector used. The post-process copy the output files and take the actions specified in the config file to read and save the metrics which were also specified in the 'metrics' section within the config file. A '.csv ' is created with the metric name, value and units. There is another file created which is used to visualize the data with the '-k' option.

5.- visualize the data (-k option):

Optionally you can 'visualize' the metrics comparing two or more runs. With the '-k' options two arguments are needed: 1- the base run, all other runs will be compared to this one (the metric values will be normalized to this) and 2- the target runs. In fact this arguments are path to directories. The first is the path to the directory generated in the post-process stage for the 'template' run. And the second is a path to another directory. All the directories found will be processed, so in this way you can compare with multiple runs just including in one dir, all the output dirs from the post-process stage of all the runs you want to analise.

Configuration file:

The configuration file is the core component of the framework. The whole process is driven by this file. It is very flexible and there are just a few rules to follow. This flexibility is an advantage and a drawback at the same time since it brings you flexibility but the writer is responsible for the mistakes :) .

It has been written in YAML because of its simplicity, human readability and easy to parse. So anyone who open the config file could get a quick understanding of the process at a glance.

These are the sections in which the config file is divided:

Config file fields:

HOME: Full path to the place where KUBE is installed

OUTPUTS: Path to the directory where the results will be stored. If the leading "/" is provided then the absolute path is taken, otherwise the relative path to HOME will be used.

BATCH: List of the available filesystems.

The fields are:

'name': is mandatory and is needed to identify the batch system to use within a benchmark.

'script': the name of the template used to create the submission script. if the leading "/" is provided then the absolute path is taken, otherwise the relative path to [KUBE][HOME]/etc will be used. Example of the template script used for Noor. Notice that all *$<VAR_NAME>$* will be replaced by the specific dataset values at the moment of its creation:

```
#!/bin/bash -l

#BSUB -J %NAME%
#BSUB -o KUBE%NAME%_%NUMPROCS%.out
#BSUB -e KUBE_%NAME%_%NUMPROCS%.err
#BSUB -q %QUEUE%
#BSUB -W %WALLCLOCK%
#BSUB -n %NUMPROCS%
#BSUB %MPIFLAGS%
#BSUB -R span[ptile=%TASKS_PER_NODE%]

module purge
%MODULES%

%LAUNCHER% %LAUNCHER_FLAGS% %EXE% %ARGS%
```

'submittedmsg': is the output message (or just part of it) of the submission command containing the JOBID. The output of the submission command will be searched looking for this pattern to get the JOBID and also to determine if the job was submitted correctly to the batch system.

'submit': Is an empty field that contain two other sub-fields:

'command' : submission command

'parameters'. params to the submission commands if any

If the field 'script' is defined, then the submission command will be:

```
<command> <parameters> <script>
```

A special batch system named **MANUAL** is used in case there isn't any batch system available. This is a mandatory entry that will allow to execute jobs without any batch system. In this case the mandatory tags are **'name**' and **'submit**'. And inside submit, **'command'** and **'parameters**'. For a benchmark using manual launcher (**MANUAL** batch system) the submission command to be used would be:

```
<command> <parameters> <exe> <args>
```

Variables *%<VAR_NAME>%* specific to the dataset will be replaced at the moment the script is generated. Other *%<VAR_NAME>%* variables can be used in the **'submit**' section if they are defined... for example:

```
        - name: MANUAL
         hostslistfile: hostnames
         submit:
               command: mpirun
               parameters: -np %NUMPROCS% -machinefile %HOSTSLISTFILE%
```

In this example, **'hostslistfile**' has been added to be replaced in the **'parameters**' section. Notice that *%NUMPROCS%* will be replaced by the value of the field **'numproc**' of the current run.

The 'batch' field is mandatory for any dataset, but if the final user doesn't want to use any batch system or manual launcher, for example if you want to run just a sequential test like if you run it from the command line; then he just have to set **'batch**' to **'None**' in the desired dataset:

```
         - name: gpfs_iozone3
          active: true
          batch: None
          numprocs: 1
          modules:  module load gcc mpi-openmpi
          datasets:
               - name: iozone3_408
                 args: -Rab output.wks -g 2G  -I
                 exe: bin/iozone
                 dependencies: bin/*
                 active: true
                 outputs:
                   output: KUBE_%NAME%_%NUMPROCS%.out
                   iozone: iozone.tmp
```

BENCH: Contains the following fields: *APPS, FILESYSTEM, SYNTHETIC, NETWORKS*. Each of which contains a list of items to be benchmarked.

APPS: Contains a list with the applications to be used and its configuration. Mandatory fields for each apps are: **'name','batch','active','dataset**'

'name': Identifies the app

'batch': Selects the batch system to be used. Must be a valid name in BATCH section.

'active': Indicates whether the app will be run or not

'dataset': Any active app must have a dataset defined. It is actually a list since one application could run several datasets. Outside the 'dataset' field, user can define whatever tag he wants and it will be used inside each dataset unless the same tag is defined inside any dataset. In this case the tag defined inside the dataset has preference. Note that this tags will be used to match the variables *%<VAR_NAME>%* (in capitals) in the submission script (or the submission command) and build up a usable script (or command). So there is no limitation on the tags, except the mandatory ones. The **'numprocs**' tag may contain a comma separated list of values and this tag represents the number of cores to be used

Here is an example of an application section:

```
        - name: namd
          active: true
          batch: LSF
          # following parameters are dataset dependent...
          # This values will be used unless they are redefined in the "dataset" section
          exe: namd2_mpi
          modules:  module load gcc mpi-openmpi  libs-extra fftw2/2.1.5-openmpi-1.4.3-sse4.2-sp namd
          metrics:
             - name: wallclock
               units: secs
               command: cat %OUTPUT% | grep WallClock | gawk '{print $2}'
             - name: days/ns
               units: days/ns
               command: cat %OUTPUT% | grep  Benchmark | gawk 'END{print $8}'
             - name: cputime
               units: secs
               command: cat %OUTPUT% | grep CPUTime | gawk '{print $4}'
             - name: memory
               units: MB
               command: cat %OUTPUT%  | grep Memory | gawk '{print $6}'

          # here goes the specific dataset configuration
          datasets:
             - name: namd_nucleosome146Katoms
               active: false
               args: nucleosome146Katoms.namd  > nucleosome.out  2>nucleosome.err
               outputs:
                  output: nucleosome.out
                  another_output: KUBE_%NAME%_%NUMPROCS%.out

             - name: namd_apoa1
               active: true
               numprocs: 2,4
               args:  apoa1.namd   > apoa1.out 2>apoa1.err
               outputs:
                  output: apoa1.out
                  another_output: KUBE_%NAME%_%NUMPROCS%.out
```

The rest of tags inside **'BENCH**' are :

FILESYSTEMS:

NETWORKS:

SYNTHETICS:

They will be explained together since their behavior is similar.

The only differences of these fields compared with the **'APPS**' is that they may contain an additional tag: **'dependencies**' used to copy all the dependencies needed to create a self contained run directory. It allows wildcards and can be a comma separated list of items. In addition to this, the element contained in **'exe**' will be copied to the execution site if not full path is specified. For **'APPS**' it is understood that the dataset is self-contained, so no dependencies are allowed.

Some additional 'rules' to follow in the config file:

1.- Only **'outputs**' , **'args**', **'dependencies**' and **'exe**' fields may contain references to fields as *%<VAR_NAME>%*

2.- The basic and final element is a *dataset*. Any tag not defined at this level will be searched for in the immediate upper level: **'APPS**', **'FILESYSTEM**', ... and if not found there, then the corresponding **'BATCH**' entry is searched.

3.- if %NUMPROCS% is used, one entry per proc will be created.

4.- The **'metrics'** elements may contain references to the outputs names defined above as well as references to other tags in the config file.

5.- **'HOME**' tag must be in full path pointing to the place where KUBE is installed

6.- **'BATCH**' defines a list of available batch system configuration

7.- As a rule of thumb, all paths are relative unless the leading "/" is provided; then the absolute path is taken, otherwise the relative path is used.

8.- Any arbitrary field can be used

9.- the mandatory fields that must be defined when the dataset level is reached are: **name, active,args, exe, output** and **numprocs**

Post processing:

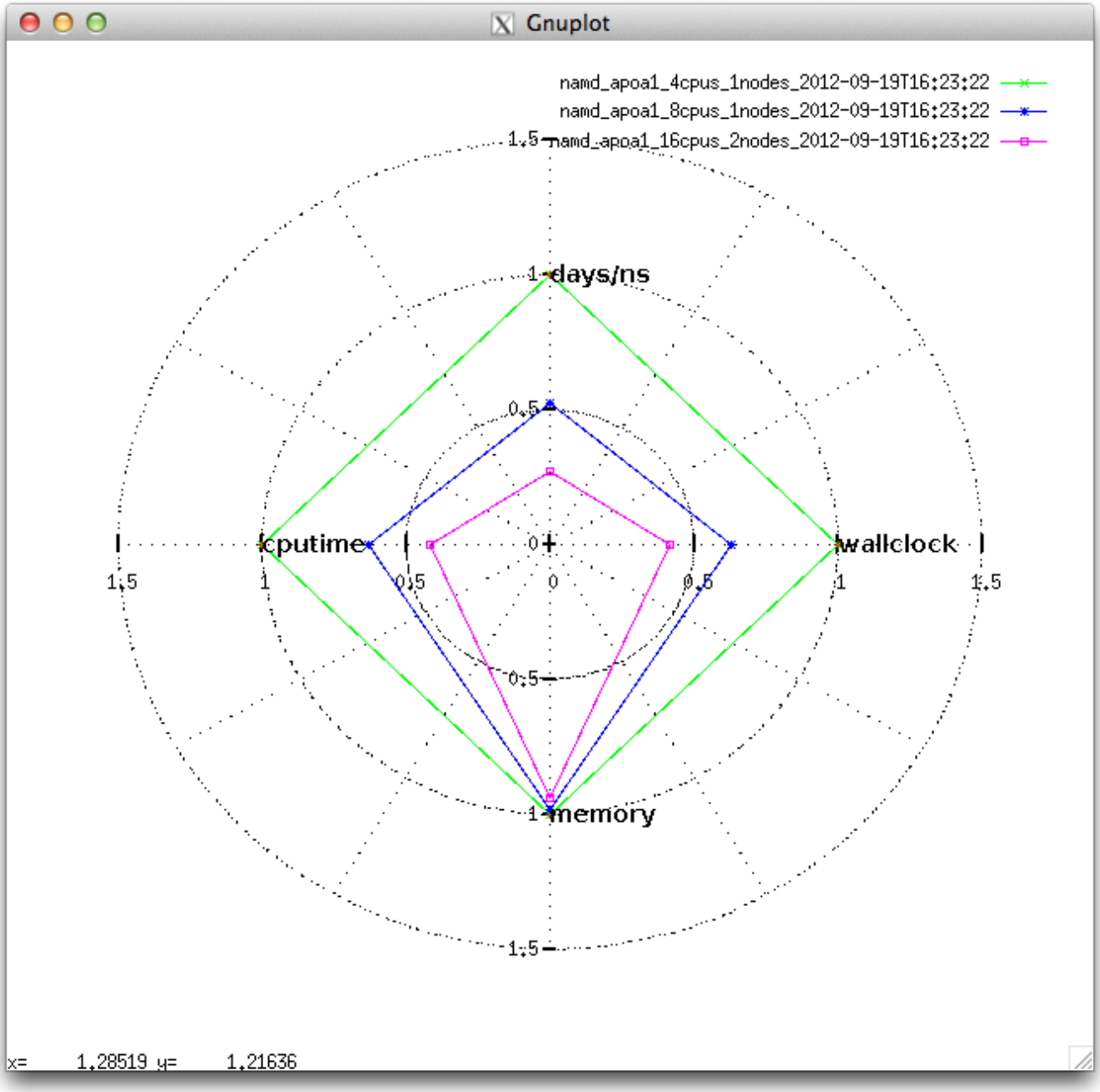
Some fields are included to make easier the analysis and comparison of the final results. The key fields here are: **'outputs**' and **'metrics**'.

When the post-process mode is run (-r option), basically the **'outputs**' are copied away from the run dir and into the analysis dir. So it is clear that **'output**' field is mandatory. Optionally you can declare some **'metrics**'. Each metric declaration contains its name, the units and the command needed to obtain the desired value. This command often implies parsing one or more output files. If more complicated actions needs to be performed in order to obtain a metric, we recommend to create a script, include it as a 'dependencies' and put inside all the logic to obtain the metric.

When metrics are defined, two additional files are created in the post-processing stage: one .csv with the metrics and a .raw file that will be used to visualize the data in the kiviat diagram

When using the kiviat diagram, *gnpulot* is needed. The -k option Displays the kiviat diagram for the specified analysis dir. The first argument is the path to the reference directory and the second argument is a path to another directory which might contain multiple dirs. If any of these dirs contain a valid analysis data, they will be used to compared against the reference dir specified in the first argument.

The following image is the Kiviat diagram for one namd run using the apoa1 dataset with 4, 8 and 16 cores. The metrics used for comparison are: Wallclock in secs, days/ns, cputime in secs and Memory in MB:


Gnuplot
namd_apoa1_4cpus_1nodes_2012-09-19T16:23:22
namd_apoa1_8cpus_1nodes_2012-09-19T16:23:22
namd_apoa1_16cpus_2nodes_2012-09-19T16:23:22
days/ns
cputime
wallclock
memory
1.5
1
0.5
0
x= 1.28519 y= 1.21636