1. Application Kernels Remote Runner (AKRR) 2
1.1 AKRR: Adding a New HPC Resource 10
1.2 AKRR: Deployment of Applications Kernels on a Resource 22
1.2.1 AKRR: Creating New Application Kernel 23
1.2.2 AKRR: Deployment of HPCC Applications Kernels on a Resource 35
1.2.3 AKRR: Deployment of IMB Applications Kernels on a Resource 40
1.2.4 AKRR: Deployment of IOR Applications Kernels on a Resource 48
1.2.5 AKRR: Deployment of NAMD Applications Kernels on a Resource 64
1.2.6 AKRR: Deployment of NWChem Applications Kernels on a Resource 76
1.3 AKRR: e-Mail Report Generation 83
1.4 AKRR: Scheduling and Rescheduling Application Kernels 84
1.5 AKRR Server Module Installation Guide 86
1.6 Setup Walltime Limit 95

Application Kernels Remote Runner (AKRR)

- Overview
  - Application Kernel Remote Runner (AKRR)
  - Application Kernel Process Control
  - Application Kernel Automatic Anomaly Detector
  - Application Kernels
- Overview of Parameters
  - AKRR Configuration Files
  - Batch Job Script Template
- Installation Instructions
  - AKRR Server Deployment
  - Adding a New HPC Resource
  - Deployment of Application Kernels on a Resource
  - Enable Application Kernel Support in OpenXDMoD (see OpenXDMoD documentations)
- Usage
  - e-Mail Report Generation
  - Scheduling and Rescheduling Application Kernels
  - Setup Walltime Limit

Overview

Application Kernel Performance Monitoring Module of XDMoD tool is designed to measure quality of service as well as preemptively identify underperforming hardware and software by deploying customized, computationally lightweight “application kernels” that are run frequently (daily to several times per week) to continuously monitor HPC system performance and reliability from the application users’ point of view. The term “computationally-lightweight” is used to indicate that the application kernel requires relatively modest resources for a given run frequency. Accordingly, through XDMoD, system managers have the ability to proactively monitor system performance as opposed to having to rely on users to report failures or underperforming hardware and software.

The application kernel module of XDMoD consists of three parts. 1) the application kernel remote runner (AKRR) executes the scheduled jobs, monitors their execution, processes the output, extracts performance metrics and exports the results to the database, 2) the application kernel process control identifies poorly performing individual jobs, 3) the application kernel automatic anomaly detector analyzes the performance of all application kernels executed on a particular resource and automatically recognizes poorly performing systems. Packaging-wise first one comes as a separate installation and the last two installed within XDMoD framework.

Application Kernel Remote Runner (AKRR)

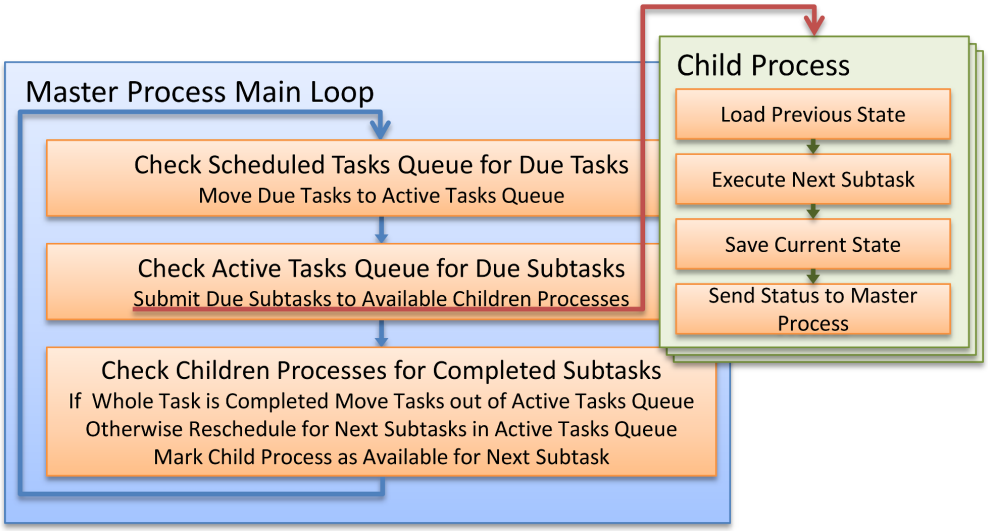
AKRR executes application kernels on HPC resources using the same mechanism as a regular user, for example it uses ssh to access the system and submits job scripts through the system scheduler. This allows for not only monitoring the performance of the application kernels themselves but also testing the whole workflow that regular users employ in order to carry out their work.

AKRR was designed to execute a large number of jobs on a number of HPC resources in 24/7 mode. To achieve high reliability, a multi-process design was chosen where the master process dispatches a small self-contained subtask to the children processes. This allows the master process code to be relatively simple and moves the more complicated code to the children processes. This way a severe error on one of the child processes does not cause the whole system to collapse.

Table 1. Subtasks of the Application Kernel Jobs .

#	Subtask/step Description
1	Create a job script
2	Copy it to HPC resource and submit to queue. Reschedule to repeat until success or allowed number of attempts is exceeded
3	Check job status on HPC resource. Reschedule to repeat until job complete or allowed in-queue time is exceeded
4	Collect the job output if it is present
5	Process the output if it was collected
6	Load results to database. Reschedule to repeat until successs or allowed number of attempts is exceeded

Although application kernels are computationally lightweight and their pure execution time lies between minutes to half an hour, the total time from job script creation to loading the results to the database can easily takes several days, due to potentially long queue wait times. In order to manage a large number of jobs and to be able to recover from critical failures, the entire application kernel execution task is split into small self-contained subtasks (see Table 1 for the subtasks). Each subtask is executed by a child process and should take only a few seconds. At the end of each subtask, the current job state is dumped to the file system. This allows us to recover to the last known state of AKRR in the case of a critical failure.



Scheduled Tasks Queue

Due Time	Resource	App.Kernel	N_{nodes}	Repeat in
2014-08-10 01:15	Stampede	NAMD	8	1 day
2014-08-10 01:15	Trestles	NWCHEM	4	1 day
2014-08-11 05:45	Gordon	GAMESS	4	3 day
2014-08-11 01:15	Lonestar	HPCC	16	1 day

Active Tasks Queue

Due Time	Resource	App.Kernel	N_{nodes}	Status	Next Step
2014-08-10 01:10	Stampede	NAMD	4	New Task	Create and Submit to Queue
2014-08-10 01:12	Gordon	NWCHEM	8	Still in Queue	Check the Queue

Figure 1. Illustration of the AKRR (Application Kernel Remote Runner) master process main loop with example scheduled and active task queue content.

The AKRR master process utilizes two queues for job tracking: the first one is named “scheduled tasks” and contains the jobs scheduled for execution in the future; the second one is named “active tasks” and contains jobs which are currently executed (Figure 1). When the scheduled time occurs, the job is moved from the scheduled tasks queue to the active tasks queue with the current due time. When a job in the active tasks queue is due, the master process dispatches a subtask of this job to a child process. The child process executes the subtask; and in the case of a successful execution, it requests the master process to schedule the next subtask for execution, otherwise the same subtask is rescheduled for future execution. When all subtasks are completed or the allowed number of rescheduling attempts is exceeded the job is moved out of the active tasks queue.

Application Kernel Process Control

Since the same input file is used each time for a given application kernel, the execution time or other metric that the application kernel is designed to measure (for example, i/o rate) should be relatively constant from one run to the next. Indeed, this is the very basis for the use of the application kernels to provide quality of service metrics. Since there are a substantial number of application kernels and there can be several metrics associated with each application kernel, it is desirable to have an automated process for determining if each application kernel is performing within its normal bounds. We refer to this scheme as process control.

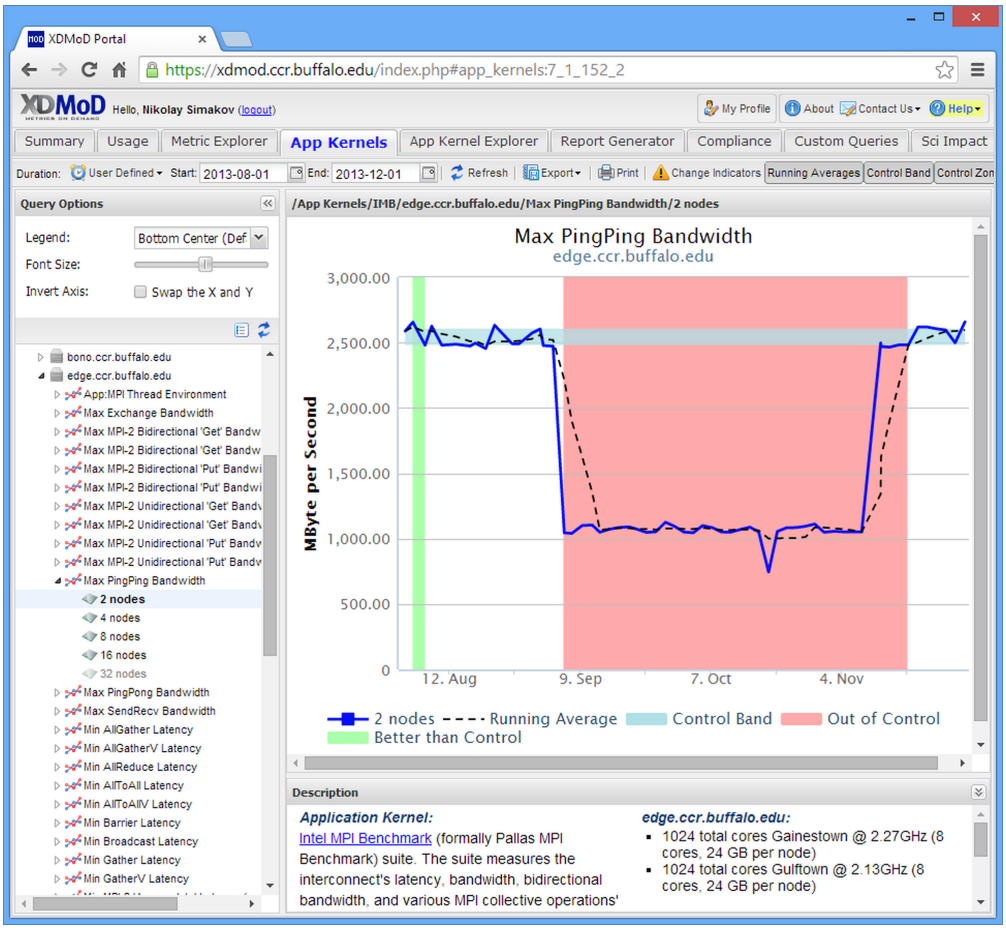


Figure 2. Application Kernel process control. The time history for the network benchmarking application kernel, IMB, demonstrates an underperforming region. The solid blue line is the data, the dashed black line is a 5-point average and the blue shading indicates the control zone range. The red zones indicate that the process is out of control in an unfavorable sense while the green zones indicate superior performance compared to the in control performance.

Process control monitors the performance of each individual application kernel metric and automatically identifies under-performing regions. Figure 2 illustrates the identified performance regions for the ping bandwidth metric of the IMB kernel. Each data point is categorized as either in control, out of control or better than control. The control region is defined to be the normal operating envelope of the application kernel. In the beginning, the control region is automatically selected and is often associated with the installation of a new application kernel. The process, in this case the performance of a given application kernel, is assumed to be nominally in control in this region. If the 5-point running average at a given point beyond the control region exceeds a specified tolerance (based upon the data range in the control region) the process is flagged as out of control. In Figure 3 the region where the network ping-ping bandwidth metric of the IOR application kernel drops substantially is automatically evaluated to be out of control. The control region is automatically readjusted to account for software environment updates. The updates are determined by the change in application signature [20]. The 5-point running average used to determine the baseline for a given metric measured by a specific application kernel argues in favor of running the application kernels fairly frequently (daily or several times a week at least). For example, if they are only run once per week, then it may be several weeks before enough runs have occurred for the process control algorithm to automatically identify a poorly performing application kernel. The end result being that the HPC resource may have been running in a degraded mode for that entire time.

Application Kernel Automatic Anomaly Detector

The automatic anomaly detector creates a performance map for all application kernels executed on a particular HPC resource (Figure 3). The performance map is a discrete heat map depicting the dependency of the application kernel performance on its execution on a given day. Three outcomes are associated with each application kernel on a given day, namel was a run in control, was it out of control, or did it fail to run. We choose to represent this by a triplet of integers. Thus 1/0/1 would refer to an application kernel that ran 2 times and one of the times was in-control and the other time failed to run. Similarly, 2/1/0 would indicate that the application kernel ran 3 times and 2 of those times was in-control and 1 time was out of control. In order to more readily identify anomalous behavior in the performance map we have elected to use a color scheme for the triplets. A triplet which is green means the application kernel ran successfully (was in-control) at least once on that day. A triplet highlighted with yellow indicates that the application kernel ran but was out of control. Finally, a triplet highlighted in red indicates the failure of the application kernel to run for all runs that day.

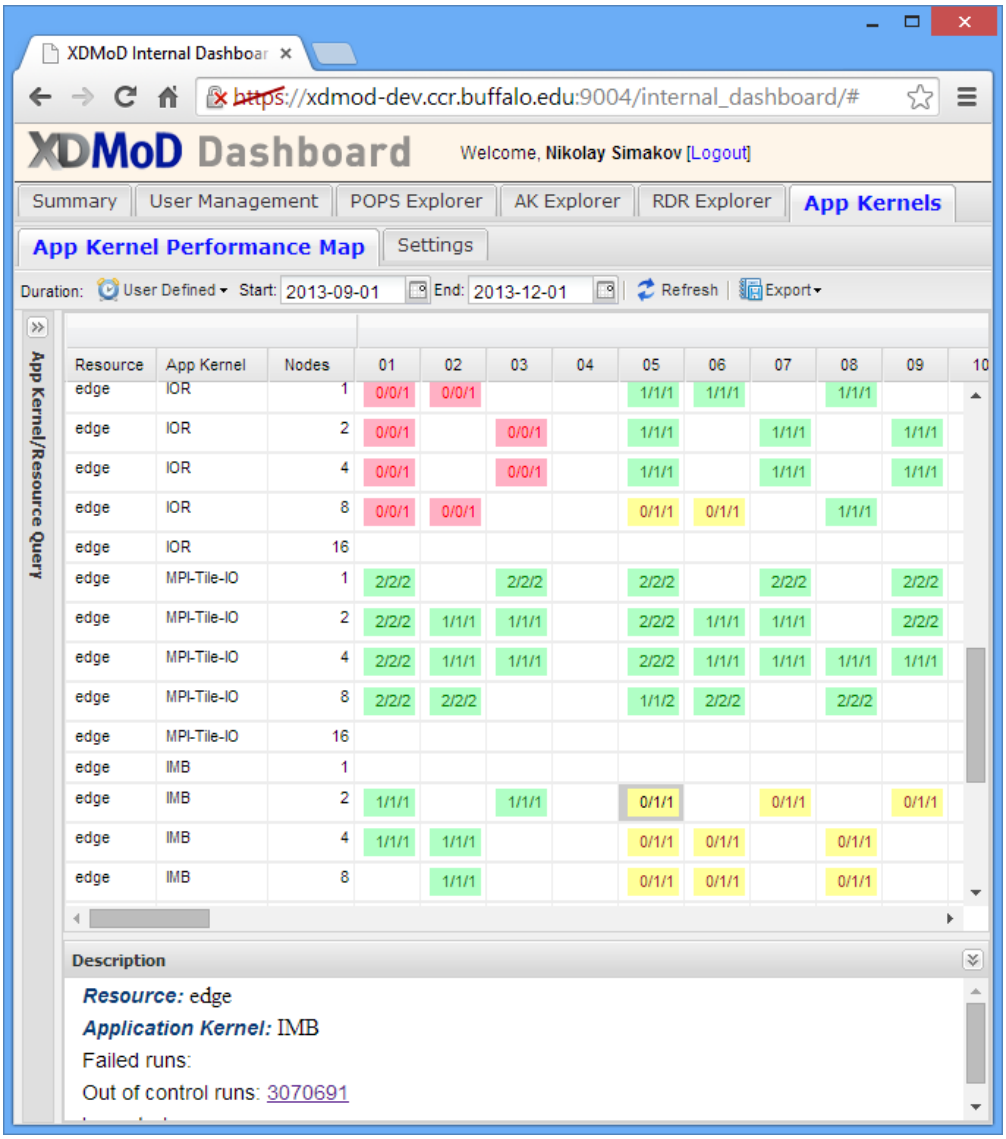


Figure 3. Portion of the application kernel performance map for the Edge cluster at CCR-UB showing which kernels ran normally (green), which ones ran sub-optimally (yellow) and which ones failed to run (red).

[XDMoD] Daily App Kernel Execution Report for 201... Message 3 of 5629

Report Period: 2014-08-06

Summary for app kerells executed on 2014/08/06Total number of runs: **180**
Number of failed runs: **94**
Number of runs without control information: **0**
Number of out of control runs: **15**
Number of runs within threshold: **71**

Number of repeatedly failed runs : **3**
Number of repeatedly underperforming runs : **8**

Resource	In Control Runs	Out Of Control Runs	No Control Information Runs	Failed Runs	Total Runs
edge	22 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	22
gordon	0 (0.0%)	0 (0.0%)	0 (0.0%)	61 (100.0%)	61
lonestar4	24 (72.7%)	7 (21.2%)	0 (0.0%)	2 (6.1%)	33
stampede	25 (75.8%)	8 (24.2%)	0 (0.0%)	0 (0.0%)	33
trestles	0 (0.0%)	0 (0.0%)	0 (0.0%)	31 (100.0%)	31
Total	71 (39.4%)	15 (8.3%)	0 (0.0%)	94 (52.2%)	180

Problem Detection through Performance Patterns

#	resource	app kernel	nodes	message
1	gordon	*	*	all app kernels on resource failed to run
2	trestles	*	*	all app kernels on resource failed to run

...

Active Tasks as of 2014-08-07 2:29am

task id	resource	app kernel	status
3138152	edge	xdmod.app.astro.enzo.node	Still in queue. Either waiting or running

...

Performance Map

			July, 2014		August, 2014					
Resource	App Kernel	Nodes	30	31	01	02	03	04	05	06
gordon	xdmod.app.astro.enzo.node	1	1/0/0	1/0/0	1/0/0			0/0/1/	0/0/1/	0/0/1/
gordon	xdmod.app.astro.enzo.node	2	1/0/0	1/0/0	1/0/0			0/0/1/	0/0/1/	0/0/1/
gordon	xdmod.app.astro.enzo.node	4	1/0/0	1/0/0	1/0/0			0/0/1/	0/0/1/	0/0/1/

App Kernel Instance #3137526

Instance Data | Output Data

AK Name: Enzo
Instance Name: xdmod.app.astro.enzo.node.2
Processing Units: 2 node
Date Collected: 2014-08-05 02:13:57
Resource: gordon
Status: failure
Deployment Instance Id: 3137526

Message

```
ERROR Can not created batch job script and submit it to remote queue

Traceback (most recent call last):
  File "/data/arrpack/arr/arrtaskappker.py", line 32, in CreateBatchJobScriptAndSubmitIt
    try:
  File "/data/arrpack/arr/arr.py", line 257, in sshResource
    rsh=sshAccess(headnode,ssh=remoteAccessMethod,username=username,password=sshPassword
  File "/data/arrpack/arr/arr.py", line 235, in sshAccess
    raise arrError(ERROR_CANT_CONNECT,"Probably %s refused the connection. "%(remotemach
arrError: Can't establish a connetion. Probably gordon.sdsc.edu refused the connection.

 End Of File (EOF) in read_nonblocking(). Exception style platform.
```

Figure 4. Fragment of an e-mail report generated by the automatic anomaly detector. The e-mail contains html links which will redirect to areas of interest within e-mail or to the XDMoD website to view a detailed report on the individual application kernel run. Throughout the report color coding is used: green for normally performing jobs, yellow for underperforming and red for completely failed runs. In this report it was determined that all jobs submitted to the Gordon HPC resource failed to execute. The performance map shows that given application kernels were performing properly at the end of July and suddenly failed to execute. Clicking on particular jobs will bring a detailed job execution report, which indicates an inability to access the resource. Further investigation revealed the change in resource access method from GlobusSSH to OpenSSH.

The automatic anomaly detector also generates a summary report (Figure 4) that combines statistics on the application kernel execution, detected problems, and the performance map. The reports can be delivered by email to subscribed users on a regular basis (daily, weekly or monthly) and immediately in case of degraded performance.

Using application kernel process control and the application kernel automatic anomaly detector, site administrators can monitor application kernel run failures for troubleshooting performance issues at their site.

Application Kernels

Currently eight application kernels are used for performance monitoring: NWChem, GAMESS, NAMD, Enzo, GRAPH500, HPCC, IMB and IOR. We briefly describe each application kernel to characterize the type of information that each provides on HPC operations.

NWChem is a heavily-used computational chemistry code that spans the gamut from molecular mechanics and molecular dynamics to full ab initio calculations. Its design goal was to handle a wide range of problems in quantum chemistry and dynamics. It was designed to be scalable to take advantage of the computational capability of large HPC clusters to address large problem-sizes.

GAMESS is an ab initio computational chemistry code.

NAMD is a parallel molecular dynamics code that was specifically designed for high performance simulations on large HPC clusters. It is routinely one of the most heavily run applications on the large XSEDE compute resources. It is primarily used for large biochemical problems. It has been scaled to thousands of cores.

GRAPH500 is a benchmark used to quantify communications rather than flops. There are two kernels, the first generates a graph and compresses it and the second does a parallel search over random vertices.

HPCC is a benchmark that consists of 7 tests: 1) Linpack, 2) matrix multiplication, 3) memory bandwidth, 4) parallel matrix transpose, 5) random memory access, 6) fast fourier transform and 7) bandwidth and latency. The goal is to measure a wide range of HPC operations.

IMB is an Intel benchmark that measures MPI node-to-node communication.

IOR is a benchmark used to test parallel file systems using POSIX, MPIIO and HDF5 interfaces.

Enzo is an adaptive mesh refinement code for astrophysical simulation of cosmological structure formation. The purpose of this code in our application kernel collection is to address the performance of scientific applications that utilize adaptive mesh refinement in solving PDEs.

The proper choice of application kernels input parameters is critical for keeping total execution time small but still being large enough to be representative of a real-life workload. Further complicating these requirements, it is also desirable to use the same problem for all node counts in order to monitor the parallel scaling. Furthermore these input parameters eventually need to be readjusted due to growth of computational power and advances in the user applications. For NWChem, GAMESS, NAMD, GRAPH500 and ENZO we choose to use the same problem size and for HPCC, IMB and IOR the problem size is proportional to the node count. The total run-time for the majority of application kernels lies in the 1-30 minutes range.

Overview of Parameters

AKRR Configuration Files

AKRR parameters files are located at $AKRR_HOME/cfg directory. These files are essentially python scripts and so python syntax is used. To refresh on the syntax, language construction most often used for configuration is listed bellow:

```
#pound sing for comments

#value assignment to variable
DB_host="127.0.0.1"
exportDB_host=DB_host

#triple quotes for long multi-line strings
batchJobHeader="""#!/bin/sh
#SBATCH --partition=general-compute
#SBATCH --nodes={akrrNNodes}
#SBATCH --ntasks-per-node={akrrPPN}
#SBATCH --time={akrrWallTimeLimit}
#SBATCH --output={akrrTaskWorkingDir}/stdout
#SBATCH --error={akrrTaskWorkingDir}/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive

. $MODULESHOME/init/sh
"""
```

Batch Job Script Template

AKRR is designed with need to execute multiple application kernels on multiple HPC resources. The complexity of this task in part comes from the fact that various HPC resources can be severely different in hardware and software. For example, HPC resources can use different queueing systems, different vendors and versions of MPI, BLAS and others libraries. Furthermore, some applications can be compiled in number of way which will affect how they should be executed (e.g. TCP/IP vs MPI). This makes impossible to create a single job script which will work on all platforms. A large set of separate job scripts for every application kernel on every HPC resource would be unbearable to maintain. AKRR addresses variability of batch job scripts by the use of templates. Job script is created from template for every new task.

The root template for batch job script is located at $AKRR_HOME/src/default.resource.inp.py and listed bellow:

```
...
#master batch job script template
batchJobTemplate="""{batchJobHeader}

{akrrCommonCommands}

{batchJobSysDepInit}

{akrrCommonTests}

{akrrStartAppKer}

{akrrCommonCleanup}
"""
...
```

The variables in figure brackets will be replaced by their values during the batch job script creation. Double figure brackets will be eventually substituted with a single figure bracket. During the batch job script creation the variables is read from following sources and in shown order:

- Resource configuration (read from file-system):
  - $AKRR_HOME/src/default.resource.inp.py
  - $AKRR_HOME/cfg/resources/{resource}.resource.inp.py
- Application kernel configuration (read from file-system):
  - $AKRR_HOME/src/default.app.inp.py
  - $AKRR_HOME/cfg/apps/{app}.app.inp.py
- Task parameters (read from database for each task):
  - resource_param
  - app_param
  - task_param

The first variable in batchJobTemplate is {batchJobHeader}, which describe the resources requests and other stuff which usually present in the beginning of all job scripts for a given HPC resourse. {batchJobHeader} is defined in resource configuration file for each resource ($AKRR_HOME/cfg/resources/{resource}.resource.inp.py):

```
...
#job script header
batchJobHeader="""#!/bin/sh
#SBATCH --partition=general-compute
#SBATCH --nodes={akrrNNodes}
#SBATCH --ntasks-per-node={akrrPPN}
#SBATCH --time={akrrWallTimeLimit}
#SBATCH --output={akrrTaskWorkingDir}/stdout
#SBATCH --error={akrrTaskWorkingDir}/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive

. $MODULESHOME/init/sh
"""
...
```

This header template will be used for generation of all batch job scripts on that resource. This way if there is a need to update the header (for example set a new charging account) this can be done in one place.

The template variable {akrrStartAppKer} replacement is different from others variables. It will be replaced by runScript[{resource}] from application kernel configuration file ($AKRR_HOME/cfg/apps/{app}.app.inp.py). This done like this because application kernels execution way is very individual for each resource. Below is list of variables which are accessible within templates:

Table 2. Variables accessible within templates

variable name	description	example
`{akrrNNodes}`	number of requested nodes	1
`{akrrWallTimeLimit}`	requested walltime, this field will be properly formated	
to be filled		

Installation Instructions

AKRR Server Deployment

Adding a New HPC Resource

Deployment of Application Kernels on a Resource

Enable Application Kernel Support in OpenXDMoD (see OpenXDMoD documentations)

Usage

e-Mail Report Generation

Scheduling and Rescheduling Application Kernels

Setup Walltime Limit

AKRR: Adding a New HPC Resource

At this step a new HPC resource will be added to AKRR. Briefly the whole procedure is as follows: 1) the resource initial configuration file is created either by importing it from OpenXDMoD or from a provided template, 2) the resource parameters are manually set in the configuration file (some of the parameters are resource characteristics, head-node access credential and batch job script header template) and 3) run resource validation and deployment script; this script will check all the settings, access the head node and deploy application kernel input parameters and run a small test job on computational nodes.

From the AKRR point of view, an HPC resource is a distinct and **homogeneous set** of computational nodes. The resource name should reflect such a set. For example, if cluster "A" in addition to typical general purpose nodes has specialized nodes (large memory, GPU or MIC accelerated nodes), it will be convenient to treat them as a separate resources and to name them as "A", "A_largemem", "A_GPU" and "A_MIC" for general purpose nodes, for large memory nodes, GPU or MIC accelerated nodes respectively. The name of the resource is separated from access node (head node), the later is specified in configuration file.

AKRR uses the user-account under which its is running to access HPC resources. For the access, it will use ssh and scp commands. Because AKRR store passwords as a plain unencrypted text, it is recommended to set-up access with private key with or without pass-phrase

Here the rush HPC cluster was added to XDMoD, the username was nikolays and headnode name was rush.ccr.buffalo.edu. Replace them with your names.

Information to Have On Hand

- Access credential to head node from machine where AKRR daemon is running (username and password or username, private key location on file system and pass-phrase)
- Queuing system type (SLURM, PBS)
- Batch job script header with resource request specifications
- HPC resource specification
    - processes per node count
- HPC resource file-system layout
    - local scratch location
    - network scratch location
    - future location for application kernel input and auxiliary files (should be in persistent location (i.e. not scratch))
    - future location for application kernel run-time files (can be on scratch)

Prerequerements

Setting HPC Resource Default Shell to BASH

AKRR accesses and uses HPC resource as a regular user and as a regular user it has its' preference to shell flavor. It is intended to be used with bash, but in most cases it can likely use csh/tcsh due to similarities. Consult your system user guide or consultants on how to do that, please note that in the majority of large HPC sites the UNIX *chsh* command is not the preferred way.

In case of *rush* it turns out that bash is the default shell.

Initialization of Resource Configuration File

The resource configuration file can be imported from OpenXDMoD or initiated from provided template.

To import resource from OpenXDMoD run following scripts:

```
$AKRR_HOME/setup/scripts/init_new_resource.sh
```

This script will:

- ask to choose a resource from OpenXDMoD resources list

- prompt for AKRR resource name
- ask for a queuing system
- ask for resource access credential
- ask for locations of AKRR working directories on resource
- finally it initiate resource configuration file and populate most of the parameters in it

If the resource is not present in OpenXDMoD resources list enter 0 when prompt for resoure id. When prompt for resource name enter human friendly name as discussed earlier, for example *fatboy_gpu, the name can be different from XDMoD name.*

Tips
If your system is fairly non-standard (for example non-default port for ssh, usage of globus-ssh for access and similar) you can run init_new_resource.sh with -m option:

```
$AKRR_HOME/setup/scripts/init_new_resource.sh -m
```

This option sets a minimalistic interactive session and the generated configuration file must be manually edited.

Below is sample output:

init_new_resource.sh Sample Output

```
[INFO]: Attempting to load installation configuration...

[INFO]: configuration loaded!
[INFO]: Checking python version...
[INFO]: Your python version is just right!
[INFO]: Beginning Initiation of New Resource...
[INFO]: Retrieving Resources from XDMoD Database...

Found following resources from XDMoD Database:

resource_id name
1 teragrid
2 dtf.sdsc.teragrid
3 dtf.ncsa.teragrid
4 dtf.anl.teragrid
5 dtf.caltech.teragrid
1546 teragrid_roaming
1547 lonestar.tacc.teragrid
1548 radon.purdue.teragrid
1549 tiger.iu.teragrid
1550 rachel.psc.teragrid
1551 cloud.purdue.teragrid
1552 datastar.sdsc.teragrid
1553 maverick.tacc.teragrid
1554 lemieux.psc.teragrid
1556 nstg.ornl.teragrid
1582 stand-alone
2215 gpfs-wan.teragrid
2218 lear.purdue.teragrid

[INPUT]: Enter resource_id for import (enter 0 for no match):
0

[INPUT]: Enter AKRR resource name:
rush

[INPUT]: Enter queuing system on resource (slurm or pbs):
slurm

[INPUT]: Enter Resource head node (access node) full name (e.g. headnode.somewhere.org):
[rush] rush.ccr.buffalo.edu
[INPUT]: Enter username for resource access:
[mikola] nikolays
[INFO]: Checking for password-less access
[INFO]: Can access resource without password
```

```
[INFO]: Connecting to rush
[INFO]: Done

[INPUT]: Enter processors (cores) per node count:
8
[INPUT]: Enter location of local scratch (visible only to single node):
[/tmp]
[INFO]: Directory exist and accessible for read/write

[INPUT]: Enter location of local scratch (visible only to all nodes), used for temporary storage of app kernel input/output:
/panasas/scratch/nikolays
Directory rush.ccr.buffalo.edu:/panasas/scratch/nikolays does not exists, will try to create it
[INFO]: Directory exist and accessible for read/write

[INPUT]: Enter future location of app kernels input and executable files:
[/user/nikolays/appker/rush]
Directory rush.ccr.buffalo.edu:/user/nikolays/appker/rush does not exists, will try to create it
[INFO]: Directory exist and accessible for read/write

[INPUT]: Enter future locations for app kernels working directories (can or even should be on scratch space):
[/panasas/scratch/nikolays/akrrdata/rush]
Directory rush.ccr.buffalo.edu:/panasas/scratch/nikolays/akrrdata/rush does not exists, will try to create it
[INFO]: Directory exist and accessible for read/write

[INFO]: Initiating rush at AKRR
[INFO]: Resource configuration is in /home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py
[INFO]: Initiation of new resource is completed.
[INFO]: Edit batchJobHeaderTemplate variable in /home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py
[INFO]: and move to resource validation and deployment step.

Execute following command to setup environment variable for future setup convenience:
export RESOURCE=rush
```

At the end it will ask to execute command to setup environment variable RESOURCE, this variable is only used for convenience during resource setup. Execute following command and replace *rush* with your resource name:

```
export RESOURCE=rush
```

Edit Resource Configuration File

Edit resource parameter file $AKRR_HOME/cfg/resources/$RESOURCE/resource.inp.py . Unless "-m" option was used **the only parameter which should need to be adjusted is *batchJobHeaderTemplate* at the end of the file.**

The initial configuration file will look like:

```
#Resource parameters
info = "template for resources with SLURM queuing system"
#Processors (cores) per node
ppn = 8
#head node for remote access
remoteAccessNode = "rush.ccr.buffalo.edu"
#Remote access method to the resource (default ssh)
remoteAccessMethod = "ssh"
#Remote copy method to the resource (default scp)
remoteCopyMethod = "scp"

#Access authentication
sshUserName = "nikolays"
sshPassword = None
sshPrivateKeyFile = None
sshPrivateKeyPassword = None

#Scratch visible across all nodes (absolute path or/and shell environment variable)
networkScratch = "/panasas/scratch/nikolays"
#Local scratch only locally visible (absolute path or/and shell environment variable)
localScratch = "/tmp"
#Locations for app. kernels working directories (can or even should be on scratch
space)
akrrData = "/panasas/scratch/nikolays/akrrdata/rush"
#Location of executables and input for app. kernels
appKerDir = "/user/nikolays/appker/rush"

#batch options
batchScheduler = "slurm"

#job script header
batchJobHeaderTemplate="""#!/bin/bash
#SBATCH --partition=general-compute
#SBATCH --nodes={akrrNNodes}
#SBATCH --ntasks-per-node={akrrPPN}
#SBATCH --time={akrrWallTimeLimit}
#SBATCH --output={akrrTaskWorkingDir}/stdout
#SBATCH --error={akrrTaskWorkingDir}/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive
"""
```

Configuration File Formatting
All AKRR configuration files utilize python syntax. The parts most often used for configuration are listed bellow:

```
#pound sign for comments

#value assignment to variable
DB_host="127.0.0.1"
exportDB_host=DB_host

#triple quotes for long multi-line strings
batchJobHeaderTemplate="""#!/bin/bash
#SBATCH --partition=general-compute
#SBATCH --nodes={akrrNNodes}
#SBATCH --ntasks-per-node={akrrPPN}
#SBATCH --time={akrrWallTimeLimit}
#SBATCH --output={akrrTaskWorkingDir}/stdout
#SBATCH --error={akrrTaskWorkingDir}/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive
"""
```

Batch job script files are automatically generated using the template. Variables in curly brackets are replaced by their values.

For example line "#SBATCH --nodes={akrrNNodes}" listed above in batchJobHeaderTemplate template variable

will become "#SBATCH --nodes=2" in batch job script if application kernel should run on two nodes.

In order to enter shell curly brackets they should be enter as double curly brackets. All double curly brackets will be replaced with single curly bracket during batch job script generation.

For example "awk "{{for (i=0;i<$_TASKS_PER_NODE;++i)print}}"` in template variable

will become "awk "{for (i=0;i<$_TASKS_PER_NODE;++i)print}"` in batch job script.

The commented parameters will assume default values. Below is the description of the parameters and their default values:

Parameter	Optional	Description	Default Value
ppn	N	Processors (cores) per node	Must be set
remoteAccessNode	N	head node name for remote access	Must be set
remoteAccessMethod	Y	Remote access method to the resource. Default is ssh, gsissh can be used. Here command line options to ssh can be specified as well (e.g. "ssh -p 23")	'ssh'
remoteCopyMethod	Y	Remote copy method to the resource. Default is scp, gsiscp can be used. Here command line options to ssh can be specified as well.	'scp'
Access authentication			
sshUserName	N	username for remote access	Must be set

sshPassword	Y	password	None
sshPrivateKeyFile	Y	location of private key, full name must be used	None
sshPrivateKeyPassword	Y	private key pass-phrase	None
File-system locations on HPC resource			
networkScratch	N	Scratch visible across all computational nodes (absolute path or/and shell environment variable)	'$SCRATCH'
localScratch	N	Local scratch only visible locally to a computational node (absolute path or/and shell environment variable)	'/tmp'
akrrData	N	Top directory for app. kernels working directories. The last has a lifespan of task execution and can or even should be on scratch space. This directory will be automatically created if needed.	Must be set
appKerDir	N	Location of executables and input for app. kernels. The content of this directory will be filled during next step (validation and deployment)	Must be set
Batch job script settings			
batchScheduler	N	Scheduler type: slurm or pbs. sge might work as well but was not tested	Must be set
`batchJobHeaderTemplate`	N	Header for batch job script. Describe the resources requests and set AKRR_NODELIST environment variable containing list of all nodes. *See below for more detailed information.*	Must be set

How to set *batchJobHeaderTemplate*

batchJobHeaderTemplate is a template used in the generation of batch job scripts. It specifies the resources (e.g. number of nodes) and other parameters used by scheduler.

See Batch Job Script Template section of Application Kernels Remote Runner (AKRR) for more details on how AKRR uses templates.

Here is instruction how to convert batch job script header to *batchJobHeader* template. For example, for a resource using Torque/PBS or a close variant, if the **whole** PBS job script for the system of interest looks like:

```
#!/bin/sh
#PBS -l nodes=4:ppn=16:native
#PBS -m n
#PBS -q normal
#PBS -S /bin/sh
#PBS -e stderr
#PBS -o stdout
#PBS -l walltime=00:13:00
#PBS -W x=NACCESSPOLICY:SINGLEJOB
#PBS -u xdtas
#PBS -A ACCOUNT_TO_CHARGE


#Load application environment
module load namd/2.9

namd2_bin=`which namd2`
charmrun_bin=`which charmrun`

#Execute AppKer
mpirun_rsh -n 64 -hostfile $PBS_NODEFILE $namd2_bin ./input.namd >>
$AKRR_APP_STDOUT_FILE 2>&1
```

Then we need to cut the top part of it, use it to replace the top section in the *batchJobHeaderTemplate* variable, and replace the requested resources with suitable template variables:

```
batchJobHeaderTemplate="""#!/bin/sh
#PBS -l nodes={akrrNNodes}:ppn={akrrPPN}:native
#PBS -m n
#PBS -q normal
#PBS -S /bin/sh
#PBS -e stderr
#PBS -o stdout
#PBS -l walltime={akrrWallTimeLimit}
#PBS -W x=NACCESSPOLICY:SINGLEJOB
#PBS -u xdtas
#PBS -A ACCOUNT_TO_CHARGE
"""
```

Specific number of nodes become {akrrNNodes}, processors per node become {akrrPPN} and walltime becomes {akrrWallTimeLimit}. These template variables will be substituted by the desired values during generation of batch job script for a particular task. Note the presence of module system initiations (module system might not be initiated for multiple reasons, usually explicit initiation solves most of them). The name of the files to where the standard output and error are redirected **always** should be stdout and stderr respectively.

Some template variable often used in batchJobHeaderTemplate is shown in table below:

Variable Name	Description
{akrrNNodes}	Number of requested nodes
{akrrPPN}	Processors per node count, that means a total count of cores on a single node
{akrrNCores}	Number of requested cores

`{akrrWallTimeLimit}`	Requested walltime, this field will be properly formatted

{akrrTaskWorkingDir}	Location of working directory where the application kernel will be executed. It is often used to redirect standard error and output to proper location, e.g.:
	`#SBATCH --output={akrrTaskWorkingDir}/stdout`
	`#SBATCH --error={akrrTaskWorkingDir}/stderr`
	Such explicit definition of standard error and output redirected files are rarely used because AKRR submit jobs from this directory, and the default location of redirected standard error and output file are usually specified in reference to that working directory. Some batch systems, however, have been known to default to placing such output files in the user $HOME directory rather than the job submission directory.

Visual Inspection of Generated Batch Job Script

Now, we can generate test application kernel batch job script and visually inspect it for mistake presence. Run:

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a test -n 2
```

This command will generate batch job script and output it to standard output. Below is example of the output

akrr_ctl.sh batch_job -p -r $RESOURCE -a test -n 2 Sample output

```
[INFO]: Below is content of generated batch job script:
#!/bin/bash
#SBATCH --partition=general-compute
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:02:00
#SBATCH --output=/panasas/scratch/nikolays/akrrdata/rush/test/2014.12.11.10.23.05.105683/stdout
#SBATCH --error=/panasas/scratch/nikolays/akrrdata/rush/test/2014.12.11.10.23.05.105683/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive


#Node list setter
#Populate list of nodes per MPI process
_TASKS_PER_NODE=`echo $SLURM_TASKS_PER_NODE|sed "s/(x[0-9]*)//g"`
export AKRR_NODELIST=`scontrol show hostname $SLURM_NODELIST| awk "{for (i=0;i<$_TASKS_PER_NODE;++i)print}"`


#Common commands
export AKRR_NODES=2
export AKRR_CORES=16
export AKRR_CORES_PER_NODE=8
export AKRR_NETWORK_SCRATCH="/panasas/scratch/nikolays"
export AKRR_LOCAL_SCRATCH="/tmp"
export AKRR_TASK_WORKDIR="/panasas/scratch/nikolays/akrrdata/rush/test/2014.12.11.10.23.05.105683"
export AKRR_APPKER_DIR="/user/nikolays/appker/rush"
export AKRR_AKRR_DIR="/panasas/scratch/nikolays/akrrdata/rush"

export AKRR_APPKER_NAME="test"
export AKRR_RESOURCE_NAME="rush"
export AKRR_TIMESTAMP="2014.12.11.10.23.05.105683"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"

export AKRR_APPKERNEL_INPUT="/user/nikolays/appker/rush/inputs"
export AKRR_APPKERNEL_EXECUTABLE="/user/nikolays/appker/rush/execs"

source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"

export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"
```

```
cd "$AKRR_TASK_WORKDIR"

#run common tests
akrrPerformCommonTests

#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"



#normally in runScriptPreRun
#create working dir
export AKRR_TMP_WORKDIR=`mktemp -d /panasas/scratch/nikolays/test.XXXXXXXXX`
echo "Temporary working directory: $AKRR_TMP_WORKDIR"
cd $AKRR_TMP_WORKDIR

#Generate AppKer signature
appsigcheck.sh /user/nikolays/appker/rush/execs/appsig/libelf/appsiggen > $AKRR_APP_STDOUT_FILE

echo "Checking that the shell is BASH"
echo $BASH


#normally in runScriptPostRun
#clean-up
cd $AKRR_TASK_WORKDIR
if [ "${AKRR_DEBUG=no}" = "no" ]
then
echo "Deleting temporary files"
rm -rf $AKRR_TMP_WORKDIR
else
echo "Copying temporary files"
cp -r $AKRR_TMP_WORKDIR workdir
rm -rf $AKRR_TMP_WORKDIR
fi


writeToGenInfo "endTime" "`date`"

[INFO]: Removing generated files from file-system as only batch job script printing was requested
```

Test application kernel is specialized application kernel which inspects the resource deployment. Here mainly inspect the very top of the generated script and check is the resources request is generated properly. Modify batchJobHeaderTemplate in configuration file if needed.

Resource Parameters Validation and Application Kernel Input Parameters Deployment

The following command will validate resource parameters and deploy application kernel input parameters

```
python $AKRR_HOME/setup/scripts/resource_validation_and_deployment.py $RESOURCE
```

This script will perform following operations:

- Check configuration file syntax, parameters type and presence of non optional parameters
- Test the connectivity to the head-node
- Deploy application kernel input parameters and application signature calculator
- Run a test job on the resource

The script will exit in case of failure. The error must be addressed and script must be rerun until successful execution. Below is example of successful execution:

```

```

```
################################################################################
Validating rush parameters from /home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py
Syntax of /home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py is correct and all necessary parameters are present.
################################################################################
Validating resource accessibility. Connecting to rush.
================================================================================
Successfully connected to rush


Checking if shell is BASH

Shell is BASH

Checking directory locations

Checking: rush:/panasas/scratch/nikolays/akrrdata/rush
Directory exist and accessible for read/write

Checking: rush:/user/nikolays/appker/rush
Directory exist and accessible for read/write

Checking: rush:/panasas/scratch/nikolays
Directory exist and accessible for read/write

Checking: rush:/tmp
Directory exist and accessible for read/write

################################################################################
Preparing to copy application signature calculator,
app. kernel input files and
HPCC,IMB,IOR and Graph500 source code to remote resource


Copying app. kernel input tarball to /user/nikolays/appker/rush
scp /home/mikola/wsp/test/akrr/setup/scripts/../../appker_repo/inputs.tar.gz nikolays@rush:/user/nikolays/appker/rush
Unpacking app. kernel input files to /user/nikolays/appker/rush/inputs
App. kernel input files are in /user/nikolays/appker/rush/inputs

Copying app. kernel execs tarball to /user/nikolays/appker/rush
It contains HPCC,IMB,IOR and Graph500 source code and app.signature calculator
scp /home/mikola/wsp/test/akrr/setup/scripts/../../appker_repo/execs.tar.gz nikolays@rush:/user/nikolays/appker/rush
Unpacking HPCC,IMB,IOR and Graph500 source code and app.signature calculator files to /user/nikolays/appker/rush/execs
HPCC,IMB,IOR and Graph500 source code and app.signature calculator are in /user/nikolays/appker/rush/execs

Testing app.signature calculator on headnode

App.signature calculator is working on headnode

################################################################################
Will send test job to queue, wait till it executed and will analyze the output
Will use AKRR REST API at https://localhost:8091/api/v1

Submitted test job to AKRR, task_id is 3144529



================================================================================
Tast status:
Task is in scheduled_tasks queue.
It schedule to be started on2014-12-11 10:40:05

time: 2014-12-11 10:40:05

================================================================================
Tast status:
Task is in active_tasks queue.
Status: None
Status info:
None

time: 2014-12-11 10:40:10
```

```
================================================================================
Tast status:
Task is in active_tasks queue.
Status: Created batch job script and have submitted it to remote queue.
Status info:
Remote job ID is 3107259

time: 2014-12-11 10:40:15

================================================================================
Tast status:
Task is in active_tasks queue.
Status: Still in queue. Either waiting or running
Status info:
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
3107259 general-c test.job nikolays R 0:05 2 d07n06s01,d07n08s01


time: 2014-12-11 10:45:05

================================================================================
Tast status:
Task is in active_tasks queue.
Status: Output was processed and found that kernel either exited with error or executed successfully.
Status info:
Done

time: 2014-12-11 10:45:11

================================================================================
Tast status:
Task is completed!
status: 1
statusinfo: Done

time: 2014-12-11 10:45:17


################################################################################
Test job is completed analyzing output


Test kernel execution summary:
status: 1
statusinfo: Done
processing message:
None
Local working directory for this task: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163
Location of some important generated files:
Batch job script: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/jobfiles/test.job
Application kernel output: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/jobfiles/appstdout
Batch job standard output: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/jobfiles/stdout
Batch job standard error output: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/jobfiles/stderr
XML processing results: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/result.xml
Task execution logs: /home/mikola/wsp/test/akrr/comptasks/rush/test/2014.12.11.10.40.06.033163/proc/log

The output looks good.

Adding AKRR enviroment variables to resource's .bashrc!

Enabled rush in mod_appkernel.resource for tasks execution and made it visible to XDMoD UI.
################################################################################
Result:

DONE, you can move to next step!
```

Troubleshooting

Incorrect $AKRR_NODELIST Environment variable

If you got following error messages:

"Nodes are not detected, check batchJobTemplate and setup of AKRR_NODELIST variable"

"Can not ping compute nodes from head node"

"Number of requested processes (processes per node * nodes) do not match actual processes executed"

Then there is a high chances that AKRR_NODELIST was not set properly from default templates.

AKRR_NODELIST is a list of nodes per each MPI process, i.e. same node name is repeated multiple times. For example for 2 node run on 4 cores per node machine it looks like "node3 node3 node3 node3 node7 node7 node7 node7".

By default AKRR uses templates specific to queuing system (defined in $AKRR_HOME/src/default.resource.inp.py):

```
#Node list setter
nodeListSetter={
    'pbs':"""export AKRR_NODELIST=`cat $PBS_NODEFILE`""",
    'slurm':"""export AKRR_NODELIST=`srun -l --ntasks-per-node=$AKRR_CORES_PER_NODE -n
$AKRR_CORES hostname -s|sort -n| awk '{{printf "%s ",$2}}' `"""
}
```

To modify the behavior nodeListSetterTemplate can be define in specific resource configuration file ($AKRR_HOME/cfg/resources/$RESOURCE/resource.inp.py):

portion of $AKRR_HOME/cfg/resources/$RESOURCE/resource.inp.py

```
#Node list setter
nodeListSetterTemplate="""export AKRR_NODELIST=`srun -l
--ntasks-per-node=$AKRR_CORES_PER_NODE -n $AKRR_CORES hostname -s|sort -n| awk
'{{printf "%s ",$2}}' `"""
```

For SLURM alternative to srun can be:

portion of $AKRR_HOME/cfg/resources/$RESOURCE/resource.inp.py

```
#Node list setter
nodeListSetterTemplate="""_TASKS_PER_NODE=`echo $SLURM_TASKS_PER_NODE|sed
"s/(x[0-9]*)//g"`
export AKRR_NODELIST=`scontrol show hostname $SLURM_NODELIST| awk "{{for
(i=0;i<$_TASKS_PER_NODE;++i)print}}"
"""
```

Advanced Debugging

Although resource_validation_and_deployment.py detects many problems with resource deployments, sometimes its output can be cryptic. The following strategy can be employed to find the problem.

1. Generate batch job script

2. Run it on resource
3. Analyze output
4. Fix the issues in batch job script
5. Go to 2 until executed successfully
6. merge changes in batch job script to respective template in configuration file

Batch job script can be generated by running following command:

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a test -n 2
```

This command generate batch job script and copy it to proper location on remote resource. This location will be showed in output:

```
[INFO]: Local copy of batch job script is
/home/mikola/wsp/test/akrr/data/rush/test/2014.12.11.08.58.57.412410/jobfiles/test.job

[INFO]: Application kernel working directory on rush is
/panasas/scratch/nikolays/akrrdata/rush/test/2014.12.11.08.58.57.412410
[INFO]: Batch job script location on rush is
/panasas/scratch/nikolays/akrrdata/rush/test/2014.12.11.08.58.57.412410/test.job
```

Now log into resource, go to the task working directory and manually submit to queue, check the output and determine the problem.

Now AKRR can submit jobs to that resource

Next step is AKRR: Deployment of Application Kernel on resource

AKRR: Deployment of Applications Kernels on a Resource

HPC resources differ significantly in terms of the software stack. For example, different resources (and applications running on those resources) can utilize different compilers (gcc or icc) and MPI libraries (OpenMPI, MVAPICH, or a commercial variant). Furthermore, the same application can be compiled in a number of ways which can greatly vary in how they are executed. Therefore beside general application kernel configuration, each application kernel needs to be separately configured for each resource.

The overall strategy for deploying an application kernel to a resource is following:

1. Install application or identify where it is already installed.
2. Generate initial configuration file.
3. Edit configuration file.
4. (Optional) Generate batch job script and execute it manually, update configuration file as needed.
5. Perform validation run, update configuration file as needed.
6. Schedule regular execution of application kernel.

First, each step will be described in general as it applies to all application kernels and the details on example individual application kernels deployment will follow.

Install Application or Identify where it is Installed

AKRR comes with two flavors of application kernels. One is based on real-world applications and another is based on benchmarks. Real-world applications are often already installed system-wide on a resource for the use of regular users, here one of the purposes of application kernels is to monitor the performance of a standard often-used application on that resource. Benchmarks, however, are rarely installed system-wide and thus they need to be installed first.

Generate Initiate Configuration File

The initial configuration file is generated with *gen_appker_on_resource_cfg* utility. It will generate an initial configuration file and place it to $AKRR_HOME/cfg/resource/<app kernel name>.app.inp.py.

Edit Configuration File

The generated configuration file is fairly generic. Here you need to specify proper execution environment and specify how to execute this particular application kernel on this particular machine/resource.

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to a correct (and workable) batch job script. First the batch job script is generated with **'akrr_ctl.sh batch_job'**. Then this script is executed in an interactive session (this improves the turn-around in case of errors). If the script fails to execute, the issues can be fixed first in that script itself and then merged with the configuration file.

This step is somewhat optional because it is very similar to the next step. However the opportunity to work in an interactive session will often improve the turn-around time because there is no need to stay in queue for each iteration.

Perform Validation Run

For this step **appkernel_validation.py** utility is used to validate application kernel installation on particular resource. It execute the application kernel and analyses its results. If it fails the problems need to be fixed and another round of validation should be performed

Schedule regular execution of application kernel.

Finally, if validation was successful the application kernel can be submited for regular execution on that resource.

Details on the Individual Application Kernels Deployment

AKRR: Deployment of NAMD Applications Kernels on a Resource

AKRR: Deployment of IOR Applications Kernels on a Resource

AKRR: Deployment of IMB Applications Kernels on a Resource

AKRR: Deployment of NWChem Applications Kernels on a Resource

AKRR: Deployment of HPCC Applications Kernels on a Resource

AKRR: Creating New Application Kernel

AKRR: Creating New Application Kernel

In this tutorial we will create a new application kernel, an example that did not previously exist in the AKRR distribution.

The outline of new application kernel creation is as follows:

1. Install new application and input files on target resource
2. Add generic instructions on application kernel execution to AKRR
3. Add resource specific instructions to AKRR
4. Create a parser for new application kernel
5. Test new application kernel

MPI-Pi, Monte-Carlo -Calculation Application Kernel

In this tutorial we will use a classic MPI-Pi calculation program as an application kernel. Reading input from a file and some timing functionality was added to a standard mpi_pi program in order to illustrate a practical example on integration of a new application kernel.

MPI-Pi calculates pi using a Monte-Carlo method: darts are randomly thrown to a unit square target with a round unit circle dartboard on it, the ratio of darts that hit the dartboard to the total number of darts thrown gives the ratio of areas between circle dartboard and the square target, which is then the Monte Carlo estimate for pi.

For our MPI-Pi application kernel to resemble what one can encounter during creation of more complex application kernels we have augmented the classical MPI-Pi program with features usually seen in much bigger programs:

1. read number of darts to throw and total number of rounds from an input file
2. print out input parameters
3. print out version number

4. timing function to measure time spend in pi calculation
5. created performance metric, Darts thrown Per Second (DaPS)

Below is MPI-Pi installation reference and output sample:

source code	only one file download from here: mpi_pi_reduce.c
installation	compile it directly: mpicc -o mpi_pi_reduce -O3 mpi_pi_reduce.c
running	mpirun -n <number_of_processors> <path_to>/mpi_pi_reduce <input file>
input file example	5000000 /* number of throws at dartboard */ 100 /* number of times "darts" is iterated */

output example	<pre>MPI task 0 has started...
	MPI task 1 has started...
	MPI task 2 has started...
	MPI task 3 has started...
	MPI task 5 has started...
	MPI task 4 has started...
	version: 2015.3.9
	number of throws at dartboard:
	5000000
	number of rounds for dartz throwing
	100
	After 5000000 throws, average value
	of pi = 3.14172000
	After 10000000 throws, average value
	of pi = 3.14158253
	....
	After 495000000 throws, average value
	of pi = 3.14158363
	After 500000000 throws, average value
	of pi = 3.14158676
	Real value of PI: 3.1415926535897
	Time for PI calculation: 8.011
	Giga Darts Throws per Second (GDaPS):
	0.374</pre>

For further convenience lets set RESOURCE and APPKER environment variables:

```
export RESOURCE="rush"
export APPKER="mpipi"
```

Install new application kernel executables and input files on target resource

First we need to install the application on target resource:

```
#for further convenience lets set RESOURCE and APPKER
export RESOURCE="rush"
export APPKER="mpipi"

#on remote resource
#go to application kernel executable directory
cd $AKRR_APPKER_DIR/execs

#make new directory for application kernel executable
mkdir mpipi
cd mpipi

#get the source code
wget http://compmodel.org/mpi_pi/mpi_pi_reduce.c

#load your favorite MPI compiler
module load intel-mpi/5.0.2

#compile mpipi
mpicc -o mpi_pi_reduce -O3 mpi_pi_reduce.c
```

The source code for mpi_pi can also be found at mpi_pi_reduce.c

Second we need to install the input file on the target resource:

```
#on remote resource
#go to application kernel executable directory
cd $AKRR_APPKER_DIR/inputs

#make new directory for application kernel executable
mkdir mpipi
cd mpipi

#create input file
cat > input_file << END
5000000 /* number of throws at dartboard */
100 /* number of times "darts" is iterated */
END
```

Add general instructions on application kernel execution to AKRR

Now we need to add instructions for the new app kernel execution (which holds for most resources):

```
#on AKRR server machine
#go to AKRR directory
cd $AKRR_HOME
#get to generic application kernel directory
cd src/appkernels
#initiate new app kernel entry from a template
cp ../templates/template.app.inp.py $APPKER.app.inp.py
```

Edit $AKRR_HOME/src/appkernels/mpipi.app.inp.py, at this point we only need to set executable and input:

$AKRR_HOME/src/appkernels/mpipi.app.inp.py

```
#default walltime for all resources
walllimit=13

#parser to process output
parser="xdmod.generic.py"

#path to run script relative to AppKerDir on particular resource
executable="execs/mpipi/mpi_pi_reduce"

#inputs for application kernel relative to AppKerDir on particular resource
input="inputs/mpipi/input_file"
```

At first we will just use the standard parser which will only give us the walltime and the nodes on which the application kernel was executed. Later on we will customize the mpi_pi specialized parser.

Add resource specific instruction to AKRR

Now we need to create specific instructions how to execute the app kernel on that particular resource:

```
#on AKRR server machine
#go to AKRR directory
cd $AKRR_HOME
#get to resource configuration directory of the generic application kernel directory
cd cfg/resources/$RESOURCE
#initiate new app kernel entry from a template
cp ../../../src/templates/resource.app.inp.py ./$APPKER.app.inp.py
```

Edit cfg/resources/$RESOURCE/$APPKER.app.inp.py, the final file for this particular resource (using Slurm and its integrated task launcher) looks like:

$AKRR_HOME/src/appkernels/mpipi.app.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application enviroment
module load intel-mpi
module list
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

#set how to run app kernel
RUN_APPKERNEL="srun {appKerDir}/{executable} ./input_file"
"""
```

Now we can make a first run

```
#on AKRR server

#check the resulting batch file
$AKRR_HOME/bin/akrr_ctl.sh batch_job -r edge12core -a mpipi -n 2

#make a first run
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 edge12core mpipi
```

If the last command executed successfully then even with the generic parser we already can use that app kernel

```
#restart AKRR server
$AKRR_HOME/bin/akrr.sh stop
$AKRR_HOME/bin/akrr.sh start

#on AKRR server
#examine generated batch job file
$AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2

#make a first run
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 $RESOURCE $APPKER
```

Record the location of job files directory (e.g. /somewhere/akrr/comptasks/rush/mpipi/2015.03.18.14.38.57.281990/jobfiles) we will use for parser development.

Create a customized parser for new application kernel

Until now we have been using a default parser which only gives walltime and the execution node names. However we want to add more parameters and metrics to be monitored. In order to do so we need to create a customized parser.

Initialize the new parser from generic parser:

```
#on AKRR server, get to parsers directory
cd $AKRR_HOME/src/appkernelsparsers
#Initiate new parser from generic parser
cp xdmod.generic.py mpipi_parser.py
```

Examine mpipi_parser.py

mpipi_parser.py as initialized from xdmod.generic.py

```
 # Generic Parser
import re
import os
import sys
#Set proper path for stand alone test runs
if __name__ == "__main__":

sys.path.append(os.path.join(os.path.dirname(os.path.realpath(__file__)),'../../src'))
from akrrappkeroutputparser import AppKerOutputParser,testParser

def
processAppKerOutput(appstdout=None,stdout=None,stderr=None,geninfo=None,appKerNResVars
```

```
=None):
    #set App Kernel Description
    if(appKerNResVars!=None and 'app' in appKerNResVars and 'name' in
appKerNResVars['app']):
        akname=appKerNResVars['app']['name']
    else:
        akname='unknown'

    #initiate parser
    parser=AppKerOutputParser(
        name             = akname
    )
    #set obligatory parameters and statistics
    #set common parameters and statistics (App:ExeBinSignature and RunEnv:Nodes)
    parser.setCommonMustHaveParsAndStats()
    #set app kernel custom sets
    #parser.setMustHaveParameter('App:Version')
    parser.setMustHaveParameter('RunEnv:Nodes')

    parser.setMustHaveStatistic('Wall Clock Time')
    #parse common parameters and statistics
    parser.parseCommonParsAndStats(appstdout,stdout,stderr,geninfo)
    if hasattr(parser,'appKerWallClockTime'):
        parser.setStatistic("Wall Clock Time",
parser.appKerWallClockTime.total_seconds(), "Second")

    #Here can be custom output parsing
#     #read output
#     lines=[]
#     if os.path.isfile(appstdout):
#         fin=open(appstdout,"rt")
#         lines=fin.readlines()
#         fin.close()
#
#     #process the output
#     parser.successfulRun=False
#     j=0
#     while j<len(lines):
#         m=re.search(r'My mega parameter\s+(\d+)',lines[j])
#         if m:parser.setParameter("mega parameter",m.group(1))
#
#         m=re.search(r'My mega parameter\s+(\d+)',lines[j])
#         if m:parser.setStatistic("mega statistics",m.group(1),"Seconds")
#
#         m=re.search(r'Done',lines[j])
#         if m:parser.successfulRun=False
#
#         j+=1


    if __name__ == "__main__":
        #output for testing purpose
        print "Parsing complete:",parser.parsingComplete(Verbose=True)
        print "Following statistics and parameter can be set as obligatory:"
        parser.printParsNStatsAsMustHave()
        print "Resulting XML:"
        print parser.getXML()

    #return complete XML otherwise return None
```

```
    return parser.getXML()


if __name__ == "__main__":
    """stand alone testing"""
    jobdir=sys.argv[1]
    print "Proccessing Output From",jobdir
```

```
    testParser(jobdir,processAppKerOutput)
```

You need to modify the processAppKerOutput function to include custom output processing. The processAppKerOutput function consists of several parts:

1. Parser initiation
2. Setting of obligatory parameters and statistics
3. Processing of application output

Processing of application output

Find the commented section in mpipi_parser.py, it shows an example on output processing. The idea is to read application output (the output file set from appstdout argument) and with the help of regular expressions extract the parameters and metrics which we need.

Here is a sample output form the mpipi application

```
Here is sample output:
MPI task 0 has started...
MPI task 1 has started...
MPI task 2 has started...
MPI task 3 has started...
MPI task 5 has started...
MPI task 4 has started...
version: 2015.3.9
number of throws at dartboard: 5000000
number of rounds for dartz throwing 100
After 5000000 throws, average value of pi = 3.14172000
After 10000000 throws, average value of pi = 3.14158253
....
After 495000000 throws, average value of pi = 3.14158363
After 500000000 throws, average value of pi = 3.14158676
Real value of PI: 3.1415926535897
Time for PI calculation: 8.011
Giga Darts Throws per Second (GDaPS): 0.374
```

Bellow is table of what we want to extract:

Name	Units	Parameter or Statistics	Line in output	Notes
App:Version	-	Parameter	version: 2015.3.9	
Number of Darts Throws	-	Parameter	number of throws at dartboard: 5000000	
Number of Rounds	-	Parameter	number of rounds for dartz throwing 100	
Time for PI Calculation	Seconds	Statistics	Time for PI calculation: 8.011	
Darts Throws per Second	GDaPS	Statistics	Giga Darts Throws per Second (GDaPS): 0.374	

The parser can be immediately tested on our previous run:

```
#on AKRR server
python $AKRR_HOME/src/appkernelsparsers/mpipi_parser.py
$AKRR_HOME/comptasks/rush/mpipi/2015.03.18.14.38.57.281990/jobfiles
```

The output of generic parser is:

```
Proccessing Output From
/home/mikola/work/akrr_ci/akrr/comptasks/rush/mpipi/2015.03.18.14.33.43.631495/jobfile
s
Read pickeled task handler from:

/home/mikola/work/akrr_ci/akrr/comptasks/rush/mpipi/2015.03.18.14.33.43.631495/proc/00
0005.st
Parsing complete: True
Following statistics and parameter can be set as obligatory:
parser.setMustHaveParameter('App:ExeBinSignature')
parser.setMustHaveParameter('RunEnv:Nodes')
parser.setMustHaveStatistic('Wall Clock Time')

Resulting XML:
<?xml version="1.0" ?>
<rep:report xmlns:rep="report">
  <body>
    <performance>
      <ID>mpipi</ID>
      <benchmark>
        <ID>mpipi</ID>
        <parameters>
          <parameter>
            <ID>App:ExeBinSignature</ID>

<value>H4sIAOvICVUCAwXBUQ7CIAwA0P+doidoh2VG/ffbhBt0yFwzZKSwxN3e954hvMID6FOXRi2a9LhS0W3
PcjaSzewtXciOttK3alW6jG7CkdHd0HlkRs94ZefvEyGCNih7BymQfikeXeacYNYidsKiOQ3DH4w+TgFyAAAA<
/value>
          </parameter>
          <parameter>
            <ID>RunEnv:Nodes</ID>
            <value>H4sIAOvICVUCA0sxNM4zMFFIIYuyJJ3iAgAvFgslcQAAAA==</value>
          </parameter>
        </parameters>
        <statistics>
          <statistic>
            <ID>Wall Clock Time</ID>
            <value>25.0</value>
            <units>Second</units>
          </statistic>
        </statistics>
      </benchmark>
    </performance>
  </body>
  <exitStatus>
    <completed>true</completed>
  </exitStatus>
</rep:report>
```

The XML at the end will be eventually dumped to DB for further ingestion by OpenXDMoD.

Now, modify the output parsing part. Note the parser.successfulRun variable it should be set to true if a measure of successful execution was found and to false otherwise. After output processing is done you also can add parser.setMustHaveParameter and parser.setMustHaveStatistic. If the parameters/statistics specified by such function are not set the parser will treat the run as a failure. This way we try to ensure that all necessary metrics are set.

The final parser looks like:

mpipi_parser.py

```
# Generic Parser
import re
import os
import sys
#Set proper path for stand alone test runs
if __name__ == "__main__":

sys.path.append(os.path.join(os.path.dirname(os.path.realpath(__file__)),'../../src'))
from akrrappkeroutputparser import AppKerOutputParser,testParser

def
processAppKerOutput(appstdout=None,stdout=None,stderr=None,geninfo=None,appKerNResVars
=None):
    #set App Kernel Description
    if(appKerNResVars!=None and 'app' in appKerNResVars and 'name' in
appKerNResVars['app']):
        akname=appKerNResVars['app']['name']
    else:
        akname='unknown'

    #initiate parser
    parser=AppKerOutputParser(
        name             = akname
    )
    #set obligatory parameters and statistics
    #set common parameters and statistics (App:ExeBinSignature and RunEnv:Nodes)
    parser.setCommonMustHaveParsAndStats()
    #set app kernel custom sets
    parser.setMustHaveParameter('App:ExeBinSignature')
    parser.setMustHaveParameter('App:Version')
    parser.setMustHaveParameter('Number of Darts Throws')
    parser.setMustHaveParameter('Number of Rounds')
    parser.setMustHaveParameter('RunEnv:Nodes')

    parser.setMustHaveStatistic('Darts Throws per Second')
    parser.setMustHaveStatistic('Time for PI Calculation')
    parser.setMustHaveStatistic('Wall Clock Time')
    #parse common parameters and statistics
    parser.parseCommonParsAndStats(appstdout,stdout,stderr,geninfo)
    if hasattr(parser,'appKerWallClockTime'):
        parser.setStatistic("Wall Clock Time",
parser.appKerWallClockTime.total_seconds(), "Second")

    #Here can be custom output parsing
    #read output
    lines=[]
    if os.path.isfile(appstdout):
```

```
        fin=open(appstdout,"rt")
        lines=fin.readlines()
        fin.close()

    #process the output
    parser.successfulRun=False
    j=0
    while j<len(lines):
        m=re.search(r'version:\s+(.+)',lines[j])
        if m:parser.setParameter('App:Version',m.group(1))

        m=re.search(r'number of throws at dartboard:\s+(\d+)',lines[j])
        if m:parser.setParameter('Number of Darts Throws',m.group(1))

        m=re.search(r'number of rounds for dartz throwing\s+(\d+)',lines[j])
        if m:parser.setParameter('Number of Rounds',m.group(1))

        m=re.search(r'Time for PI calculation:\s+([0-9\.]+)',lines[j])
        if m:parser.setStatistic("Time for PI Calculation",m.group(1),"Seconds")

        m=re.search(r'Giga Darts Throws per Second \(GDaPS\):\s+([0-9\.]+)',lines[j])
        if m:parser.setStatistic("Darts Throws per Second",m.group(1),"GDaPS")
        m=re.search(r'Giga Darts Throws per Second',lines[j])
        if m:parser.successfulRun=True

        j+=1


    if __name__ == "__main__":
        #output for testing purpose
        print "Parsing complete:",parser.parsingComplete(Verbose=True)
        print "Following statistics and parameter can be set as obligatory:"
        parser.printParsNStatsAsMustHave()
        print "\nResulting XML:"
        print parser.getXML()

    #return complete XML otherwise return None
    return parser.getXML()


if __name__ == "__main__":
    """stand alone testing"""
    jobdir=sys.argv[1]
    print "Proccessing Output From",jobdir
```

```
    testParser(jobdir,processAppKerOutput)
```

Now we can have the mpipi application kernel use mpipi_parser.py. Modify the parser variable in $AKRR_HOME/src/appkernels/mpipi.app.inp.py.

Finally we can make a test run with the new parser, but first we need to restart the AKRR server (AKRR only checks for modifications in $AKRR_HOME/cfg/resources/):

```
#on AKRR server
#restart AKRR server
$AKRR_HOME/bin/akrr.sh stop
$AKRR_HOME/bin/akrr.sh start

#run new test
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 rush mpipi
```

Finally we need to copy the resource specific application kernel configuration to the template directory to reuse it in the future for deployment on a different resource

```
#on AKRR server
cp $AKRR_HOME/cfg/resources/$RESOURCE/mpipi.app.inp.py $AKRR_HOME/src/templates/
```

Deployment of Application Kernel on a Different Resource

Below is an outline for deploying our new application kernel on a different resource:

1) Install executables and input files to resource (see "Install new application kernel executables and input files on target resource" section of this page)

2) Generate resource specific application kernel configuration

```
python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
```

3) Edit resource specific application kernel configuration ($AKRR_HOME/cfg/resources/$RESOURCE/$APPKER.inp.py)

4) Check generated batch job

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
```

5) Perform validation run

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 $RESOURCE $APPKER
```

AKRR: Deployment of HPCC Applications Kernels on a Resource

Here the deployment of xdmod.benchmark.hpcc application kernel is described. This application kernel is based on HPCC benchmark.

For simplicity lets define APPKER and RESOURCE enviroment varible which will contain the HPC resource name:

```
export RESOURCE=rush
export APPKER=xdmod.benchmark.hpcc
```

Expand source

- Installing HPCC
- Generate Initiate Configuration File
- Edit Configuration File
- Generate Batch Job Script and Execute it Manually (Optional)
- Perform Validation Run
- Schedule regular execution of application kernel.

Installing HPCC

In this section example of HPCC installation process will be described, see also HPCC benchmark documentation for more installation details (http://icl.cs.utk.edu/hpcc/).

```
#Go to Application Kernel executable directory
cd $AKRR_APPKER_DIR/execs


#Load modules
module load intel
module load intel-mpi


#We need to make bench of interfaces to mkl library, unfortunately they are not precompiled and
the
#makefiles utilize the relative file-tree.
#So instead copying only interface and mess with makefile, we will copy the whole thing
cd lib

cp -r /util/academic/intel/composer_xe_2013.5.192/mkl ./

#make fftw2x interface to mkl
cd mkl/interfaces/fftw2x_cdft/

make libintel64 PRECISION=MKL_DOUBLE interface=ilp64
#results in ../../lib/intel64/libfftw2x_cdft_DOUBLE_ilp64.a
#make FFTW C wrapper library
cd ../fftw2xc/
make libintel64 PRECISION=MKL_DOUBLE
#results in ../../lib/intel64/libfftw2xc_double_intel.a

#get the code
cd ../../../..
wget http://icl.cs.utk.edu/projectsfiles/hpcc/download/hpcc-1.4.2.tar.gz
tar xvzf hpcc-1.4.2.tar.gz
cd hpcc-1.4.2
#prepare make file
#you can start with something close from hpl/setup directory
#or start from one of our make file
#place Make.intel64_edge12core file to hpcc-1.4.2/hpl
cd hpl
```

```
wget http://compmodel.org/akrr/Make.intel64_edge12core
#edit Make.intel64_edge12core to fit your system (mainly MKLROOT, which specify mkl location)
cd ..

#compile hpcc in hpcc-1.4.2 root directory, run
make arch=intel64_edge12core
#if it is compiled successfully hpcc binary should appear in hpcc-1.4.2 directory


#create a link to hpcc
cd ..
ln -s hpcc-1.4.2 hpcc



#now we can test it on headnode (optional)
cd hpcc
mv _hpccinf.txt hpccinf.txt
MKLROOT=/user/akrr/appker/Rush-debug/execs/lib/mkl
source $MKLROOT/bin/mklvars.sh intel64
mpirun -np 4 ./hpcc
cat hpccoutf.txt
```

```
Make.intel64_edge12core
```

Generate Initiate Configuration File

Generate Initiate Configuration File:
On AKRR server

```
> python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
[INFO]: Generating application kernel configuration for xdmod.benchmark.hpcc on Rush-debug
[INFO]: Application kernel configuration for xdmod.benchmark.hpcc on Rush-debug is in:
        /user/akrr/akrr/cfg/resources/Rush-debug/xdmod.benchmark.hpcc.app.inp.py
```

Edit Configuration File

In contrast to others application kernels IMB uses only one process per node. Therefore during configuration of IMB it is importent to ensure that only one process is run per node.

Below is a listing of configuration file located at $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.mpi.imb.app.inp.py for SLURM:
$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.hpcc.app.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application enviroment
module load intel
module load intel-mpi
module list
MKLROOT=/user/akrr/appker/Rush-debug/execs/lib/mkl
source $MKLROOT/bin/mklvars.sh intel64
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
ulimit -s unlimited
#set how to run app kernel
RUN_APPKERNEL="srun {appKerDir}/{executable}"
"""
```

Here srun with --ntasks-per-node=1 is used to set one process per node execution.

Below is an example for PBS:
$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application enviroment
module swap mvapich2 impi
module list
#set how to run app kernel
RUN_APPKERNEL="mpirun -n $AKRR_CORES -machinefile $PBS_NODEFILE  {appKerDir}/{executable}"
"""
```

Here, we first generates machine file containing only one node per process and uses this file with mpirun.

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to correct workable batch job script. Here first batch job script is generated with 'akrr_ctl.sh batch_job'. Then this script is executed in interactive session (this improves the turn-around in case of errors). If script fails to execute, the issues can be fixed first in that script itself and then merged to configuration file.

This step is somewhat optional because it is very similar to next step. However the opportunity to work in interactive session improve turn-around time because there is no need to stay in queue for each iteration.

First generate the script to standard output and examine it:

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
```

Sample output of $AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2

```
#!/bin/bash
#SBATCH --partition=debug
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
##SBATCH --time=00:40:00
#SBATCH --time=01:00:00
#SBATCH
--output=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.04.382969
/stdout
#SBATCH
--error=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.04.382969/
stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive

#Common commands
export AKRR_NODES=2
export AKRR_CORES=16
export AKRR_CORES_PER_NODE=8
export AKRR_NETWORK_SCRATCH="/gpfs/scratch/akrr"
export AKRR_LOCAL_SCRATCH="/scratch"
export
AKRR_TASK_WORKDIR="/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43
.04.382969"
export AKRR_APPKER_DIR="/user/akrr/appker/Rush-debug"
export AKRR_AKRR_DIR="/gpfs/scratch/akrr/akrrdata/Rush-debug"
export AKRR_APPKER_NAME="xdmod.benchmark.hpcc"
export AKRR_RESOURCE_NAME="Rush-debug"
export AKRR_TIMESTAMP="2015.03.30.10.43.04.382969"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"
export AKRR_APPKERNEL_INPUT="/user/akrr/appker/Rush-debug/inputs/hpcc/hpccinf.txt.8x2"
export AKRR_APPKERNEL_EXECUTABLE="/user/akrr/appker/Rush-debug/execs/hpcc/hpcc"
source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"
#Populate list of nodes per MPI process
export AKRR_NODELIST=`srun -l --ntasks-per-node=$AKRR_CORES_PER_NODE -n $AKRR_CORES hostname
-s|sort -n| awk '{printf "%s ",$2}' `
export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"
cd "$AKRR_TASK_WORKDIR"
#run common tests
akrrPerformCommonTests
#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"

#create working dir
mkdir workdir
cd workdir
#Copy inputs
cp /user/akrr/appker/Rush-debug/inputs/hpcc/hpccinf.txt.8x2 ./hpccinf.txt
EXE=/user/akrr/appker/Rush-debug/execs/hpcc/hpcc
#Load application enviroment
module load intel
module load intel-mpi
module list
MKLROOT=/user/akrr/appker/Rush-debug/execs/lib/mkl
source $MKLROOT/bin/mklvars.sh intel64
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
ulimit -s unlimited
```

```
#set how to run app kernel
RUN_APPKERNEL="srun /user/akrr/appker/Rush-debug/execs/hpcc/hpcc"

#Generate AppKer signature
appsigcheck.sh $EXE $AKRR_TASK_WORKDIR/.. > $AKRR_APP_STDOUT_FILE

#Execute AppKer
writeToGenInfo "appKerStartTime" "`date`"
$RUN_APPKERNEL >> $AKRR_APP_STDOUT_FILE 2>&1
writeToGenInfo "appKerEndTime" "`date`"

cat hpccoutf.txt  >> $AKRR_APP_STDOUT_FILE 2>&1
cd ..
writeToGenInfo "cpuSpeed" "`grep 'cpu MHz' /proc/cpuinfo`"
#clean-up
if [ "${AKRR_DEBUG=no}" = "no" ]
then
        echo "Deleting input files"
```

```
        rm -rf workdir
fi
writeToGenInfo "endTime" "`date`"
```

Next generate the script on resource:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2
[INFO]: Local copy of batch job script is
/gpfs/user/akrr/akrr/data/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.41.220248/jobfiles/x
dmod.benchmark.hpcc.job
[INFO]: Application kernel working directory on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.41.220248
[INFO]: Batch job script location on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.41.220248/xdmod.be
nchmark.hpcc.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
#get to working directory
cd /gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.41.220248
#check that xdmod.benchmark.hpcc.job is there
ls
#start interactive session
salloc --nodes=2 --ntasks-per-node=8 --time=01:00:00 --exclusive
--constraint="CPU-L5520|CPU-L5630"
#wait till you get access to interactive session

#get to working directory again if you was not redirected there
cd /gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.hpcc/2015.03.30.10.43.41.220248
#run ior application kernel
bash xdmod.benchmark.hpcc.job
```

Examine appstdout file. If it looks ok you can move to the next step

Perform Validation Run

On this step appkernel_validation.py utility is used to validate application kernel installation on the resource. It executes the application kernel and analyses its results. If it fails the problems need to be fixed and another round of validation (as detailed above) should be performed.

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 $RESOURCE $APPKER
```

If at the end you receive "DONE, you can move to next step!" message that means that the hpcc was successfully installed

Schedule regular execution of application kernel.

Now this application kernel can be submitted for regular execution:

```
#Perform a test run on all nodes count
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time between 1:00
and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8 -t0 "01:00" -t1
"05:00" -p 1
```

see `Scheduling and Rescheduling Application Kernels` and `Setup Walltime Limit` for more details

AKRR: Deployment of IMB Applications Kernels on a Resource

Here the deployment of xdmod.benchmark.mpi.imb application kernel is described. This application kernel is based on Intel MPI Benchmark (IOR)

which design to measure the network performance.

For further convenience of application kernel deployment lets define APPKER and RESOURCE environment variable which will contain the HPC resource name:

```
export RESOURCE=rush
export APPKER=xdmod.benchmark.mpi.imb
```


- Installing IMB
  - Building IMB Executables
- Generate Initiate Configuration File
- Edit Configuration File
- Generate Batch Job Script and Execute it Manually (Optional)
- Perform Validation Run
- Schedule regular execution of application kernel.

Installing IMB

In this section the IMB installation process will be described, see also IMB documentation for installation details (https://software.intel.com/en-us/articles/intel-mpi-benchmarks).

Building IMB Executables

First we need to install IMB. Below is a sample listing of commands for IMB installation:

```
#cd to application kernel executable directory
cd $AKRR_APPKER_DIR/execs

#obtain latest version of IMB
wget https://software.intel.com/sites/default/files/managed/d8/cf/IMB_4.0.2.tgz
tar xvzf IMB_4.0.2.tgz

#load MPI compiler
module load intel intel-mpi
source /util/academic/intel/composer_xe_2013/bin/compilervars.sh intel64

cd imb/src
make -f make_ict
```

Generate Initiate Configuration File

Generate Initiate Configuration File:

On AKRR server

```
> python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
[INFO]: Generating application kernel configuration for xdmod.benchmark.mpi.imb on
Rush-debug
[INFO]: Application kernel configuration for xdmod.benchmark.mpi.imb on Rush-debug is
in:
        /user/akrr/akrr/cfg/resources/Rush-debug/xdmod.benchmark.mpi.imb.app.inp.py
```

Edit Configuration File

In contrast to others application kernels IMB uses only one process per node. Therefore during configuration of IMB it is importent to ensure that only one process is run per node.

Below is a listing of configuration file located at $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.mpi.imb.app.inp.py for SLURM:

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.mpi.imb.app.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application enviroment
module load intel-mpi
source /util/academic/intel/composer_xe_2013/bin/compilervars.sh intel64
module list
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

#set how to run mpi applications, one process per node
RUNMPI="srun --ntasks-per-node=1 -n {akrrNNodes}"
"""
```

Here srun with --ntasks-per-node=1 is used to set one process per node execution.

Below is an example for PBS:

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application enviroment
module swap mvapich2 impi/4.1.3.048/intel64
module list

ulimit -s unlimited
#set how to run mpi applications, one process per node
cat $PBS_NODEFILE|uniq >> all_nodes
cat all_nodes
RUNMPI="mpirun -n $AKRR_NODES -machinefile all_nodes"
"""
```

Here, we first generates machine file containing only one node per process and uses this file with mpirun.

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to correct workable batch job script. Here first batch job script is generated with 'akrr_ctl.sh batch_job'. Then this script is executed in interactive session (this improves the turn-around in case of errors). If script fails to execute, the issues can be fixed first in that script itself and then merged to configuration file.

This step is somewhat optional because it is very similar to next step. However the opportunity to work in interactive session improve turn-around time because there is no need to stay in queue for each iteration.

First generate the script to standard output and examine it:

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
```

Sample output of $AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2

Expand source

```
#!/bin/bash
#SBATCH --partition=debug
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
##SBATCH --time=00:30:00
#SBATCH --time=01:00:00
#SBATCH
--output=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.
38.56.259154/stdout
#SBATCH
--error=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.3
8.56.259154/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive

#Common commands
export AKRR_NODES=2
export AKRR_CORES=16
export AKRR_CORES_PER_NODE=8
export AKRR_NETWORK_SCRATCH="/gpfs/scratch/akrr"
export AKRR_LOCAL_SCRATCH="/scratch"
export
AKRR_TASK_WORKDIR="/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015
.03.24.11.38.56.259154"
export AKRR_APPKER_DIR="/user/akrr/appker/Rush-debug"
export AKRR_AKRR_DIR="/gpfs/scratch/akrr/akrrdata/Rush-debug"
export AKRR_APPKER_NAME="xdmod.benchmark.mpi.imb"
export AKRR_RESOURCE_NAME="Rush-debug"
export AKRR_TIMESTAMP="2015.03.24.11.38.56.259154"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"
export AKRR_APPKERNEL_INPUT="/user/akrr/appker/Rush-debug/execs/imb/src/IMB-EXT"
export AKRR_APPKERNEL_EXECUTABLE="/user/akrr/appker/Rush-debug/execs/imb/src/IMB-MPI1"
source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"
#Populate list of nodes per MPI process
export AKRR_NODELIST=`srun -l --ntasks-per-node=$AKRR_CORES_PER_NODE -n $AKRR_CORES
hostname -s|sort -n| awk '{printf "%s ",$2}' `
export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"
cd "$AKRR_TASK_WORKDIR"
#run common tests
akrrPerformCommonTests
#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"

#create working dir
mkdir workdir
cd workdir
export AKRR_APPKER_EXEC_DIR=/user/akrr/appker/Rush-debug/execs/imb/src
#Load application enviroment
module load intel-mpi
source /util/academic/intel/composer_xe_2013/bin/compilervars.sh intel64
module list
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
ulimit -s unlimited
#set how to run mpi applications, one process per node
```

```
RUNMPI="srun --ntasks-per-node=1 -n 2"

#Generate AppKer signature
appsigcheck.sh /user/akrr/appker/Rush-debug/execs/imb/src/IMB-MPI1
$AKRR_TASK_WORKDIR/.. >> $AKRR_APP_STDOUT_FILE
appsigcheck.sh /user/akrr/appker/Rush-debug/execs/imb/src/IMB-EXT
$AKRR_TASK_WORKDIR/.. >> $AKRR_APP_STDOUT_FILE

#Execute AppKer
echo "Checking that running one process per node (for debugging)"
${RUNMPI} hostname
writeToGenInfo "appKerStartTime" "`date`"
${RUNMPI} ${AKRR_APPKER_EXEC_DIR}/IMB-MPI1 -multi 0 -npmin 2 -iter 1000 >>
$AKRR_APP_STDOUT_FILE 2>&1
${RUNMPI} ${AKRR_APPKER_EXEC_DIR}/IMB-EXT  -multi 0 -npmin 2 -iter 1000 >>
$AKRR_APP_STDOUT_FILE 2>&1
writeToGenInfo "appKerEndTime" "`date`"

#clean-up
cd ..
if [ "${AKRR_DEBUG=no}" = "no" ]
then
        echo "Deleting input files"
```

```
        rm -rf workdir
fi
writeToGenInfo "endTime" "`date`"
```

Next generate the script on resource:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2
[INFO]: Local copy of batch job script is
/gpfs/user/akrr/akrr/data/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.40.21.99464
9/jobfiles/xdmod.benchmark.mpi.imb.job
[INFO]: Application kernel working directory on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.40.21.994
649
[INFO]: Batch job script location on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.40.21.994
649/xdmod.benchmark.mpi.imb.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
#get to working directory
cd
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.40.21.994
649
#check that xdmod.benchmark.mpi.imb.job is there
ls
#start interactive session
salloc --nodes=2 --ntasks-per-node=8 --time=01:00:00 --exclusive
--constraint="CPU-E5645"
#wait till you get access to interactive session

#get to working directory again if you was not redirected there
cd
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.benchmark.mpi.imb/2015.03.24.11.40.21.994
649
#run ior application kernel
bash xdmod.benchmark.mpi.imb.job
```

Examine appstdout file, which contains application kernel output:

On remote resource, appstdout content Expand source

```
===ExeBinSignature=== COMPILER: GCC 4.4.x (1 times, 54 bytes)
===ExeBinSignature=== COMPILER: GCC 4.4.3 (1 times, 54 bytes)
===ExeBinSignature=== DYNLIB: Glibc 2.12
===ExeBinSignature=== DYNLIB: GCC Runtime Support Library 4.3.0
===ExeBinSignature=== MD5: 727689a550d3b332fe90a1ff4e88c669
*/gpfs/user/akrr/appker/Rush-debug/execs/imb/src/IMB-MPI1
===ExeBinSignature=== MD5: 5c06eb99fc28fbdba25d5af18217defe
*/util/academic/intel/impi/5.0.2.044/intel64/lib/libmpifort.so.12
===ExeBinSignature=== MD5: 2d996954fb1ee044d21fee9104aeb215
*/util/academic/intel/impi/5.0.2.044/intel64/lib/libmpi.so.12
```

```
===ExeBinSignature=== MD5: 0a45eff701e366f551b431ded93e145e */lib64/libdl.so.2
===ExeBinSignature=== MD5: a93ccf14a901a3e0070a27f2ab9a04c0 */lib64/librt.so.1
===ExeBinSignature=== MD5: 84c3a05eb4a61bf47ae4e93270c8d6a5 */lib64/libpthread.so.0
===ExeBinSignature=== MD5: 9b467352e772717f3c2a9205b05d484d */lib64/libm.so.6
===ExeBinSignature=== MD5: d41adeb80302b25810deefa2eff90bc0 */lib64/libgcc_s.so.1
===ExeBinSignature=== MD5: 87df72ea547ba24ffc062b03af9c26fe */lib64/libc.so.6
===ExeBinSignature=== COMPILER: GCC 4.4.x (1 times, 54 bytes)
===ExeBinSignature=== COMPILER: GCC 4.4.3 (1 times, 54 bytes)
===ExeBinSignature=== DYNLIB: Glibc 2.12
===ExeBinSignature=== DYNLIB: GCC Runtime Support Library 4.3.0
===ExeBinSignature=== MD5: 1c2df3b12449ea9bebc72acd62330aee
*/gpfs/user/akrr/appker/Rush-debug/execs/imb/src/IMB-EXT
===ExeBinSignature=== MD5: 5c06eb99fc28fbdba25d5af18217defe
*/util/academic/intel/impi/5.0.2.044/intel64/lib/libmpifort.so.12
===ExeBinSignature=== MD5: 2d996954fb1ee044d21fee9104aeb215
*/util/academic/intel/impi/5.0.2.044/intel64/lib/libmpi.so.12
===ExeBinSignature=== MD5: 0a45eff701e366f551b431ded93e145e */lib64/libdl.so.2
===ExeBinSignature=== MD5: a93ccf14a901a3e0070a27f2ab9a04c0 */lib64/librt.so.1
===ExeBinSignature=== MD5: 84c3a05eb4a61bf47ae4e93270c8d6a5 */lib64/libpthread.so.0
===ExeBinSignature=== MD5: 9b467352e772717f3c2a9205b05d484d */lib64/libm.so.6
===ExeBinSignature=== MD5: d41adeb80302b25810deefa2eff90bc0 */lib64/libgcc_s.so.1
===ExeBinSignature=== MD5: 87df72ea547ba24ffc062b03af9c26fe */lib64/libc.so.6
#------------------------------------------------------------
#    Intel (R) MPI Benchmarks 4.0 Update 1, MPI-1 part
#------------------------------------------------------------
# Date                  : Tue Mar 24 11:45:27 2015
# Machine               : x86_64
# System                : Linux
# Release               : 2.6.32-431.17.1.el6.x86_64
# Version               : #1 SMP Wed May 7 23:32:49 UTC 2014
# MPI Version           : 3.0
# MPI Thread Environment:
# New default behavior from Version 3.2 on:
# the number of iterations per message size is cut down
# dynamically when a certain run time (per message size sample)
# is expected to be exceeded. Time limit is defined by variable
# "SECS_PER_SAMPLE" (=> IMB_settings.h)
# or through the flag => -time


# Calling sequence was:
# /user/akrr/appker/Rush-debug/execs/imb/src/IMB-MPI1 -multi 0 -npmin 2 -iter 1000
#
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#
# List of Benchmarks to run:
# (Multi-)PingPong
# (Multi-)PingPing
# (Multi-)Sendrecv
# (Multi-)Exchange
# (Multi-)Allreduce
# (Multi-)Reduce
# (Multi-)Reduce_scatter
```

```
# (Multi-)Allgather
# (Multi-)Allgatherv
# (Multi-)Gather
# (Multi-)Gatherv
# (Multi-)Scatter
# (Multi-)Scatterv
# (Multi-)Alltoall
# (Multi-)Alltoallv
# (Multi-)Bcast
# (Multi-)Barrier
#-----------------------------------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#-----------------------------------------------------------------------------
       #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]   Mbytes/sec
            0         1000         2.50         2.50         2.50         0.00
            1         1000         2.51         2.51         2.51         0.38
            2         1000         2.52         2.52         2.52         0.76
            4         1000         2.50         2.50         2.50         1.52
            8         1000         2.51         2.51         2.51         3.04
           16         1000         2.84         2.84         2.84         5.38
           32         1000         2.84         2.84         2.84        10.75
           64         1000         2.79         2.79         2.79        21.85
          128         1000         2.86         2.86         2.86        42.75
          256         1000         3.02         3.02         3.02        80.81
          512         1000         3.29         3.29         3.29       148.35
         1024         1000         3.61         3.61         3.61       270.21
         2048         1000         4.46         4.46         4.46       438.22
         4096         1000         5.22         5.22         5.22       747.82
         8192         1000        10.39        10.39        10.39       751.71
        16384         1000        14.72        14.72        14.72      1061.48
        32768         1000        20.93        20.94        20.93      1492.68
        65536          640        58.15        58.16        58.15      1074.68
       131072          320        92.61        92.63        92.62      1349.50
       262144          160       136.39       136.42       136.41      1832.51
       524288           80       235.87       235.93       235.90      2119.26
      1048576           40       420.95       421.09       421.02      2374.80
      2097152           20       807.90       808.15       808.02      2474.79
      4194304           10      1637.24      1637.90      1637.57      2442.15

...
<many more lines output similar to previous>
...
# All processes entering MPI_Finalize
```

```
```

If it looks ok you can move to the next step

Perform Validation Run

On this step appkernel_validation.py utility is used to validate application kernel installation on the resource. It executes the application kernel and analyses its results. If it fails the problems need to be fixed and another round of validation (as detailed above) should be performed.

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 $RESOURCE $APPKER
```

appkernel_validation.py Sample output

```
DONE, you can move to next step!
```

Schedule regular execution of application kernel.

Now this application kernel can be submitted for regular execution:

```
#Perform a test run on all nodes count
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time
between 1:00 and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8 -t0
"01:00" -t1 "05:00" -p 1
```

see Scheduling and Rescheduling Application Kernels and Setup Walltime Limit for more details

AKRR: Deployment of IOR Applications Kernels on a Resource

Here the deployment of xdmod.benchmark.io.ior application kernel is described. This application kernel is based on IOR benchmark which design to measure the performance of parallel file-systems.

For simplicity lets define APPKER and RESOURCE environment variable which will contain the HPC resource name:

```
export RESOURCE=rush
export APPKER=xdmod.benchmark.io.ior
```


- Installing IOR
  - Installing HDF5
  - Installing Parallel NETCDF (optional)
  - Installing IOR
- Generate Initiate Configuration File
- Edit Configuration File
  - Configuring parameters which defines how IOR will be executed
  - Setting up appKernelRunEnvironmentTemplate

        - Seting up environment
        - Setting up How to run MPI application on All Nodes
        - Setting up How to run MPI application on All Nodes with Nodes Offset
        - Setting up Luster file striping
- Generate Batch Job Script and Execute it Manually (Optional)
- Perform Validation Run
- Schedule regular execution of application kernel.
- FAQ
- During linking stage of compilation got: "undefined reference to `gpfs_fcntl'"

Installing IOR

In this section the IOR installation process will be described, see also IOR benchmark documentation for installation details (http://sourceforge.net/projects/ior-sio/).

Besides POSIX and MPIIO APIs for input-output, IOR can also test parallel HDF5 and NetCDF. You can choose which API to use, based on the API utilization in your center. XDMoD usually tests all of them. HDF5 is a popular format and many scientific HPC application use it. NetCDF is arguably a bit less popular and the kernel results take longer. So if you know (or strongly suspect) that nobody uses NetCDF on your system you might want to skip it.

Installing HDF5

Ideally, it is preferred to use a system-wide installed parallel HDF5 library. This way xdmod.benchmark.io.ior will also test the workability of this particular library installation. HDF5 is a popular format and are often deployed system-wide. Ensure that it was compiled with parallel support (for example by checking presence of h5pcc in $HDF5_HOME/bin).

Bellow is brief notes on parallel hdf5 installation, http://www.hdfgroup.org/HDF5/ for HDF5 installation details.

```
#get to application kernel executable directory
cd $AKRR_APPKER_DIR/execs

#create lib directory if needed and temporary directory for compilation
mkdir -p lib
mkdir -p lib\tmp
cd lib\tmp

#obtain parallel-netcdf source code
wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.14.tar.gz
tar xvzf hdf5-1.8.14.tar.gz
cd hdf5-1.8.14

#configure hdf5
./configure --prefix=$AKRR_APPKER_DIR/execs/lib/hdf5-1.8.14 --enable-parallel
CC=`which mpiicc` CXX=`which mpiicpc`
make

#install
make install
cd $AKRR_APPKER_DIR/execs

#optionally clean-up
rm -rf $AKRR_APPKER_DIR/execs/lib/tmp/hdf5-1.8.14
```

Installing Parallel NETCDF (optional)

IOR can also use parallel NetCDF format to test file system IO. Parallel NetCDF tends to be slower than other APIs and therefore significantly increases the application kernel execution time. Therefore, if you know that parallel NetCDF is used on your system and you want to monitor its performance then go ahead and add it. If you use system-wide installed library, check that it is parallel. Regular serial NetCDF will not work (IOR needs the parallel version). Bellow is brief note on parallel-netcdf installation, refer to http://trac.mcs.anl.gov/projects/parallel-netcdf for more details - note that currently IOR needs the linked version of parallel NetCDF, not the (unfortunately incompatible) NetCDF-4 parallel API.

```
#get to application kernel executable directory
cd $AKRR_APPKER_DIR/execs

#create lib directory if needed and temporary directory for compilation
mkdir -p lib
mkdir -p lib\tmp
cd lib\tmp

#obtain parallel-netcdf source code
wget http://ftp.mcs.anl.gov/pub/parallel-netcdf/parallel-netcdf-1.3.1.tar.gz
tar xvzf parallel-netcdf-1.3.1.tar.gz
cd parallel-netcdf-1.3.1


#check the location of mpi compilers
which mpicc
MPI_TOP_DIR=$(dirname $(dirname `which mpicc`))
#sample output:
#/usr/local/packages/mvapich2/2.0/INTEL-14.0.2/bin/mpicc


#configure parallel-netcdf, specify installation location and which mpi compiler to
use
./configure --prefix=$AKRR_APPKER_DIR/execs/lib/pnetcdf-1.3.1 --with-mpi=$MPI_TOP_DIR

#compile (do not use parallel compilation i.e. -j option)
make

#install
make install
cd $AKRR_APPKER_DIR/execs

#optionally clean-up
rm -rf $AKRR_APPKER_DIR/execs/lib/tmp/parallel-netcdf-1.3.1*
```

Installing IOR

Now we need to install IOR. Below is a sample listing of commands for IOR installation, note that we download a version of the IOR code from our github repository (we made some minor modification to the IOR mainly stdout flushes to prevent race for output from multiple parallel processes). Refer to IOR benchmark documentation for more installation details (http://sourceforge.net/projects/ior-sio/).

```
#cd to application kernel executable directory
cd $AKRR_APPKER_DIR/execs
#remove older version of ior
rm -rf ior


#obtain latest version of IOR from our repository
git clone https://github.com/nsimakov/ior.git
#make configuration scripts
cd ior
./bootstrap
#check that proper compilers are loaded
module list
Currently Loaded Modulefiles:
1) intel/14.0.2 4) tgusage/3.0 7) ant/1.9.4 10) tgresid/2.3.4
2) mvapich2/2.0/INTEL-14.0.2 5) globus/5.0.4-r1 8) java/1.7.0 11) xsede/1.0
3) gx-map/0.5.3.3-r1 6) tginfo/1.1.4 9) uberftp/2.6



#set netcdf and hdf5 enviroment
module load hdf5/1.8.12/INTEL-140-MVAPICH2-2.0
#configure IOR, note the specification of netcdf and hdf include and lib directories
./configure --with-hdf5=yes --with-ncmpi=yes
CPPFLAGS="-I$AKRR_APPKER_DIR/execs/lib/pnetcdf-1.3.1/include
-I/usr/local/packages/netcdf/4.2.1.1/INTEL-140-MVAPICH2-2.0/include"
LDFLAGS="-L/usr/local/packages/hdf5/1.8.12/INTEL-140-MVAPICH2-2.0/lib
-L$AKRR_APPKER_DIR/execs/lib/pnetcdf-1.3.1/lib"
#compile
make

#the binary should be src/ior
ls $AKRR_APPKER_DIR/execs/ior/src/ior
```

Generate Initiate Configuration File

Generate Initiate Configuration File:

On AKRR server

```
> python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
[INFO]: Generating application kernel configuration for xdmod.benchmark.io.ior on rush
[INFO]: Application kernel configuration for xdmod.benchmark.io.ior on SuperMIC is in:


/home/mikola/wsp/test/akrr/cfg/resources/rush/xdmod.benchmark.io.ior.app.inp.py
```

Edit Configuration File

Configuring IOR is more complex than the NAMD based application kernel. The main issue with IOR is to try and bypass memory caching on

compute and storage nodes (which will inflate the performance numbers, as it is doing i/o to memory rather than the actual storage system).

To avoid caching on multi-node tests (two nodes and larger) we use MPI processes reordering, for example if the file was written from node A and node B, after that node A will read what node B wrote and node B will read what node A wrote.

For singe node runs, the test is actually performed on 2 nodes where one node writes and another node reads. This way we (hopefully) obtain single node performance metrics without hitting local caches.

We have little influence on the storage node configuration/caches so the strategy here is to do all writes first (for all tests) and then do reads, the hope is that writing large files will overwrite the cache on storage node and the following reads will be done with minimal influence from storage nodes cache.

Configuring the IOR application kernel is essentially setting up the IOR execution in a way reflecting the above strategy for bypassing cache.

The most effective strategy to properly setup `xdmod.benchmark.io.ior` is to use an interactive session to test and define configurable parameters.

Configuring parameters which defines how IOR will be executed

First, lets configure parameters which generally do not require an interactive debug session.

Below is a listing of the generated default configuration file located at $AKRR_HOME/cfg/resources/$RESOURCE/`xdmod.benchmark.io.ior`

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py

```
#which IO API/formats to check
testPOSIX=True
testMPIIO=True
testHDF5=True
testNetCDF=True

#will do write test first and after that read, that minimize the caching impact from
storage nodes
#require large temporary storage easily 100s GiB
doAllWritesFirst=True

appKernelRunEnvironmentTemplate="""
#load application environment
module load hdf5/1.8.12/INTEL-140-MVAPICH2-2.0
module list

#set executable location
EXE=$AKRR_APPKER_DIR/execs/ior/src/ior

#set how to run mpirun on all nodes
for node in $AKRR_NODELIST; do echo $node>>all_nodes; done
RUNMPI="mpiexec -n $AKRR_CORES -f all_nodes"

#set how to run mpirun on all nodes with offset, first print all nodes after node 1
and then node 1
sed -n "$(($AKRR_CORES_PER_NODE+1)),$(($AKRR_CORES))p" all_nodes > all_nodes_offset
sed -n "1,$(($AKRR_CORES_PER_NODE))p" all_nodes >> all_nodes_offset
RUNMPI_OFFSET="mpiexec -n $AKRR_CORES -f all_nodes_offset"

#set striping for lustre file system
RESOURCE_SPECIFIC_OPTION_N_to_1="-O lustreStripeCount=$AKRR_NODES"
RESOURCE_SPECIFIC_OPTION_N_to_N=""
#other resource specific options
RESOURCE_SPECIFIC_OPTION=""
"""
```

The first several lines specify which IO APIs to test:

Fragment of $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py

```
#which IO API/formats to check
testPOSIX=True
testMPIIO=True
testHDF5=True
testNetCDF=True
```

The one which you may want to set to False is testNetCDF (see discussion above during parallel NetCDF library installation).

Next several lines instruct IOR application kernel to do all writes first and then do all reads:

Fragment of $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py

```
#will do write test first and after that read, that minimize the caching impact from
storage nodes
#require large temporary storage easily 100s GiB
doAllWritesFirst=True
```

The only reason not to do writes first is if you have limited storage size, as the size of all generated output files are quite substantial (after testing is done all generated files are removed, but you mush have sufficient space to hold the interim results). For example on 8 nodes of machine with 12 cores per node and all tests done the total size of writes will be 10 tests * 200 MB per core * 12 cores per node * 8 node =192 GB.

Setting up *appKernelRunEnvironmentTemplate*

Now we need to set *appKernelRunEnvironmentTemplate* template variable.

We will do it section by section and will use interactive session on resource to test the entries.

First lets generate **test** application kernel batch job script (**not IOR script**, we will use this test job script to set AKRR predifined environment variable to use during entries validation):

On AKRR Server

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a test -n 2
[INFO]: Local copy of batch job script is
/home/mikola/wsp/test/akrr/data/SuperMIC/test/2014.12.16.12.12.51.198128/jobfiles/test
.job

[INFO]: Application kernel working directory on SuperMIC is
/work/xdtas/akrrdata/test/2014.12.16.12.12.51.198128
[INFO]: Batch job script location on SuperMIC is
/work/xdtas/akrrdata/test/2014.12.16.12.12.51.198128/test.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
#get to working directory
cd /work/xdtas/akrrdata/test/2014.12.16.12.12.51.198128
#check that test.job is there
ls
#start interactive session
qsub -I -l walltime=01:00:00,nodes=2:ppn=20
#wait till you get access to interactive session

#get to working directory again if not already there
cd /work/xdtas/akrrdata/test/2014.12.16.12.12.51.198128
#load everything from test.job
source test.job
#check AKRR predifined environment variable are loaded
echo $AKRR_NODES
#output should be 2

echo $AKRR_NODELIST
#output should be space separated list of hosts
```

Now we ready to configure *appKernelRunEnvironmentTemplate (in $AKRR_HOME/cfg/resources/Rush-debug/xdmod.benchmark.io.ior.app.inp.py)*.

Seting up environment

In first section we set proper environment for IOR to work, we also place IOR executable location to EXE variable (binary in $EXE will be used to generate application signature):

appKernelRunEnvironmentTemplate fragment of xdmod.benchmark.io.ior.inp.py

```
#load application environment
module load hdf5/1.8.12/INTEL-140-MVAPICH2-2.0
module list

#set executable location
EXE=$AKRR_APPKER_DIR/execs/ior/src/ior
```

Make the appropriate changes for your system in $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py and execute in interactive session, check that IOR is working:

On remote resource

```
#load application environment
module load hdf5/1.8.12/INTEL-140-MVAPICH2-2.0
module list

#set executable location
EXE=$AKRR_APPKER_DIR/execs/ior/src/ior
```

Check that IOR is working:

On remote resource

```
#let check is it working:
mpirun $EXE
```

The default file sizes for ior are quite small, so running ior with no arguments as above should return very quickly - unless something is wrong.

If it is not working modify the environment appropriately in $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py.

Setting up How to run MPI application on All Nodes

Next, we setup how to run mpirun/mpiexec on all nodes:

appKernelRunEnvironmentTemplate fragment of xdmod.benchmark.io.ior.inp.py

```
#set how to run mpirun on all nodes
for node in $AKRR_NODELIST; do echo $node>>all_nodes; done
RUNMPI="mpiexec -n $AKRR_CORES -f all_nodes"
```

Nearly all mpi implementations accept a plain list of hosts(nodes) as a machines file on which to run the MPI tasks, although the options to use that list may vary. In this script we generate a list of all nodes (one per MPI process) and place in the $RUNMPI environment variable how to execute mpirun (or whatever MPI task launcher is preferred on your platform).

Adjust this section in $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py and again execute in an interactive session, to check that IOR is working:

On remote resource

```
#set how to run mpirun on all nodes
for node in $AKRR_NODELIST; do echo $node>>all_nodes; done
RUNMPI="mpiexec -n $AKRR_CORES -f all_nodes"
```

Note that you **must** supply number of processes to your mpi launcher, some test do not use all processes. Therefore, without explicit specification of processes numbers the tests will be incorrect (for single node metrics).

Single-node performance is obtain from two node run where one will do writes and another reads.

Check that IOR is working:

On remote resource

```
#let check is it working:
$RUNMPI $EXE -vv
```

"-vv" option will make IOR more verbose and shows the processes assignment to nodes:

Sample of $RUNMPI $EXE -vv output Expand source

```
IOR-3.0.1: MPI Coordinated Test of Parallel I/O
Began: Tue Dec 16 11:34:22 2014
Command line used: /home/xdtas/appker/supermic/execs/ior/src/ior -vv
Machine: Linux smic006 2.6.32-358.23.2.el6.x86_64 #1 SMP Sat Sep 14 05:32:37 EDT 2013
x86_64
Using synchronized MPI timer
Start time skew across all tasks: 0.00 sec
Test 0 started: Tue Dec 16 11:34:22 2014
Path: /worka/work/xdtas/akrrdata/test/2014.12.16.12.12.51.198128
FS: 698.3 TiB   Used FS: 0.5%   Inodes: 699.1 Mi   Used Inodes: 0.7%
Participating tasks: 40
task 0 on smic006
task 1 on smic006
<many lines like previous>
task 9 on smic006
Summary:
        api                = POSIX
        test filename      = testFile
        access             = single-shared-file
        pattern            = segmented (1 segment)
        ordering in a file = sequential offsets
        ordering inter file= no tasks offsets
        clients            = 40 (20 per node)
        repetitions        = 1
        xfersize           = 262144 bytes
        blocksize          = 1 MiB
        aggregate filesize = 40 MiB
Using Time Stamp 1418751262 (0x54906d1e) for Data Signature
access    bw(MiB/s)  block(KiB) xfer(KiB)  open(s)    wr/rd(s)   close(s)   total(s)
iter
------    ---------  ---------- ---------  --------   --------   --------   --------
----
Commencing write performance test: Tue Dec 16 11:34:22 2014
write     26.05      1024.00    256.00     0.004612   1.52       1.48       1.54
0
Commencing read performance test: Tue Dec 16 11:34:24 2014
read      1149.09    1024.00    256.00     0.002482   0.034575   0.033708   0.034810
0
remove    -          -          -          -          -          -          0.002008
0
Max Write: 26.05 MiB/sec (27.31 MB/sec)
Max Read:  1149.09 MiB/sec (1204.91 MB/sec)
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev    Mean(s) Test# #Tasks tPN reps
fPP reord reordoff reordrand seed segcnt blksiz xsize aggsize API RefNum
write          26.05      26.05      26.05       0.00    1.53557 0 40 20 1 0 0 1 0 0 1
1048576 262144 41943040 POSIX 0
read         1149.09    1149.09    1149.09       0.00    0.03481 0 40 20 1 0 0 1 0 0 1
1048576 262144 41943040 POSIX 0
Finished: Tue Dec 16 11:34:24 2014
```

If it is not working modify the executed commands and copy the good ones to $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py.

Setting up How to run MPI application on All Nodes with Nodes Offset

Next, we setup how to run mpirun/mpiexec on all nodes with one node offset:

appKernelRunEnvironmentTemplate fragment of xdmod.benchmark.io.ior.inp.py

```
#set how to run mpirun on all nodes with offset, first print all nodes after node 1
and then node 1
sed -n "$(($AKRR_CORES_PER_NODE+1)),$(($AKRR_CORES))p" all_nodes > all_nodes_offset
sed -n "1,$(($AKRR_CORES_PER_NODE))p" all_nodes >> all_nodes_offset
RUNMPI_OFFSET="mpiexec -n $AKRR_CORES -f all_nodes_offset"
```

Nearly all mpi flavours accept plain list of hosts(nodes) as a machines file, some of them uses different option to load that list. In this script we generate a list of all nodes (one per MPI process) and place to RUNMPI variable how to execute mpirun (or whatever luncher is used on your platform).

Adjust this section in $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py and execute in interactive session, check that IOR is working:

On remote resource

```
#set how to run mpirun on all nodes with offset, first print all nodes after node 1
and then node 1
sed -n "$(($AKRR_CORES_PER_NODE+1)),$(($AKRR_CORES))p" all_nodes > all_nodes_offset
sed -n "1,$(($AKRR_CORES_PER_NODE))p" all_nodes >> all_nodes_offset
echo "all_nodes_offset:"
cat all_nodes_offset
RUNMPI_OFFSET="mpiexec -n $AKRR_CORES -f all_nodes_offset"
```

Check that IOR is working:

On remote resource

```
#let check is it working:
$RUNMPI_OFFSET $EXE -vv
```

If it is not working modify the executed commands and copy the good ones to $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.benchmark.io.ior.inp.py.

Setting up Luster file striping

If you use Lustre you might want to use file striping for better parallel performance. The variables RESOURCE_SPECIFIC_OPTION, RESOURCE_SPECIFIC_OPTION_N_to_1 and RESOURCE_SPECIFIC_OPTION_N_to_N and will be passed to IOR as command line options.

RESOURCE_SPECIFIC_OPTION will be passed to all IOR execution.

RESOURCE_SPECIFIC_OPTION_N_to_1 will be passed to IOR for tests there all processes writes to a single file and RESOURCE_SPECIFIC_OPTION_N_to_N will be passed to IOR for tests there all processes writes to their own independent files.

Bellow is a fragment example from xdmod.benchmark.io.ior.inp.py which will instruct IOR to use striping equal to the number of nodes when all processes write to a single file.

appKernelRunEnvironmentTemplate fragment of xdmod.benchmark.io.ior.inp.py

```
#set striping for lustre file system
RESOURCE_SPECIFIC_OPTION_N_to_1="-O lustreStripeCount=$AKRR_NODES"
RESOURCE_SPECIFIC_OPTION_N_to_N=""
#other resource specific options
RESOURCE_SPECIFIC_OPTION=""
```

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to correct workable batch job script. Here first batch job script is generated with 'akrr_ctl.sh batch_job'. Then this script is executed in interactive session (this improves the turn-around in case of errors). If script fails to execute, the issues can be fixed first in that script itself and then merged to configuration file.

This step is somewhat optional because it is very similar to next step. However the opportunity to work in interactive session improve turn-around time because there is no need to stay in queue for each iteration.

First generate the script to standard output and examine it:

```
$AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
```

Sample output of $AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2

Expand source

```
[INFO]: Below is content of generated batch job script:
#!/bin/bash
#PBS -l nodes=2:ppn=20
#PBS -m n
#PBS -q workq
#PBS -e stderr
#PBS -o stdout
#PBS -l walltime=03:00:00
#PBS -u xdtas
#PBS -A TG-CCR120014


#Populate list of nodes per MPI process
export AKRR_NODELIST=`cat $PBS_NODEFILE`


#Common commands
export AKRR_NODES=2
export AKRR_CORES=40
export AKRR_CORES_PER_NODE=20
export AKRR_NETWORK_SCRATCH="/work/xdtas/scratch"
export AKRR_LOCAL_SCRATCH="/tmp"
export
AKRR_TASK_WORKDIR="/work/xdtas/akrrdata/xdmod.benchmark.io.ior/2014.12.16.13.39.55.613
578"
export AKRR_APPKER_DIR="/home/xdtas/appker/supermic"
export AKRR_AKRR_DIR="/work/xdtas/akrrdata"
```

```
export AKRR_APPKER_NAME="xdmod.benchmark.io.ior"
export AKRR_RESOURCE_NAME="SuperMIC"
export AKRR_TIMESTAMP="2014.12.16.13.39.55.613578"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"

export AKRR_APPKERNEL_INPUT="/home/xdtas/appker/supermic/inputs"
export AKRR_APPKERNEL_EXECUTABLE="/home/xdtas/appker/supermic/execs/ior/C/IOR"

source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"

export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"

cd "$AKRR_TASK_WORKDIR"

#run common tests
akrrPerformCommonTests

#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"



# MPI IO hints (optional)
# MPI IO hints are environment variables in the following format:
#
# 'IOR_HINT__<layer>__<hint>=<value>', where <layer> is either 'MPI'
# or 'GPFS', <hint> is the full name of the hint to be set, and <value>
# is the hint value.  E.g., 'export IOR_HINT__MPI__IBM_largeblock_io=true'
# 'export IOR_HINT__GPFS__hint=value' in mpi_io_hints


#create working dir
export AKRR_TMP_WORKDIR=`mktemp -d /work/xdtas/scratch/ior.XXXXXXXXX`
echo "Temporary working directory: $AKRR_TMP_WORKDIR"
cd $AKRR_TMP_WORKDIR



#load application environment
module load hdf5/1.8.12/INTEL-140-MVAPICH2-2.0
module list

#set executable location
EXE=/home/xdtas/appker/supermic/execs/ior/src/ior

#set how to run mpirun on all nodes
for node in $AKRR_NODELIST; do echo $node>>all_nodes; done
echo "all_nodes:"
cat all_nodes
RUNMPI="mpiexec -n $AKRR_CORES -f all_nodes"

#set how to run mpirun on all nodes with offset, first print all nodes after node 1
and then node 1
sed -n "$(($AKRR_CORES_PER_NODE+1)),$(($AKRR_CORES))p" all_nodes > all_nodes_offset
sed -n "1,$(($AKRR_CORES_PER_NODE))p" all_nodes >> all_nodes_offset
echo "all_nodes_offset:"
cat all_nodes_offset
```

```
RUNMPI_OFFSET="mpiexec -n $AKRR_CORES -f all_nodes_offset"

#set how to run mpirun on first node
sed -n "1,$(($AKRR_CORES_PER_NODE))p" all_nodes > first_node
echo "first_node:"
cat first_node
RUNMPI_FIRST_NODE="mpiexec -n $AKRR_CORES_PER_NODE -f first_node"

#set how to run mpirun on second node
sed -n "$(($AKRR_CORES_PER_NODE+1)),$((2*$AKRR_CORES_PER_NODE))p" all_nodes >
second_node
echo "second_node:"
cat second_node
RUNMPI_SECOND_NODE="mpiexec -n $AKRR_CORES_PER_NODE -f second_node"

#set striping for lustre file system
RESOURCE_SPECIFIC_OPTION_N_to_1="-O lustreStripeCount=$AKRR_NODES"
RESOURCE_SPECIFIC_OPTION_N_to_N=""

#other resource specific options
RESOURCE_SPECIFIC_OPTION=""


#Generate AppKer signature
appsigcheck.sh $EXE $AKRR_TASK_WORKDIR/.. > $AKRR_APP_STDOUT_FILE



#blockSize and transferSize
COMMON_TEST_PARAM="-b 200m -t 20m"
#2 level of verbosity, don't clear memory
COMMON_OPTIONS="-vv"
CACHING_BYPASS="-Z"

#list of test to perform
TESTS_LIST=("-a POSIX $RESOURCE_SPECIFIC_OPTION_N_to_1"
"-a POSIX -F $RESOURCE_SPECIFIC_OPTION_N_to_N"
"-a MPIIO $RESOURCE_SPECIFIC_OPTION_N_to_1"
"-a MPIIO -c $RESOURCE_SPECIFIC_OPTION_N_to_1"
"-a MPIIO -F $RESOURCE_SPECIFIC_OPTION_N_to_N"
"-a HDF5 $RESOURCE_SPECIFIC_OPTION_N_to_1"
"-a HDF5 -c $RESOURCE_SPECIFIC_OPTION_N_to_1"
"-a HDF5 -F $RESOURCE_SPECIFIC_OPTION_N_to_N")

#combine common parameters
COMMON_PARAM="$COMMON_OPTIONS $RESOURCE_SPECIFIC_OPTION $CACHING_BYPASS
$COMMON_TEST_PARAM"


echo "Using $AKRR_TMP_WORKDIR for test...." >> $AKRR_APP_STDOUT_FILE 2>&1

#determine filesystem for file
canonicalFilename=`readlink -f $AKRR_TMP_WORKDIR`
filesystem=`awk -v canonical_path="$canonicalFilename" '{if ($2!="/" &&
1==index(canonical_path, $2)) print $3 " " $1 " " $2;}' /proc/self/mounts`
echo "File System To Test: $filesystem" >> $AKRR_APP_STDOUT_FILE 2>&1
writeToGenInfo "fileSystem" "$filesystem"

#start the tests
```

```
writeToGenInfo "appKerStartTime" "`date`"

#do write first
for TEST_PARAM in "${TESTS_LIST[@]}"
do
    echo "# Starting Test: $TEST_PARAM" >> $AKRR_APP_STDOUT_FILE 2>&1
    fileName=`echo ior_test_file_$TEST_PARAM |tr  '-' '_'|tr  ' ' '_'|tr  '=' '_'`

    #run the test
    command_to_run="$RUNMPI $EXE $COMMON_PARAM $TEST_PARAM -w -k -o
$AKRR_TMP_WORKDIR/$fileName"
    echo "executing: $command_to_run" >> $AKRR_APP_STDOUT_FILE 2>&1
    $command_to_run >> $AKRR_APP_STDOUT_FILE 2>&1
done
#do read last
for TEST_PARAM in "${TESTS_LIST[@]}"
do
    echo "# Starting Test: $TEST_PARAM" >> $AKRR_APP_STDOUT_FILE 2>&1
    fileName=`echo ior_test_file_$TEST_PARAM |tr  '-' '_'|tr  ' ' '_'|tr  '=' '_'`

    #run the test
    command_to_run="$RUNMPI_OFFSET $EXE $COMMON_PARAM $TEST_PARAM -r -o
$AKRR_TMP_WORKDIR/$fileName"
    echo "executing: $command_to_run" >> $AKRR_APP_STDOUT_FILE 2>&1
    $command_to_run >> $AKRR_APP_STDOUT_FILE 2>&1
done

writeToGenInfo "appKerEndTime" "`date`"




#clean-up
cd $AKRR_TASK_WORKDIR
if [ "${AKRR_DEBUG=no}" = "no" ]
then
        echo "Deleting temporary files"
        rm -rf $AKRR_TMP_WORKDIR
else
        echo "Copying temporary files"
        cp -r $AKRR_TMP_WORKDIR workdir
        rm -rf $AKRR_TMP_WORKDIR
fi



writeToGenInfo "endTime" "`date`"
```

```
[INFO]: Removing generated files from file-system as only batch job script printing
was requested
```

Next generate the script on resource:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2
[INFO]: Local copy of batch job script is
/home/mikola/wsp/test/akrr/data/SuperMIC/xdmod.benchmark.io.ior/2014.12.16.13.43.05.85
1076/jobfiles/xdmod.benchmark.io.ior.job

[INFO]: Application kernel working directory on SuperMIC is
/work/xdtas/akrrdata/xdmod.benchmark.io.ior/2014.12.16.13.43.05.851076
[INFO]: Batch job script location on SuperMIC is
/work/xdtas/akrrdata/xdmod.benchmark.io.ior/2014.12.16.13.43.05.851076/xdmod.benchmark
.io.ior.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
#get to working directory
cd /work/xdtas/akrrdata/xdmod.benchmark.io.ior/2014.12.16.13.43.05.851076
#check that xdmod.benchmark.io.ior.job is there
ls
#start interactive session
qsub -I -l walltime=01:00:00,nodes=2:ppn=20
#wait till you get access to interactive session

#get to working directory again if you was not redirected there
cd /work/xdtas/akrrdata/xdmod.benchmark.io.ior/2014.12.16.13.43.05.851076
#run ior application kernel
bash xdmod.benchmark.io.ior.job
```

Examine appstdout file, which contains application kernel output:

On remote resource, appstdout content Expand source

```
===ExeBinSignature=== DYNLIB: Glibc 2.12
<many lines like previous>
===ExeBinSignature=== MD5: a6e6262120f548dc3a4486ac3df13863
*/home/xdtas/appker/supermic/execs/ior/src/ior
<many lines like previous>
Using /work/xdtas/scratch/ior.xnUWSQf0V for test....
File System To Test: lustre 172.17.40.34@o2ib:172.17.40.35@o2ib:/smic /worka
# Starting Test: -a POSIX -O lustreStripeCount=2
executing: mpiexec -n 40 -f all_nodes /home/xdtas/appker/supermic/execs/ior/src/ior
-vv  -Z -b 200m -t 20m -a POSIX -O lustreStripeCount=2 -w -k -o
/work/xdtas/scratch/ior.xnUWSQf0V/ior_test_file__a_POSIX__O_lustreStripeCount_2
IOR-3.0.1: MPI Coordinated Test of Parallel I/O

Began: Tue Dec 16 12:44:13 2014
Command line used: /home/xdtas/appker/supermic/execs/ior/src/ior -vv -Z -b 200m -t 20m
```

```
-a POSIX -O lustreStripeCount=2 -w -k -o
/work/xdtas/scratch/ior.xnUWSQf0V/ior_test_file__a_POSIX__O_lustreStripeCount_2
Machine: Linux smic001 2.6.32-358.23.2.el6.x86_64 #1 SMP Sat Sep 14 05:32:37 EDT 2013
x86_64
Using synchronized MPI timer
Start time skew across all tasks: 0.00 sec

Test 0 started: Tue Dec 16 12:44:13 2014
Path: /worka/work/xdtas/scratch/ior.xnUWSQf0V
FS: 698.3 TiB   Used FS: 0.5%   Inodes: 699.1 Mi   Used Inodes: 0.7%
Participating tasks: 40
task 0 on smic001
task 1 on smic001
<many lines like previous>
task 9 on smic001
Summary:
        api                = POSIX
        test filename      =
/work/xdtas/scratch/ior.xnUWSQf0V/ior_test_file__a_POSIX__O_lustreStripeCount_2
        access             = single-shared-file
        pattern            = segmented (1 segment)
        ordering in a file = sequential offsets
        ordering inter file= random task offsets >= 1, seed=0
        clients            = 40 (20 per node)
        repetitions        = 1
        xfersize           = 20 MiB
        blocksize          = 200 MiB
        aggregate filesize = 7.81 GiB
        Lustre stripe size = Use default
              stripe count = 2
Using Time Stamp 1418755453 (0x54907d7d) for Data Signature

access    bw(MiB/s)  block(KiB) xfer(KiB)  open(s)    wr/rd(s)   close(s)   total(s)
iter
------    ---------  ---------- ---------  --------   --------   --------   --------
----
Commencing write performance test: Tue Dec 16 12:44:13 2014
write     508.93     204800     20480      0.005252   15.72      11.85      15.72
0

Max Write: 508.93 MiB/sec (533.65 MB/sec)

Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev    Mean(s) Test# #Tasks tPN reps
fPP reord reordoff reordrand seed segcnt blksiz xsize aggsize API RefNum
write         508.93     508.93     508.93       0.00   15.71921 0 40 20 1 0 0 1 1 0 1
209715200 20971520 8388608000 POSIX 0

Finished: Tue Dec 16 12:44:28 2014
# Starting Test: -a POSIX -F

...
<many output similar to previous>
...
```

```
Finished: Tue Dec 16 12:47:42 2014
```

If it looks ok you can move to the next step

Perform Validation Run

On this step appkernel_validation.py utility is used to validate application kernel installation on the resource. It executes the application kernel and analyses its results. If it fails the problems need to be fixed and another round of validation (as detailed above) should be performed.

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py -n 2 $RESOURCE $APPKER
```

appkernel_validation.py Sample output

```
...

DONE, you can move to next step!
```

Schedule regular execution of application kernel.

Now this application kernel can be submitted for regular execution:

```
#Perform a test run on all nodes count
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time
between 1:00 and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8 -t0
"01:00" -t1 "05:00" -p 1
```

see Scheduling and Rescheduling Application Kernels and Setup Walltime Limit for more details

FAQ

During linking stage of compilation got: "undefined reference to `gpfs_fcntl'"

The linker does not by default link the necessary GPFS library, you can instead do this step manually, for example:

```
cd src
mpiicc  -g -O2   -o ior -lgpfs ior.o utilities.o parse_options.o aiori-POSIX.o aiori-MPIIO.o
```

Or rerun configuration with corrected LIBS (correct configure option for your needs), for example:

```
./configure --with-hdf5=no --with-ncmpi=no  LIBS="-lgpsf"
```

AKRR: Deployment of NAMD Applications Kernels on a Resource

Here the deployment of xdmod.app.md.namd application kernel is described. This application kernel is based on NAMD application.

For simplicity lets define APPKER and RESOURCE enviroment varible which will contain the HPC resource name:

```
export RESOURCE=rush
export APPKER=xdmod.app.md.namd
```

Install Application or Identify where it is Installed

NAMD is very often installed system-wide. One of the purposes of application kernels is to monitor the performance of application which is used by regular users. If NAMD is not installed then you need to install it or choose not to use it.

Majority of HPC resources utilize some kind of module system. Execute it and see is NAMD already installed

On resource

```
> module avail
...
namd/2.10b1-IBVERBS       namd/2.8-IBVERBS(default) namd/2.8-MPI-CUDA
namd/2.9-MPI              namd/2.9b2-TCP
namd/2.10b1-IBVERBS-smp   namd/2.8-IBVERBS-SRC      namd/2.8-TCP
namd/2.9-MPI-CUDA         namd/2.9b3-IBVERBS
namd/2.7b1-mx             namd/2.8-IBVERBS-smp      namd/2.9-IBVERBS
namd/2.9b2
namd/2.7b2-IB             namd/2.8-MC               namd/2.9-IBVERBS-SRC
namd/2.9b2-IBVERBS
namd/2.8-CUDA             namd/2.8-MPI              namd/2.9-IBVERBS-SRC-DBG
namd/2.9b2-IBVERBS-smp

...
```

write down the name of module you want to use.

Generate Initiate Configuration File

Generate Initiate Configuration File:

On AKRR server

```
> python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
[INFO]: Generating application kernel configuration for xdmod.app.md.namd on rush
[INFO]: Application kernel configuration for xdmod.app.md.namd on rush is in:
        /home/mikola/wsp/test/akrr/cfg/resources/rush/xdmod.app.md.namd.app.inp.py
```

Edit Configuration File

Below is listing of generated configuration file located at $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.app.md.namd.inp.py

(jtp: the config file I generated was named xdmod.app.md.namd.app.inp.py)

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.app.md.namd.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application environment
module load namd
export CONV_RSH=ssh
#set executable location
EXE=`which namd2`
charmrun_bin=`which charmrun`

#prepare nodelist for charmmrun
for n in $AKRR_NODELIST; do echo host $n>>nodelist; done

#set how to run app kernel
RUN_APPKERNEL="$charmrun_bin +p$AKRR_CORES ++nodelist nodelist $EXE ./input.namd"
"""
```

It contain only one parameter *appKernelRunEnvironmentTemplate* which need to be edited:

1) First part is "Load application environment", here you need to set proper enviroment. For example:

```
#Load application environment
module load namd/2.9-IBVERBS

export CONV_RSH=ssh
```

The last line above specify to use ssh for application launching.

2) Second part is "set executable location", it set the location of executables absolute path to namd2 should be placed to EXE variable (application signature will be calculated for that executable). For example:

```
#set executable location
EXE=`which namd2`
charmrun_bin=`which namd2`
```

3)Third part is "prepare nodelist for charmmrun", it setup nodelist file which later will be used by charmm run

```
#prepare nodelist for charmmrun
for n in $AKRR_NODELIST; do echo host $n>>nodelist; done
```

4)Fourth part is "set how to run app kernel", it set RUN_APPKERNEL, which specify how to execute namd:

```
#set how to run app kernel
RUN_APPKERNEL="$charmrun_bin +p$AKRR_CORES ++nodelist nodelist $EXE ./input.namd"
```

If MPI version of namd is used the AKRR_HOME/resource/$RESOURCE/xdmod.app.md.namd.inp.py can look like:

$AKRR_HOME/resource/$RESOURCE/xdmod.app.md.namd.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application environment
module load namd/2.9
module list

#set executable location
EXE=`which namd2`

#set how to run app kernel
RUN_APPKERNEL="mpirun -n $AKRR_CORES -hostfile $PBS_NODEFILE $EXE ./input.namd"
"""
```

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to correct workable batch job script. Here first batch job script is generated with 'akrr_ctl.sh batch_job'. Then this script is executed in interactive session (this improves the turn-around in case of errors). If script fails to execute, the issues can be fixed first in that script itself and then merged to configuration file.

This step is somewhat optional because it is very similar to next step. However the opportunity to work in interactive session improve turn-around time because there is no need to stay in queue for each iteration.

First generate the script to standard output and examine it:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
[INFO]: Below is content of generated batch job script:
#!/bin/bash
#SBATCH --partition=general-compute
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --time=00:13:00
#SBATCH
--output=/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.11.23.05.47
.655898/stdout
#SBATCH
--error=/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.11.23.05.47.
655898/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive


#Populate list of nodes per MPI process
_TASKS_PER_NODE=`echo $SLURM_TASKS_PER_NODE|sed "s/(x[0-9]*)//g"`
export AKRR_NODELIST=`scontrol show hostname $SLURM_NODELIST| awk "{for
(i=0;i<$_TASKS_PER_NODE;++i)print}"`


#Common commands
export AKRR_NODES=2
export AKRR_CORES=16
export AKRR_CORES_PER_NODE=8
export AKRR_NETWORK_SCRATCH="/panasas/scratch/nikolays"
export AKRR_LOCAL_SCRATCH="/tmp"
export
```

```
AKRR_TASK_WORKDIR="/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.1
1.23.05.47.655898"
export AKRR_APPKER_DIR="/user/nikolays/appker/rush"
export AKRR_AKRR_DIR="/panasas/scratch/nikolays/akrrdata/rush"

export AKRR_APPKER_NAME="xdmod.app.md.namd"
export AKRR_RESOURCE_NAME="rush"
export AKRR_TIMESTAMP="2014.12.11.23.05.47.655898"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"

export AKRR_APPKERNEL_INPUT="/user/nikolays/appker/rush/inputs/namd/apoa1_nve"
export AKRR_APPKERNEL_EXECUTABLE="/user/nikolays/appker/rush/execs"

source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"

export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"

cd "$AKRR_TASK_WORKDIR"

#run common tests
akrrPerformCommonTests

#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"


#create working dir
export AKRR_TMP_WORKDIR=`mktemp -d /panasas/scratch/nikolays/namd.XXXXXXXXX`
echo "Temporary working directory: $AKRR_TMP_WORKDIR"
cd $AKRR_TMP_WORKDIR

#Copy inputs
cp /user/nikolays/appker/rush/inputs/namd/apoa1_nve/* ./



#Load application environment
module load namd/2.9-IBVERBS
export CONV_RSH=ssh

#set executable location
EXE=`which namd2`
charmrun_bin=`which charmrun`

#prepare nodelist for charmmrun
export CONV_RSH=ssh
for n in $AKRR_NODELIST; do echo host $n>>nodelist; done

#set how to run app kernel
RUN_APPKERNEL="$charmrun_bin  +p$AKRR_CORES ++nodelist nodelist $EXE ./input.namd"


#Generate AppKer signature
appsigcheck.sh $EXE $AKRR_TASK_WORKDIR/.. > $AKRR_APP_STDOUT_FILE


#Execute AppKer
$RUN_APPKERNEL >> $AKRR_APP_STDOUT_FILE 2>&1
```

```
#clean-up
cd $AKRR_TASK_WORKDIR
if [ "${AKRR_DEBUG=no}" = "no" ]
then
        echo "Deleting temporary files"
        rm -rf $AKRR_TMP_WORKDIR
else
        echo "Copying temporary files"
        cp -r $AKRR_TMP_WORKDIR workdir
        rm -rf $AKRR_TMP_WORKDIR
fi



writeToGenInfo "endTime" "`date`"

[INFO]: Removing generated files from file-system as only batch job script printing
```

```
was requested
```

Next generate the script on resource:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2
[INFO]: Local copy of batch job script is
/home/mikola/wsp/test/akrr/data/rush/xdmod.app.md.namd/2014.12.11.23.08.56.260097/jobf
iles/xdmod.app.md.namd.job

[INFO]: Application kernel working directory on rush is
/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.11.23.08.56.260097
[INFO]: Batch job script location on rush is
/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.11.23.08.56.260097/x
dmod.app.md.namd.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
> cd
/panasas/scratch/nikolays/akrrdata/rush/xdmod.app.md.namd/2014.12.11.23.08.56.260097
> ls
xdmod.app.md.namd.job
> salloc --partition=debug --nodes=2 --ntasks-per-node=8 --time=01:00:00 --exclusive
--constraint="CPU-L5520|CPU-L5630"
salloc: Granted job allocation 3111545
> bash xdmod.app.md.namd.job
Temporary working directory: /panasas/scratch/nikolays/namd.PO1PlQHP1
Deleting temporary files
```

Examine appstdout file, which contains application kernel output:

On remote resource

```
> cat appstdout
===ExeBinSignature=== COMPILER: Intel Compiler Suite 9.0 (8 times, 8699 bytes)
<many more lines like previous>
===ExeBinSignature=== DYNLIB: Glibc 2.12
<many more lines like previous>
===ExeBinSignature=== MD5: 722f0a7211eb93225272d1b41ce72dc1
*/ifs/util/util64/namd/NAMD_2.9_Linux-x86_64-ibverbs/namd2
<many more lines like previous>Charmrun> started all node programs in 2.813 seconds.
Charmrun> IBVERBS version of charmrun
Converse/Charm++ Commit ID: v6.4.0-beta1-0-g5776d21
Charm++> scheduler running in netpoll mode.
CharmLB> Load balancer assumes all CPUs are same.
Charm++> Running on 2 unique compute nodes (8-way SMP).
Charm++> cpu topology info is gathered in 0.024 seconds.
Info: NAMD 2.9 for Linux-x86_64-ibverbs
Info:
Info: Please visit http://www.ks.uiuc.edu/Research/namd/
```

```
Info: for updates, documentation, and support information.
Info:
Info: Please cite Phillips et al., J. Comp. Chem. 26:1781-1802 (2005)
Info: in all publications reporting results obtained with NAMD.
Info:
Info: Based on Charm++/Converse 60400 for net-linux-x86_64-ibverbs-iccstatic
Info: Built Mon Apr 30 14:03:10 CDT 2012 by jim on dakar.ks.uiuc.edu
Info: 1 NAMD  2.9  Linux-x86_64-ibverbs  16    d07n33s01  nikolays
Info: Running on 16 processors, 16 nodes, 2 physical nodes.
Info: CPU topology information available.
Info: Charm++/Converse parallel runtime startup completed at 0.0293858 s
Info: 104.539 MB of memory in use based on /proc/self/stat
Info: Configuration file is ./input.namd
Info: Changed directory to .
TCL: Suspending until startup complete.
Info: SIMULATION PARAMETERS:
Info: TIMESTEP               2
Info: NUMBER OF STEPS        1200
Info: STEPS PER CYCLE        20
Info: PERIODIC CELL BASIS 1  108.861 0 0
Info: PERIODIC CELL BASIS 2  0 108.861 0
Info: PERIODIC CELL BASIS 3  0 0 77.758
Info: PERIODIC CELL CENTER   0 0 0

...
<more MD output>
...
LDB: ============= START OF LOAD BALANCING ============== 124.231
LDB: Largest compute 102 load 0.625922 is 5.3% of average load 11.834041
LDB: Average compute 0.063011 is 0.5% of average load 11.834041
LDB: TIME 124.252 LOAD: AVG 11.834 MAX 14.0314  PROXIES: TOTAL 320 MAXPE 21 MAXPATCH 4
None MEM: 273.891 MB
LDB: TIME 124.252 LOAD: AVG 11.834 MAX 14.0314  PROXIES: TOTAL 341 MAXPE 28 MAXPATCH 4
RefineTorusLB MEM: 273.891 MB
LDB: TIME 124.259 LOAD: AVG 11.834 MAX 12.0211  PROXIES: TOTAL 348 MAXPE 29 MAXPATCH 4
RefineTorusLB MEM: 273.891 MB
LDB: ============== END OF LOAD BALANCING =============== 124.287
Info: useSync: 1 useProxySync: 0
LDB: ================ DONE WITH MIGRATION ================ 124.421
Info: Benchmark time: 16 CPUs 0.159719 s/step 0.924299 days/ns 273.891 MB memory
Info: Benchmark time: 16 CPUs 0.163324 s/step 0.945163 days/ns 273.891 MB memory
TIMING: 500  CPU: 80.9697, 0.160976/step  Wall: 85.9195, 0.1706/step, 0.0331721 hours
remaining, 273.890625 MB of memory in use.
ENERGY:     500      2119.3671     10836.2513      5712.6270       177.9253
-300553.8492     16944.6540         0.0000         0.0000     33420.9359
-231342.0886       171.6325   -264763.0245   -231171.7203       168.3237
-1980.8569     -1951.1641    921491.4634     -1734.6988     -1734.6435

Info: Benchmark time: 16 CPUs 0.164331 s/step 0.950992 days/ns 273.891 MB memory
TIMING: 1000  CPU: 159.195, 0.15645/step  Wall: 164.16, 0.156481/step, 0.0086934 hours
remaining, 274.308594 MB of memory in use.
ENERGY:    1000      2032.8252     10860.4644      5705.6667       181.0165
-302020.5561     17931.8201         0.0000         0.0000     33958.7428
-231350.0205       174.3944   -265308.7632   -231170.8073       173.3328
-1704.1627     -1675.3187    921491.4634     -1821.4274     -1821.4057

WRITING EXTENDED SYSTEM TO OUTPUT FILE AT STEP 1200
WRITING COORDINATES TO OUTPUT FILE AT STEP 1200
The last position output (seq=-2) takes 0.232 seconds, 283.754 MB of memory in use
```

```
WRITING VELOCITIES TO OUTPUT FILE AT STEP 1200
The last velocity output (seq=-2) takes 0.228 seconds, 283.754 MB of memory in use
```

```
================================================

WallClock: 250.522873  CPUTime: 231.987732  Memory: 283.753906 MB
```

If it looks ok you can move to the next step

Perform Validation Run

On this step appkernel_validation.py utility is used to validate application kernel installation on particular resource. It execute application kernel and analyses its' results. If it fails the problems need to be fixed and another round of validation should be performed.

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py $RESOURCE $APPKER
```

appkernel_validation.py Sample output

```
Validating xdmod.app.md.namd application kernel installation on rush
################################################################################
Validating rush parameters from
/home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py
Syntax of /home/mikola/wsp/test/akrr/cfg/resources/rush/resource.inp.py is correct and
all necessary parameters are present.
Syntax of /home/mikola/wsp/test/akrr/src/appkernels/xdmod.app.md.namd.app.inp.py is
correct and all necessary parameters are present.
################################################################################
Validating resource accessibility. Connecting to rush.
================================================================================
Successfully connected to rush


Checking directory locations

Checking: rush:/panasas/scratch/nikolays/akrrdata/rush
Directory exist and accessible for read/write

Checking: rush:/user/nikolays/appker/rush
Directory exist and accessible for read/write

Checking: rush:/panasas/scratch/nikolays
Directory exist and accessible for read/write

Checking: rush:/tmp
Directory exist and accessible for read/write

################################################################################
Will send test job to queue, wait till it executed and will analyze the output
Will use AKRR REST API at https://localhost:8091/api/v1

Submitted test job to AKRR, task_id is 3144530



================================================================================
Tast status:
Task is in scheduled_tasks queue.
...
```

```
Task is in active_tasks queue.

================================================================================
Tast status:
Task is in active_tasks queue.
Status: Created batch job script and have submitted it to remote queue.
Status info:
Remote job ID is 3113860

time: 2014-12-12 11:11:58


================================================================================
Tast status:
Task is in active_tasks queue.
Status: Still in queue. Either waiting or running
Status info:
 JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
 3113860 general-c xdmod.ap nikolays R 3:39 2 d07n04s01,d07n07s01


time: 2014-12-12 11:20:00


================================================================================
Tast status:
Task is in active_tasks queue.
Status: Not in queue. Either exited with error or executed successfully. Copied all
files to local machine. Deleted all files from remote machine
Status info:
Not in queue. Either exited with error or executed successfully. Copied all files to
local machine. Deleted all files from remote machine

time: 2014-12-12 11:21:23

================================================================================
Tast status:
Task is in active_tasks queue.
Status: Output was processed and found that kernel either exited with error or
executed successfully.
Status info:
Done

time: 2014-12-12 11:21:28

================================================================================
Tast status:
Task is completed!
 status: 1
 statusinfo: Done

time: 2014-12-12 11:21:34

Test job is completed analyzing output


Test kernel execution summary:
status: 1
statusinfo: Done
```

```
processing message:
None
Local working directory for this task:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
Location of some important generated files:
 Batch job script:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/jobfiles/xdmod.app.md.namd.job
 Application kernel output:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/jobfiles/appstdout
 Batch job standard output:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/jobfiles/stdout
 Batch job standard error output:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/jobfiles/stderr
 XML processing results:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/result.xml
 Task execution logs:
/home/mikola/wsp/test/akrr/comptasks/rush/xdmod.app.md.namd/2014.12.12.11.11.42.749107
/proc/log



Enabling xdmod.app.md.namd on rush for execution
```

```
Successfully enabled xdmod.app.md.namd on rush

DONE, you can move to next step!
```

Schedule regular execution of application kernel.

Now this application kernel can be submitted for regular execution:

```
#Perform a test run on all nodes count
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time
between 1:00 and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8 -t0
"01:00" -t1 "05:00" -p 1
```

see Scheduling and Rescheduling Application Kernels and Setup Walltime Limit for more details

AKRR: Deployment of NWChem Applications Kernels on a Resource

Here the deployment of xdmod.app.chem.nwchem application kernel is described. This application kernel is based on NWChem application.

For further convenience of application kernel deployment lets define APPKER and RESOURCE environment variable which will contain the HPC resource name:

```
export RESOURCE=rush
export APPKER=xdmod.app.chem.nwchem
```

Install Application or Identify where it is Installed

NWChem is very often installed system-wide. One of the purposes of application kernels is to monitor the performance of application which is used by regular users. If NWChem is not installed then you might need to install this application on your resource or opted not to use it.

Majority of HPC resources utilize some kind of module system. Execute it and see is NWChem already installed

On resource

```
> module avail
...
nwchem/6.0
...
```

write down the name of module you want to use.

Generate Initiate Configuration File

Generate Initiate Configuration File:

On AKRR server

```
> python $AKRR_HOME/setup/scripts/gen_appker_on_resource_cfg.py $RESOURCE $APPKER
[INFO]: Generating application kernel configuration for xdmod.app.chem.nwchem on
Rush-debug
[INFO]: Application kernel configuration for xdmod.app.chem.nwchem on Rush-debug is
in:
        /user/akrr/akrr/cfg/resources/Rush-debug/xdmod.app.chem.nwchem.app.inp.py
```

Edit Configuration File

Below is listing of example configuration file located at $AKRR_HOME/cfg/resources/$RESOURCE/xdmod.app.chem.nwchem.app.inp.py

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.app.chem.nwchem.app.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application environment
module load nwchem
module list

#set executable location
EXE=`which nwchem`

#Set how to ran app kernel
RUN_APPKERNEL="mpirun -n $AKRR_CORES -hostfile $PBS_NODEFILE $EXE $INPUT"
"""
```

It contain only one parameter *appKernelRunEnvironmentTemplate* which need to be edited:

1) First part is "Load application environment", here you need to set proper enviroment. For example:

```
#Load application environment
module load nwchem
module list
```

2) Second part is "set executable location", it set the location of executables absolute path to nwchem should be placed to EXE variable (application signature will be calculated for that executable). For example:

```
#set executable location
EXE=`which nwchem`
```

3)Fourth part is "set how to run app kernel", it set RUN_APPKERNEL, which specify how to execute namd:

```
#Set how to ran app kernel
RUN_APPKERNEL="mpirun -n $AKRR_CORES -hostfile $PBS_NODEFILE $EXE $INPUT"
```

Configuration file for SLURM can look like:

$AKRR_HOME/cfg/resources/$RESOURCE/xdmod.app.chem.nwchem.app.inp.py

```
appKernelRunEnvironmentTemplate="""
#Load application environment
module load nwchem
module list
source
/util/academic/intel/composer_xe_2011_sp1.13.367/mkl/bin/intel64/mklvars_intel64.sh
export
LD_LIBRARY_PATH=/util/academic/intel/composer_xe_2011_sp1.13.367/mkl/lib/intel64:$LD_L
IBRARY_PATH
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
ulimit -s unlimited

#set executable location
EXE=`which nwchem`

#Set how to ran app kernel
RUN_APPKERNEL="srun $EXE $INPUT"
"""
```

Generate Batch Job Script and Execute it Manually (Optional)

The purpose of this step is to ensure that the configuration lead to correct workable batch job script. Here, at first batch job script is generated with 'akrr_ctl.sh batch_job'. Then this script is executed in interactive session (this improves the turn-around in case of errors). If script fails to execute, the issues can be fixed first in that script itself and then merged to configuration file.

This step is somewhat optional because it is very similar to next step. However the opportunity to work in interactive session improve turn-around time because there is no need to stay in queue for each iteration.

First generate the script to standard output and examine it:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -p -r $RESOURCE -a $APPKER -n 2
[INFO]: Below is content of generated batch job script:
#!/bin/bash
#SBATCH --partition=debug
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
##SBATCH --time=00:22:00
#SBATCH --time=01:00:00
#SBATCH
--output=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.13.59
.19.804826/stdout
#SBATCH
--error=/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.13.59.
19.804826/stderr
#SBATCH --constraint="CPU-L5520|CPU-L5630"
#SBATCH --exclusive

#Common commands
export AKRR_NODES=2
export AKRR_CORES=16
export AKRR_CORES_PER_NODE=8
export AKRR_NETWORK_SCRATCH="/gpfs/scratch/akrr"
```

```
export AKRR_LOCAL_SCRATCH="/scratch"
export
AKRR_TASK_WORKDIR="/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.0
3.24.13.59.19.804826"
export AKRR_APPKER_DIR="/user/akrr/appker/Rush-debug"
export AKRR_AKRR_DIR="/gpfs/scratch/akrr/akrrdata/Rush-debug"
export AKRR_APPKER_NAME="xdmod.app.chem.nwchem"
export AKRR_RESOURCE_NAME="Rush-debug"
export AKRR_TIMESTAMP="2015.03.24.13.59.19.804826"
export AKRR_APP_STDOUT_FILE="$AKRR_TASK_WORKDIR/appstdout"
export AKRR_APPKERNEL_INPUT="/user/akrr/appker/Rush-debug/inputs/nwchem/aump2.nw"
export AKRR_APPKERNEL_EXECUTABLE="/user/akrr/appker/Rush-debug/execs"
source "$AKRR_APPKER_DIR/execs/bin/akrr_util.bash"

#Populate list of nodes per MPI process
export AKRR_NODELIST=`srun -l --ntasks-per-node=$AKRR_CORES_PER_NODE -n $AKRR_CORES
hostname -s|sort -n| awk '{printf "%s ",$2}' `
export PATH="$AKRR_APPKER_DIR/execs/bin:$PATH"

cd "$AKRR_TASK_WORKDIR"

#run common tests
akrrPerformCommonTests

#Write some info to gen.info, JSON-Like file
writeToGenInfo "startTime" "`date`"
writeToGenInfo "nodeList" "$AKRR_NODELIST"

#create working dir
export AKRR_TMP_WORKDIR=`mktemp -d /gpfs/scratch/akrr/nwchem.XXXXXXXXX`
echo "Temporary working directory: $AKRR_TMP_WORKDIR"
cd $AKRR_TMP_WORKDIR

#Copy inputs
cp /user/akrr/appker/Rush-debug/inputs/nwchem/aump2.nw ./
INPUT=$(echo /user/akrr/appker/Rush-debug/inputs/nwchem/aump2.nw | xargs basename )
# set the NWCHEM_PERMANENT_DIR and NWCHEM_SCRATCH_DIR in the input file
# first, comment out any NWCHEM_PERMANENT_DIR and NWCHEM_SCRATCH_DIR in the input file
if [ -e $INPUT ]
then
    sed -i -e "s/scratch_dir/#/g" $INPUT
    sed -i -e "s/permanent_dir/#/g" $INPUT
    # then add our own
    echo "scratch_dir $AKRR_TMP_WORKDIR" >> $INPUT
    echo "permanent_dir $AKRR_TMP_WORKDIR" >> $INPUT
fi

#Load application environment
module load nwchem
module list

#set executable location
EXE=`which nwchem`
#Set how to ran app kernel
RUN_APPKERNEL="srun $EXE $INPUT"

#Generate AppKer signature
appsigcheck.sh $EXE $AKRR_TASK_WORKDIR/.. > $AKRR_APP_STDOUT_FILE
```

```
#Execute AppKer
writeToGenInfo "appKerStartTime" "`date`"
$RUN_APPKERNEL >> $AKRR_APP_STDOUT_FILE 2>&1
writeToGenInfo "appKerEndTime" "`date`"

#clean-up
cd $AKRR_TASK_WORKDIR
if [ "${AKRR_DEBUG=no}" = "no" ]
then
        echo "Deleting temporary files"
        rm -rf $AKRR_TMP_WORKDIR
else
        echo "Copying temporary files"
        cp -r $AKRR_TMP_WORKDIR workdir
        rm -rf $AKRR_TMP_WORKDIR
fi
writeToGenInfo "endTime" "`date`"

[INFO]: Removing generated files from file-system as only batch job script printing
```

```
was requested
```

Next generate the script on resource:

```
> $AKRR_HOME/bin/akrr_ctl.sh batch_job -r $RESOURCE -a $APPKER -n 2
[INFO]: Local copy of batch job script is
/gpfs/user/akrr/akrr/data/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.14.01.11.221494/
jobfiles/xdmod.app.chem.nwchem.job
[INFO]: Application kernel working directory on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.14.01.11.22149
4
[INFO]: Batch job script location on Rush-debug is
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.14.01.11.22149
4/xdmod.app.chem.nwchem.job
```

The output contains the working directory for this task on remote resource. On remote resource get to that directory and start interactive session (request same number of nodes, in example above the script was generated for 2 nodes).

On remote resource

```
#request interactive session
salloc --nodes=2 --ntasks-per-node=8 --time=01:00:00 --exclusive
--constraint="CPU-E5645"

#get to working directory
cd
/gpfs/scratch/akrr/akrrdata/Rush-debug/xdmod.app.chem.nwchem/2015.03.24.14.01.11.22149
4


#check job file presence
ls

#run job
bash xdmod.app.chem.nwchem.job

#examine output
cat appstdout
```

Examine appstdout file, which contains application kernel output:

appstdout

```
===ExeBinSignature=== COMPILER: Intel Fortran Compiler 11.1 (74 times, 661408 bytes)
<many lines like previous>
===ExeBinSignature=== DYNLIB: Glibc 2.12
<many lines like previous>
===ExeBinSignature=== MD5: 2cef1ed85019fbdcc7eeac0f34cc08b3
*/gpfs/util/academic/nwchem/nwchem-6.0/bin/nwchem-impi-openib
<many lines like previous>

...

ARMCI configured for 2 cluster nodes. Network protocol is 'OpenIB Verbs API'.
 argument  1 = aump2.nw




              Northwest Computational Chemistry Package (NWChem) 6.0
              ------------------------------------------------------

<many lines of quantum chemistry output>
 --------------
 CCSD(T) Energy
 --------------
 Reference energy:                   -134.684334043874742
 CCSD corr. energy:                    -0.654354731467711
 T(CCSD) corr. energy:                 -0.020388563019405
 Total CCSD+T(CCSD) energy:          -135.359077338361857
 CCSD corr. energy:                    -0.654354731467711
 (T) corr. energy:                     -0.020150171803070
 Total CCSD(T) energy:               -135.358838947145529

<should be done with>
 Total times  cpu:       48.2s     wall:       54.9s
```

If it looks ok you can move to the next step

Perform Validation Run

On this step appkernel_validation.py utility is used to validate application kernel installation on particular resource. It execute application kernel and analyses its' results. If it fails the problems need to be fixed and another round of validation should be performed.

```
python $AKRR_HOME/setup/scripts/appkernel_validation.py $RESOURCE $APPKER
```

Schedule regular execution of application kernel.

Now this application kernel can be submitted for regular execution:

```
#Perform a test run on all nodes count
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time
between 1:00 and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r $RESOURCE -a $APPKER -n 1,2,4,8 -t0
"01:00" -t1 "05:00" -p 1
```

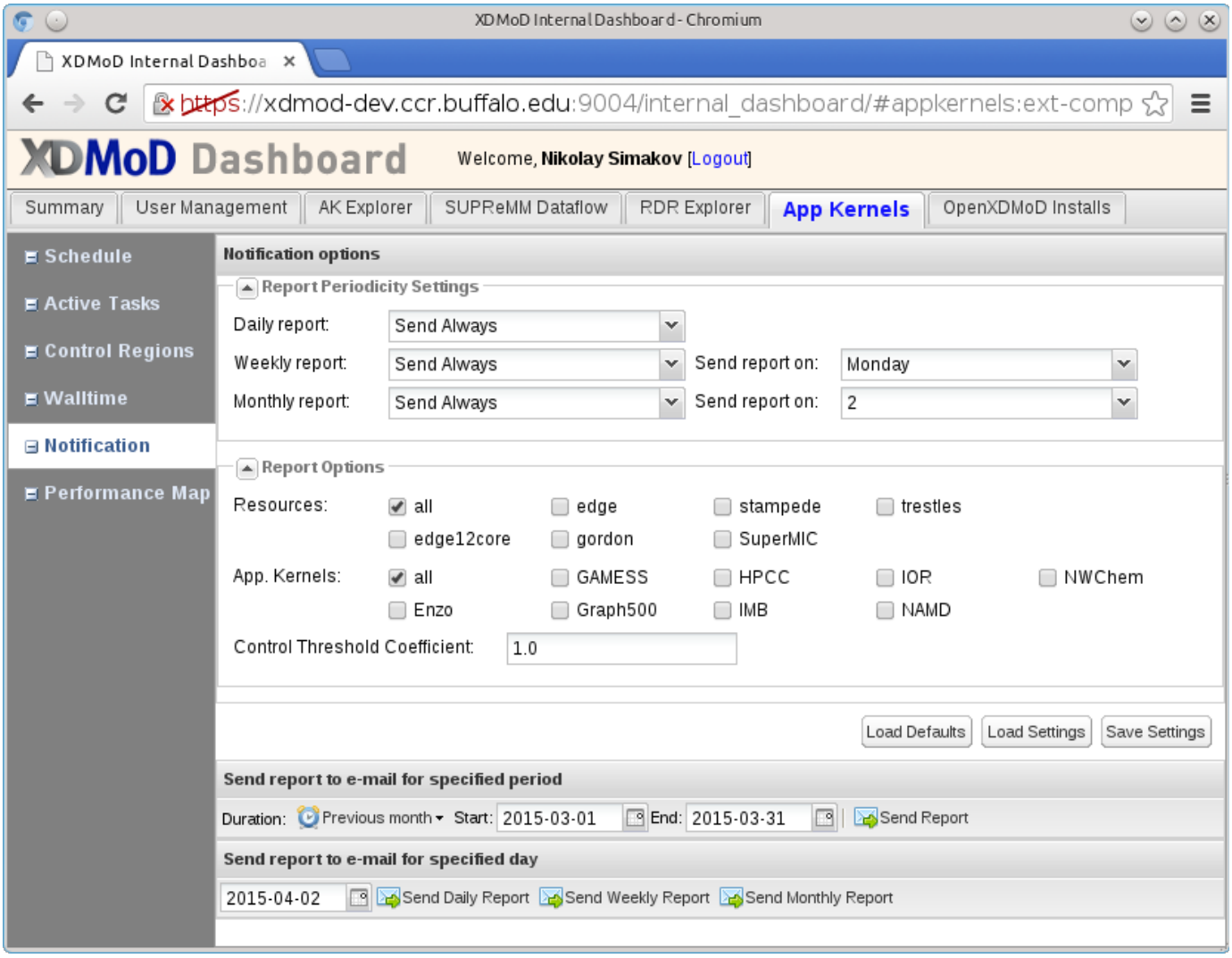
see Scheduling and Rescheduling Application Kernels and Setup Walltime Limit for more details

AKRR: e-Mail Report Generation

An e-mail report can be generated periodically and send to interested people. The report contains information about application kernel executed during the specified period. It also have automatic problem detection mechanisms

Setting up Periodic Report Generation

To sign-up for the e-mail report login to OpenXDMoD internal dashboard and select application kernel tab on top panel and then select notification panel on left panel (see figure below):



Notification settings panel allows to sign-up for periodic reports (daily, weekly and monthly). The report can be send either always or only when certain problem was detected. The reports can be generated for a selected resources and application kernels. The panel also allows immediate

report. generation for selected period.

Generated Report

Below is an example of generated report:

Subject: [XDMoD] Daily App Kernel Execution Report for 2015-04-01
From: XDMoD <donotreply@xdmod.ccr.buffalo.edu>
Date: 04/02/2015 12:26 PM
To: nikolays@ccr.buffalo.edu

Report Period: 2015-04-01

Summary for app kernels executed on 2015/04/01

Number of repeatedly failed runs : **0**
Number of repeatedly underperforming runs : **1**

Total number of runs: **9**
Number of failed runs: **0**
Number of runs without control information: **0**
Number of out of control runs: **1**
Number of runs within threshold: **8**
Number of application kernels in queue on resource as of 2015-04-02 12:26pm: **0**

Table 1. Summary Table of App Kernel Results for Each Resource

Resource	In Control Runs	Out Of Control Runs	No Control Information Runs	Failed Runs	Total Runs
edge12core	8 (88.9%)	1 (11.1%)	0 (0.0%)	0 (0.0%)	9
Total	8 (88.9%)	1 (11.1%)	0 (0.0%)	0 (0.0%)	9

Table 2. Summary of Underperforming (Out of Control) and Failed Runs

#	Resource	App. Kernel	Nodes	Message	Link to Details	Link to Performance Plot
1	edge12core	Graph500	2	Underperforming at least 3 times consecutively	Details	Plot
2	edge12core	IOR	1	Overperforming at least 3 times consecutively	Details	Plot

Table 3. Performance Heat Map of All App Kernels on Each System

KEY: Each day summarized in table cell as pair of a symbol and a number. Symbol represent the status of last application kernel execution on that day and number shows total number of runs. Each cell is colored according to the status of last application kernel run. Below is the codes description:

Code	Description
N	Application kernel was executed within control interval
U	Application kernel was under-performing
O	Application kernel was over-performing
F	Application kernel failed to run
C	This run was used to calculate control region
R	Application kernel have run, but control information is not available
	There was no application kernel runs

The status code is linked to full report of the last run.

	March, 2015													
Nodes	19	20	21	22	23	24	25	26	27	28	29	30	31	01	Plot
Resource: **edge12core** Application Kernel: **NAMD**, control region panel															
1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	Plot
2	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	Plot
4	N/1	N/1	F/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	F/1	N/1	Plot
8	N/1	N/1	F/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	N/1	F/1	N/1	Plot
Resource: **edge12core** Application Kernel: **Graph500**, control region panel															
1		N/1			N/1			N/1			N/1			N/1	Plot
2		N/1			U/1			U/1			U/1			U/1	Plot
4		N/1			N/1			N/1			N/1			N/1	Plot
8		N/1			N/1			N/1			N/1				Plot
Resource: **edge12core** Application Kernel: **IOR**, control region panel															
1	N/1		O/1		O/1		O/1		O/1		O/1		O/1		Plot
2		N/1		N/1		N/1		N/1		N/1		N/1		N/1	Plot
4	N/1		N/1		N/1		N/1		N/1		N/1		N/1		Plot
8		N/1		F/1		N/1		F/1		N/1		N/1		N/1	Plot

The reports starts with brief summary of application kernel execution on the reported period. Table 1 represent summary by resources. Table 2 shows automatically detected problem. And Table 3 show the performance map where each day is summariez by a symbol and color code. Many places in the report are links and will bring to OpenXDMoD with more detailed information.

AKRR: Scheduling and Rescheduling Application Kernels

Application Kernels can be scheduled for execution using two interfaces: 1) command line interface and 2) using OpenXDMoD user interface.

Command Line Interface

`new_task` command for AKRR command line interface will set up a new task(s), below is help output:

```
> python $AKRR_HOME/src/akrrctl.py new_task -h
usage: akrrctl.py new_task [-h] [-r RESOURCE] [-a APPKERNEL] [-n NODES]
                           [-s START_TIME] [-t0 TIME_START] [-t1 TIME_END]
                           [-p PERIODICITY]
Create a new task given the specified parameters
optional arguments:
  -h, --help            show this help message and exit
  -r RESOURCE, --resource RESOURCE
                        Specify the resource that the new task should be
                        created for.
  -a APPKERNEL, --appkernel APPKERNEL
                        Specify which application kernel to use for the new
                        task.
  -n NODES, --nodes NODES
                        Specify how many nodes the new task should be setup
                        with.
  -s START_TIME, --start_time START_TIME
                        Specify what time the newly created task should start.
  -t0 TIME_START, --time_start TIME_START
                        Specify the time at which the random distribution
                        begins
  -t1 TIME_END, --time_end TIME_END
                        Specify the time at which the random distribution ends
  -p PERIODICITY, --periodicity PERIODICITY
                        Specify the amount of time that should elapse between
                        executions.
```

Some helpful examples:

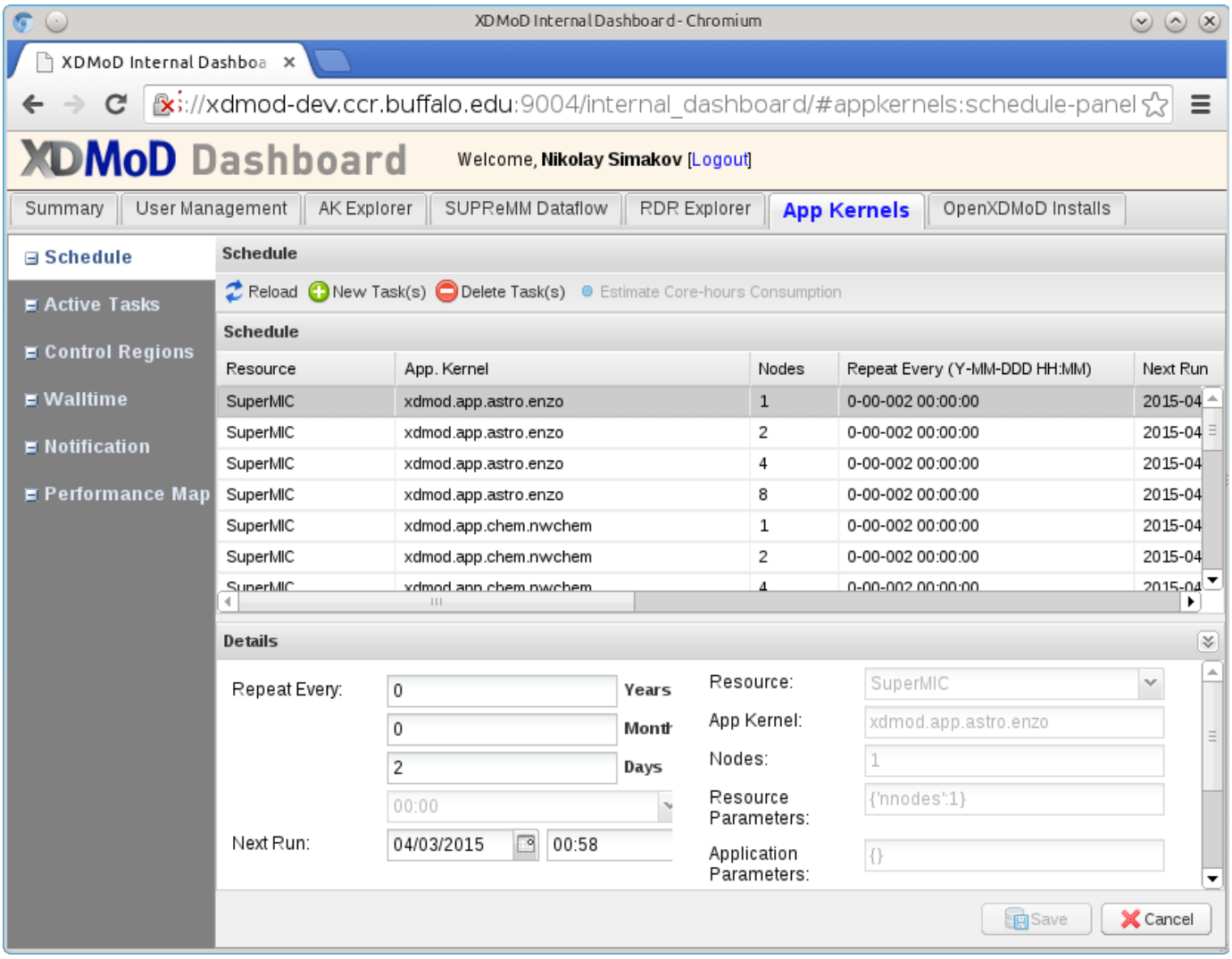
```
#Start xdmod.app.md.namd on edge_test at 2015-4-27 01:00 with periodicity of 1 day
python $AKRR_HOME/src/akrrctl.py new_task -r edge_test -a xdmod.app.md.namd -n 2 -s
"2015-4-27 01:00" -p "1 00:00"

#Same but start today at 1
python $AKRR_HOME/src/akrrctl.py new_task -r edge_test -a xdmod.app.md.namd -n 2 -s
"01:00" -p 1

#Start immediately on nodes 1,2,4,8 but don't execute periodically
python $AKRR_HOME/src/akrrctl.py new_task -r edge_test -a xdmod.app.md.namd -n 1,2,4,8

#Start daily execution from today on nodes 1,2,4,8 and distribute execution time
between 1:00 and 5:00
python $AKRR_HOME/src/akrrctl.py new_task -r edge_test -a xdmod.app.md.namd -n 1,2,4,8
-t0 "01:00" -t1 "05:00"
```

OpenXDMoD Schedule Panel



In internal dashboard select app kernel tab from top panel and then select schedule tab from left panel. The panel allows to create new tasks, delete the old one as well as modify them.

AKRR Server Module Installation Guide

- Introduction
- Services: (Servers / Databases)
- Pre-Installation Steps
- Information to Have On Hand
- Installation Steps
  - Unpack Archive
  - Possible Preparatory Work (not recommended)
  - AKRR Setup and AKRR Daemon First Start
  - Checking and Updating AKRR Configuration File
  - Checking the Installation
- Adding Resources for Running Application Kernels

Introduction

The installation of the Application Kernel Remote Runner (AKRR) is a multi step process that, while not difficult, benefits from careful attention to the steps laid out below. We will begin with a basic overview of the services required by AKRR followed by some pre-installation steps that will need to be taken to ensure a successful installation. Next we'll walk through the installation and subsequent setup with a detailed discussion of what is going on during each step and finally we'll wrap up with how to test if your installation is functioning properly.

Services: (Servers / Databases)

The service breakdown for AKRR looks something like this:

- AKRR Host Server (Where AKRR resides and where it will serve a REST-ful API from)
- mod_akrr Database
- mod_appkernel Database
- modw (XDMoD) Database

In tests, the AKRR Host Server, mod_akrr database and mod_appkernel database were all located on the same VM but configurations exist such that they can be separated if it better suits your infrastructure.

Pre-Installation Steps

Before we begin the actual installation there are a few pieces of software that will need to be installed on the AKRR / AK server. My examples will assume a Debian / Ubuntu OS, please adjust to fit your OS / package manager.

Execute the following statement:

```
sudo apt-get install python2.7 python2.7-mysqldb openssl curl
```

OPTIONAL If the MySQL server will be hosted locally, install the server package

```
sudo apt-get install mysql-server
```

This will install the basic python environment along with the required database drivers and (if it's not already installed).

Information to Have On Hand

Since we assume that if you are installing AKRR / AK that you already have a working XDMoD environment.

You **must** have on hand the following pieces of information about the XDMoD database.

- the root db user and their password (or similar account) for this db server / instance.

For 'mod_akrr' and 'mod_appkernel' access we support the following scenarios:

- Using an existing user that will be granted the required rights.
- Supplying the script with the required information to create a new user with the required rights.

For 'modw' access we support the following scenarios:

- Using an existing user that will be granted the rights required.
- Supplying the script with the required information to create a new user with the required rights.

NOTE: If you do choose to use an existing user take extra care when entering the password as the install script auto-generates the config files that AKRR utilizes when connecting and this can be a common source of error.

Installation Steps

Unpack Archive

Untar the 'akrr.tar.gz' file where you would like to install AKRR, set AKRR_HOME environment variable for easy reference and enter AKRR top directory

```
tar -xvf akrr.tar.gz -C <put_your_desired_installation_dir_here>
export AKRR_HOME=<your_desired_installation_dir>/akrr
cd $AKRR_HOME
```

The following directories should be seen in $AKRR_HOME:

Directory	Description
`bin`	contains execution scripts
`cfg`	AKRR configuration
src	AKRR sources
`3rd_party`	3rd party libraries used by AKRR
`setup`	Contains AKRR installation scripts
`appker_repo`	Application Kernel repositary, contains input parameters for application kernel execution, application kernel signature calculator and some application kernels sources
`data`	After deployment will contain logs and active tasks, this location can be reconfigured
`comptasks`	After deployment will contain completed tasks, this location can be reconfigured

Possible Preparatory Work (not recommended)

modw.resourcefact table for testing resource importing from OpenXDMoD
In this point OpenXDMoD already supposed to be installed. AKRR uses modw database and resourcefact to import some information about resources. If for some reason you decided to install AKRR alone, so you need to create modw database and resourcefact table to emulate its presence. modw.resourcefact is only used during resource importing. To create modw.resourcefact execute following:

```
mysql -u root -p < $AKRR_HOME/setup/prep/modw_faking.sql
```

AKRR Setup and AKRR Daemon First Start

During this step AKRR will be configured and AKRR daemon will be started. In more details the setup script creates AKRR databases, sets new user for accessing these databases, creates self-signed SSL certificate for AKRR REST API (AKRR REST API used only internally for communication between AKRR and OpenXDMoD), creates and populates AKRR database tables, followed by first start of AKRR daemon and creation of cronjobs for periodic log rotation and AKRR daemon status check.

Execute the 'setup.sh' script. Provide the required information when prompted (for python location, the database user names / passwords).

NOTE: This script should be run as your akrr user (if you have created one). It will modify your .bashrc file and your crontab.

```
$AKRR_HOME/setup/setup.sh
```

Successful execution will look like:

setup.sh Sample Output

```
[INFO]: Attempting to load installation configuration...
[INFO]: configuration created!
[INPUT]: Please specify a path to the python binary you want to use:
[/usr/bin/python]
[INFO]: Checking python version...
[INFO]: Your python version is just right!
[INFO]: Before Installation continues we need to setup the database.
[INPUT]: Please specify a database user for AKRR (This user will be created if it does not already exist):
[akrruser]
[INPUT]: Please specify a password for the AKRR database user:
Password:
[INPUT]: Please reenter password:
Password:
[INPUT]: Please specify the user that will be connecting to the XDMoD database (modw):
[akrruser]
[INFO]: Same user as for AKRR database user, will set same password
[INPUT]: Please provide an administrative database user under which the installation sql script should
run (This user must have privileges to create users and databases):
root
[INPUT]: Please provide the password for the the user which you previously entered:
Password:
[INPUT]: Please enter the e-mail where cron will send messages (leave empty to opt out):
nikolays@buffalo.edu
[INFO]: Creating AKRR databases and granting permissions for AKRR user.
[INFO]: Generating self-signed certificate for REST-API
Generating a 4096 bit RSA private key
writing new private key to '/home/mikola/work/akrr_ci/akrr/cfg/server.key'
[INFO]: New self-signed certificate have been generated
[INFO]: Generating Settings File...
[INFO]: Settings written to: /home/mikola/work/akrr_ci/akrr/cfg/akrr.inp.py
[INFO]: Removing access for group members and everybody for all files as it might contain sensitive information.
[INFO]: Checking 'mod_akrr' Database / User privileges...
[INFO]: 'mod_akrr' Database check complete - Status: True
[INFO]: Checking 'mod_appkernel' Database / User privileges...
[INFO]: 'mod_appkernel' Database check complete - Status: True
[INFO]: Checking 'modw' Database / User privileges...
[INFO]: 'modw' Database check complete - Status: True
[INFO]: All Databases / User privileges check out!
**************************************************
Creating mod_akrr Tables / Views...
**************************************************
CREATING: ACTIVETASKS
Result of: ACTIVETASKS -> 0
...
**************************************************
mod_akrr Tables / Views Created!
**************************************************
Creating mod_appkernel Tables / Views...
**************************************************
CREATING: a_data
Result of: a_data -> 0
CREATED: a_data SUCCESSFULLY!
```

```
...
************************************************
mod_appkernel Tables / Views Created!
Directory /home/mikola/work/akrr_ci/akrr/data/srv does not exist, creating it.
Writing logs to:
      /home/mikola/work/akrr_ci/akrr/data/srv/2015.01.20_13.09.000963.log
AKRR Server PID is 10284
Redirect the standard I/O to
      /home/mikola/work/akrr_ci/akrr/data/srv/2015.01.20_13.09.000963.log
following log: /home/mikola/work/akrr_ci/akrr/data/srv/2015.01.20_13.09.000963.log
Starting Application Remote Runner
AKRR Scheduler PID is 10284
Starting REST-API Service

##############################################################################################################
###################
# Got into the running loop on 2015-01-20 13:09:45
##############################################################################################################
###################

Starting REST-API Service
Bottle v0.12.7 server starting up (using SSLWSGIRefServer())...
Listening on https://localhost:8091/

AKRR Server successfully reached the loop.
[INFO]: Beginning check of the AKRR Rest API...
[INFO]: REST API is up and running!
[INFO]: Cron Scripts Processed!
[INFO]: Updating .bashrc
[INFO]: Updating AKRR record in $HOME/.bashrc, backing to $HOME/.bashrc_akrrbak
[INFO]: Appended AKRR records to $HOME/.bashrc
[INFO]: AKRR Installed Successfully!
```

If the last message you see states that AKRR was not installed then there should be some descriptive error messages above that should help point you to the right corrective action. The most likely issues revolve around interactions with the database, whether that means issues with firewalls or issues with user permissions. Now the AKRR server should be up and running.AKRR daemon can be stoped as:

```
$AKRR_HOME/bin/akrr.sh stop
```

or started as:

```
$AKRR_HOME/bin/akrr.sh start
```

The status of the AKRR daemon can be checked as:

```
$AKRR_HOME/bin/akrr.sh status
```

Note that AKRR daemon status checker is setup through cronjob and in order to start AKRR after reboot, power failure or AKRR internal problem.Therefore if AKRR daemon should be completely stopped respective cronjob entries should be deleted.

Checking and Updating AKRR Configuration File

In the previous step the AKRR Configuration File ($AKRR_HOME/cfg/akrr.inp.py) was automatically generated and should be ready for use. Unless it is a non-standard installation, there should be no need for any changes. The configuration file should look like:

$AKRR_HOME/cfg/akrr.inp.py Expand source

```
##############################################################################
# XDMoD DB
```

```
###############################################################################
# Hostname of the database currently sering the 'modw' database ( the main XDMoD
database ).
xd_db_host = "localhost"
# The port that the 'modw' database is currently being served from.
xd_db_port = 3306
# The user that has read only access to the 'modw.resourcefact' table.
xd_db_user = "akrruser"
# The password for the 'xd_db_user'
xd_db_passwd = "qwe"
# The name that has been chosen for the 'modw' database. Note: CHANGE THIS AT YOUR
OWN RISK.
xd_db_name = "modw"
#==============================================================================
# MOD_AKRR DATABASE
#==============================================================================
# The host name of the database server that the 'mod_akrr' database is served from.
akrr_db_host = xd_db_host
# Port that the 'mod_akrr' database is being served from.
akrr_db_port = xd_db_port
# Database user that will have full access to the 'mod_akrr' database.
akrr_db_user = "akrruser"
# Password for the 'akrr_db_user'
akrr_db_passwd = "qwe"
# The name that has been chosen for the 'mod_akrr' database. Note: CHANGE THIS AT
YOUR OWN RISK.
akrr_db_name = "mod_akrr"
#==============================================================================
# External DB (In most cases same as Internal DB)
# NOTE: this database ( and the credentials required ) are usually the same as
#       the 'mod_akrr' database.
#==============================================================================
export_db_host = akrr_db_host
export_db_port = akrr_db_port
export_db_user = akrr_db_user
export_db_passwd = akrr_db_passwd
export_db_name = akrr_db_name
###############################################################################
# App Kernel DB
# NOTE: this database ( and the credentials required ) are usually the same as
#       the 'mod_akrr' database.
###############################################################################
# Hostname of the database serving the 'mod_appkernel' database.
ak_db_host = akrr_db_host
# Port that the 'mod_appkernel' database is being served from.
ak_db_port = akrr_db_port
# User with full access to the 'mod_appkernel' database.
ak_db_user = akrr_db_user
# Password for the 'ak_db_user'
ak_db_passwd = akrr_db_passwd
# The name that has been chosen to represent the 'mod_appkernel' database. Note:
CHANGE THIS AT YOUR OWN RISK.
ak_db_name = "mod_appkernel"
###############################################################################
# REST API
###############################################################################
# The hostname of the server that will be serving the RESTAPI. If you are testing
and want to bind it to the loop-back
# address please note that 'localhost' produced more positive results than
```

```
'127.0.0.1' did. Your mileage may vary.
restapi_host = "localhost"
# The port that the REST API will attempt to bind to on startup. Please change if
you have a conflict.
# Please also ensure that this port is available for connection ( aka. please create
a firewall rule if necessary. ).
restapi_port = 8091
# the root url fragment that will be pre-pended to all REST API routes [ ex. GET
https://restapi/api/v1/scheduled_tasks
# hits the 'scheduled_tasks' route of the REST API ]. This fragment allows for
versioning of the API.
restapi_apiroot = '/api/v1'
# The name of the SSL cert file ( required for HTTPS connections )
restapi_certfile = 'server.pem'
# Token expiration time in seconds
restapi_token_expiration_time = 3600
# User defined as having 'read / write' permission to the REST API
restapi_rw_username = 'rw'
# The password for the 'rw' user.
restapi_rw_password = "f3xStkrcj0frPsul"
# User defined as having 'read-only' premissions to the REST API
restapi_ro_username = 'ro'
# The password for the 'ro' user.
restapi_ro_password = "GFyIyqpWiolqLzEN"
################################################################################
# Directories layout (relative paths are relative to location of this file)
################################################################################
# This location is used to store various bits of information about the AKRR
# process such as the .pid file ( to track when AKRR is running ) as well as
# logs.
data_dir = "../data"
# This location is used to
completed_tasks_dir = "../comptasks"
################################################################################
#
#  PARAMETERS BELLOW THIS POINT DO NOT OFTEN NEED TO BE CHANGED.
#  PROCEED AT YOUR OWN RISK!
#
################################################################################
################################################################################
# AKRR parameters
################################################################################
#Number of sub-processes (workers) to handle tasks
max_task_handlers = 4
# The 'id' of the pickling protocol to use.
task_pickling_protocol = 0
# The amount of time that the tasks loop should sleep in between loops.
scheduled_tasks_loop_sleep_time = 1.0
################################################################################
# Error handling and repeat time
################################################################################
# class datetime.timedelt format
# class datetime.timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[,
hours[, weeks]]]]]]])
import datetime
############################
# Default error handking
############################
# Maximal number of regular fatal errors (regular in sense no special treatment)
```

```
max_fatal_errors_for_task = 10
# Default repeat time
active_task_default_attempt_repeat = datetime.timedelta(minutes=30)
#**************************
# handler hangs
# ############################
# maximal time for task handler single execution
max_wall_time_for_task_handlers = datetime.timedelta(minutes=30)
# time to repeat after termination
repeat_after_forcible_termination = active_task_default_attempt_repeat
# Failure to submit to the queue on remote machine hangs, usually an issue on
machine with queue limits
max_fails_to_submit_to_the_queue = 48  # i.e.2 days
# amount of time to wait to submit the task back to the queue if it fails.
repeat_after_fails_to_submit_to_the_queue = datetime.timedelta(hours=1)
# Maximum amount of time a task is allowed to stay in the queue.
max_time_in_queue = datetime.timedelta(days=10)  # i.e. 10 days
# The amount of time that should elapse between attempts to connect to the 'export'
db.
export_db_repeat_attempt_in = datetime.timedelta(hours=1)
# The maximum number of attempts that should be made to connect to the 'export' db.
```

```
export_db_max_repeat_attempts = 48
# The default parameters that should be made available to each task.
default_task_params = {'test_run': False}
```

After the configuration file updates the AKRR daemon should be restarted:

```
$AKRR_HOME/bin/akrr.sh stop
$AKRR_HOME/bin/akrr.sh start
```

Note that only in case of AKRR configuration file ($AKRR_HOME/cfg/akrr.inp.py) updating the daemon should be restarted, if remote resource or application kernels configuration are updated their new parameters will be reloaded by daemon automatically upon the next use.

Checking the Installation

To check current status of AKRR service, one can run:

```
$AKRR_HOME/bin/akrr.sh status
```

or to check REST-API availability

```
python $AKRR_HOME/setup/scripts/akrrcheck_daemon.py
```

REST-API can also be checked manually, for example with use of curl utility:

```
#username is rw and password is in $AKRR_HOME/cfg/akrr.inp.py
#first get token
> curl -u rw:f1FVf9v-bqFyb7XL -X GET -k https://localhost:8091/api/v1/token
{
    "message": "success",
    "data": {
        "token": "2zmpOml9dpOmSkO6EVDfGYyvXEQk8jtg"
    },
    "success": true
}
#now lets check current tasks
> curl -u 2zmpOml9dpOmSkO6EVDfGYyvXEQk8jtg: -X GET -k
https://localhost:8091/api/v1/tasks
{
    "message": "success",
    "data": [],
    "success": true
}
#none which have sense
```

Adding Resources for Running Application Kernels

Now that the AKRR service is configured, you will need to add and configure resources where application kernels will be run.

Setup Walltime Limit

AKRR allows to specify walltime limits on multiple level from generic application kernel to the instance of task. The purpose of that is that on top level the walltime limit should fit most platform and way on the bottom it allows to specify very accurate walltime limit specific to particular resource and nodes count to reduce hit on scheduler and waiting times.

The walltime limit common for all nodes for specific application kernel on specific resource can be specified in configutarion file $AKRR_HOME/cfg/resources/$RESOURCE/$APPKER.inp.py (where $RESOURCE and $APPKER is the name of resource and application kernel). Just add following to that file:

```
#walltime limit in minutes common for all node counts
walllimit=20
```

In order to specify node specific walltime limit command line interface or using OpenXDMoD user interface can be used.

Command Line Interface

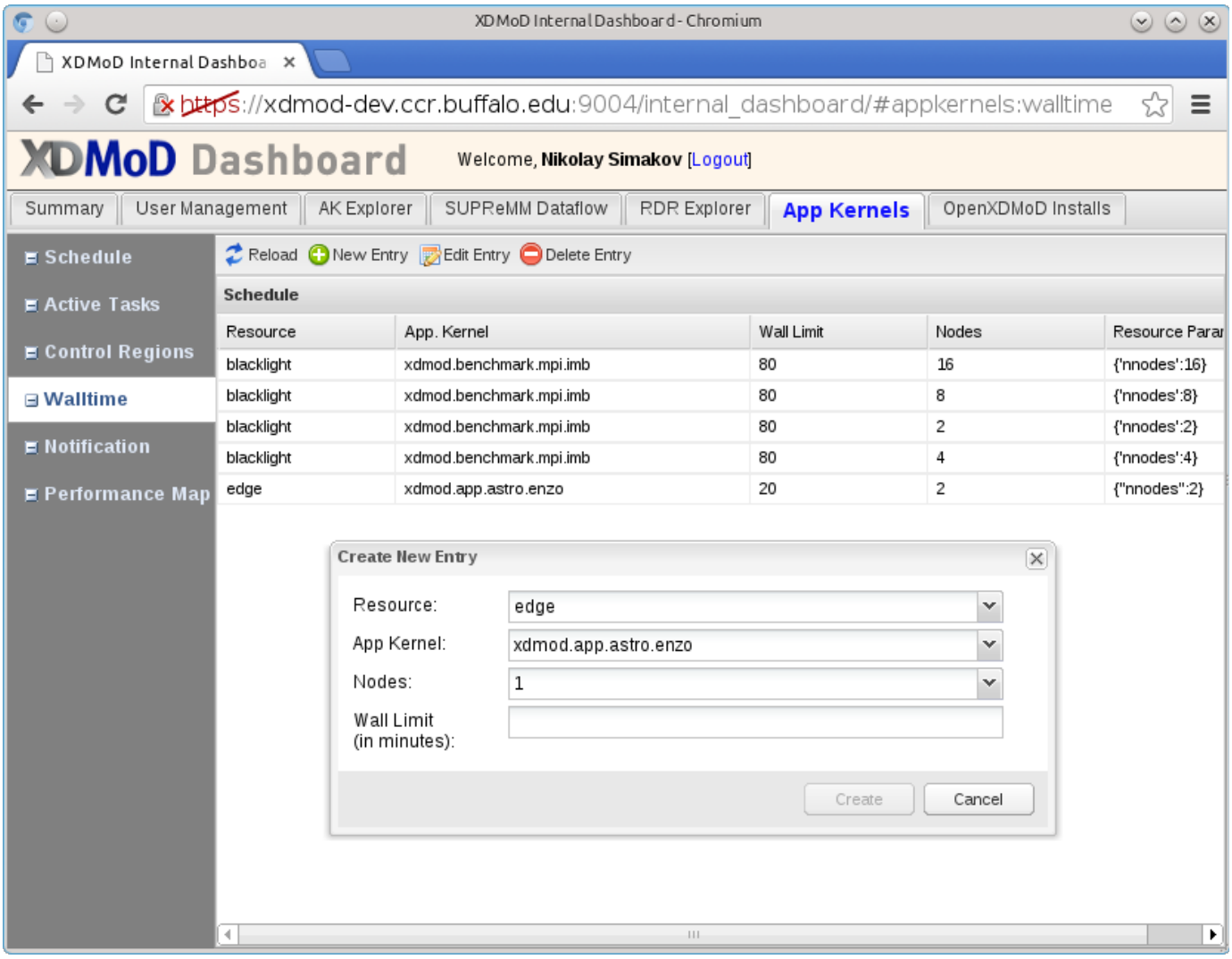
`walltime command for AKRR command line interface will set up walltime limits for specified application kernel on selected number of nodes of particular resource, below is help output:`

```
> python $AKRR_HOME/src/akrrctl.py walltime -h
usage: akrrctl.py walltime [-h] [-r RESOURCE] [-a APPKERNEL] [-n NODES]
                           [-w WALLTIME] [-c COMMENTS] [-l]
Update or insert a new wall time limit for the tasks matching the specified
parameters.
optional arguments:
  -h, --help            show this help message and exit
  -r RESOURCE, --resource RESOURCE
                        Specify the resource filter that the new wall time
                        should be applied to
  -a APPKERNEL, --appkernel APPKERNEL
                        Specify the application filter that the wall time
                        should be applied to.
  -n NODES, --nodes NODES
                        Specify the number of nodes filter that the wall time
                        should be applied to.
  -w WALLTIME, --walltime WALLTIME
                        Specify the wall time value (in minutes) that should be used
during
                        update or insert operation.
  -c COMMENTS, --comments COMMENTS
                        Comments
  -l, --list            List the wall time records that have been entered
                        already. Providing this switch allows the resource
                        (-r), appkernel (-a), nodes (-n) and walltime (-w)
                        arguments to become optional. When provided they with
                        'list' they will filter the records to be returned.
```

Some helpful examples:

```
#set walltime limit to 5 minutes for xdmod.app.md.namd on 2 nodes of edge_test
python $AKRR_HOME/src/akrrctl.py walltime -r edge_test -a xdmod.app.md.namd -n 2 -w 5
```

OpenXDMoD Walltime Panel



In internal dashboard select app kernel tab from top panel and then select walltime tab from left panel. The panel allows to define new walltime limits, delete the old one and modify them.