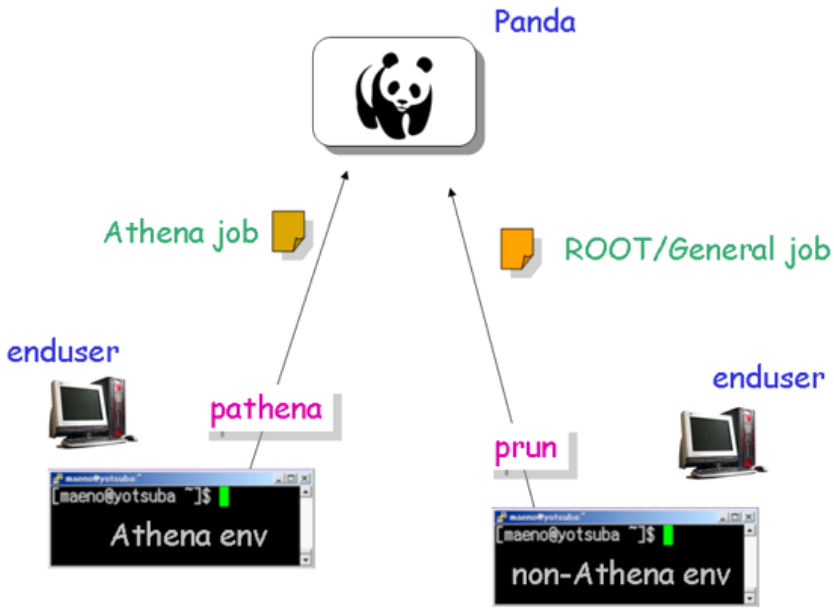
Table of Contents

How to submit ROOT/general jobs to Panda

Introduction

prun is a Panda-client software which allows users to submit general jobs to Panda. Those jobs may run ROOT(CINT,C++,pyRoot), ARA, Python, user's executable, shell script and so on. It is intended to support non-Athena type analysis, e.g., ROOT-based analysis where Athena runtime is not always available. Thus, prun can work on non-Athena runtime environment as well as Athena runtime environment. All files under the current directory are sent to remote worker-nodes (WNs) except root/large files by default. Users can construct arbitrary runtime environment on the WNs and can do anything in principle. However, please avoid careless network operations connecting to remote servers (e.g., cmt checkout, wget, and lcg-cp) unless administrators of the remote servers permit them. In the worst case, you will be banned from the grid. Your job is split to many sub-jobs and these sub-jobs are executed in parallel, so that those operations tend to become a DDoS attack and easily break the remote servers.



Getting started

Installation

prun is included in the panda-client package. See Installation and Setup for panda-client.

How to run (an example of ARA job)

The usage of prun is

```
$ prun [options]
```

where `--exec` and `--outDS` are mandatory. Try

```
$ prun -h
```

to see all available options.

Here is an example of ARA job.

```
$ cat aratest.py
```

```
def main():
    import sys
    import user
    import ROOT
    import PyCintex
    # output
    outF = ROOT.TFile('out1.root','recreate')
    import AthenaROOTAccess.transientTree
    CollectionTree = ROOT.AthenaROOTAccess.TChainROOTAccess('CollectionTree')
    # input
    inputFiles = sys.argv[-1].split(',')
    for inputFile in inputFiles:
        print "add %s" % inputFile
        CollectionTree.Add(inputFile)
    tt = AthenaROOTAccess.transientTree.makeTree(CollectionTree)
    # event loop
    for i in range(tt.GetEntries()):
        tt.GetEntry(i)
        photons = tt.PhotonAODCollection
        print [e.eta() for e in photons]

if __name__ == "__main__":
    main()
```

This script gets a list of input filenames via `sys.argv[1]` and produces an output file `out1.root`. You may locally run this script like

```
$ python aratest.py - AOD1.pool.root,AOD2.pool.root
```

Note that you need to put "-" between aratest.py and input files to prevent pyROOT from parsing command-line arguments (see a roottalk topic). Or you can put `ROOT.PyConfig.IgnoreCommandLineOptions=True` in aratest.py if you use ROOT-5.24 or higher. More detailed information is available at this link. Now you can submit this to Panda by using prun. All you need is essentially to convert the execution string (python aratest.py - AOD1.pool.root,AOD2.pool.root) to --exec "python aratest.py - %IN" and to give it to prun, where `%IN` represents a list of input files and will be replaced to the real list on each WN.

```
$ prun --exec "python aratest.py - %IN" --outDS user.tmaeno.test123 --inDS valid1.006384.PythiaH1
```

prun gathers all files under the `--workDir` directory (default=./ or `$TestArea` if `--useAthenaPackages` is used) and sends them to WNs. A single prun job instantiates one buildGen job and some runGen jobs like pathena jobs. Once jobs are submitted to Panda, you can see what's going on using PandaMonitor. buildGen stores your source files to the remote SE. Especially buildGen compiles Athena packages on a WN and stores binary files to the SE, if the `--useAthenaPackages` option is used (see detail). runGen jobs get activated as soon as buildGen finishes. runGen jobs run user's application (aratest.py in this example) over files in the `--inDS` dataset (or dataset container). The argument of the `--exec` option is executed after converting `%IN` to a list of input files on each WN. The list is a comma-separated string including names of input files available in the 'current directory' on the WN or TURLs (such as `dcap://blah...`) if the site is configured to use direct access for input files. Output files are defined in the `--outputs` option and are automatically renamed to `DatasetName_SerialNumber_OriginalName`, e.g, `user.tmaeno.test123._00001.out1.root`, and then are added to the `--outDS` dataset. Concerning the naming convention of the output dataset, see this page. You can retrieve output files by using dq2 tools. See another ARA example as well.

More Examples

The following sections show some example implementations to run ROOT and/or Python jobs. There are other possible implementations, of course. Basically the prun framework has no constraint, so users can write their own scripts or applications as they like. The point is that the --exec option defines how the list of input files is given to scripts/applications.

Sourcing ROOT for ROOT applications

ROOT is not sourced by default on the grid WNs. There are two ways to get ROOT:

Include an Athena release

ROOT applications can use the `--athenaTag` option which is explained in this section.

Use newer ROOT which is not included in ATLAS release

`--rootVer` option allows users to use any ROOT version on WN even if it is not yet included in Athena. For example,

```
prun --rootVer 5.32.00
```

this will setup ROOT-5.32.00. If you want to use 64bit version, use `--cmtConfig=x86_64-slc5-gcc43-opt` , `x86_64-slc6-gcc46-opt`, or `x86_64-slc6-gcc47-opt`.

Run CINT macro

Here is an example of CINT macro.

```
$ cat macrotest.C

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>

gROOT->Reset();

void macrotest()
{
  // read a string via file since long string causes memory error in CINT when it is read via std
  std::string argStr;
  std::ifstream ifs("input.txt");
  std::getline(ifs,argStr);

  // split by ','
  std::vector<std::string> fileList;
  for (size_t i=0,n; i <= argStr.length(); i=n+1)
    {
      n = argStr.find_first_of(',',i);
      if (n == string::npos)
        n = argStr.length();
      string tmp = argStr.substr(i,n-i);
      fileList.push_back(tmp);
    }

  // open input files
  TChain fChain("CollectionTree");
  for (int iFile=0; iFile<fileList.size(); ++iFile)
```

```
    {
      std::cout << "open " << fileList[iFile].c_str() << std::endl;
      fChain.Add(fileList[iFile].c_str());
    }

  Int_t           EventNumber;
  TBranch        *b_EventNumber;
  fChain.SetBranchAddress("EventNumber", &EventNumber, &b_EventNumber);

  // main loop
  Long64_t nentries = fChain.GetEntriesFast();
  for (Long64_t jentry=0; jentry<nentries;jentry++)
    {
      Long64_t ientry = fChain.LoadTree(jentry);
      if (ientry < 0)
        break;
      fChain.GetEntry(jentry);

      std::cout << EventNumber << std::endl;
    }
}
```

This macro reads a string via file and splits the string to the list of input files. You may run this locally like

```
$ echo NTUP1.root,Ntup2.root > input.txt; root.exe macrotest.C
```

So you can submit it as

```
$ prun --exec "echo %IN > input.txt; root.exe -b -q macrotest.C" --athenaTag=14.2.24 --inDS ...
```

Note: the root flags -b -q were found to be useful to tell root to run in batch mode (required for grid running) and quit the thread when finished (to prevent hanging), especially with Athena release 16.

Run C++ ROOT

Although pre-compiled binaries can be sent to WNs, it is safe to re-compile on remote WNs to avoid library mismatch. In this case, you need to prepare source files, Makefile and a wrapper script. FYI, it may be convenient to use to use the `--useAthenaPackages=` option which is described in this section if you need to compile multiple Athena packages. Here is a C++ example.

```
$ cat cpptest.cc

#include <string>
#include <vector>
#include <iostream>
#include <stdlib.h>

#include "TROOT.h"
#include "TFile.h"
#include "TTree.h"
#include "TChain.h"
#include "TBranch.h"

int main(int argc, char **argv)
{
  // split by ','
  std::string argStr = argv[1];
  std::vector<std::string> fileList;
  for (size_t i=0,n; i <= argStr.length(); i=n+1)
    {
      n = argStr.find_first_of(',',i);
```

```
      if (n == std::string::npos)
        n = argStr.length();
      std::string tmp = argStr.substr(i,n-i);
      fileList.push_back(tmp);
    }

  // open input files
  TChain fChain("CollectionTree");
  for (unsigned int iFile=0; iFile<fileList.size(); ++iFile)
    {
      std::cout << "open " << fileList[iFile].c_str() << std::endl;
      fChain.Add(fileList[iFile].c_str());
    }

  Int_t           EventNumber;
  TBranch        *b_EventNumber;
  fChain.SetBranchAddress("EventNumber", &EventNumber, &b_EventNumber);

  // main loop
  Long64_t nentries = fChain.GetEntriesFast();
  for (Long64_t jentry=0; jentry<nentries;jentry++)
    {
      Long64_t ientry = fChain.LoadTree(jentry);
      if (ientry < 0)
        break;
      fChain.GetEntry(jentry);

      std::cout << EventNumber << std::endl;
    }
}
```

Makefile could be

```
$ cat Makefile

ROOTCFLAGS    = $(shell root-config --cflags)
ROOTLIBS      = $(shell root-config --libs)
ROOTGLIBS     = $(shell root-config --glibs)

CXX           = g++
CXXFLAGS      =-I$(ROOTSYS)/include -O -Wall -fPIC
LD            = g++
LDFLAGS       = -g
SOFLAGS       = -shared

CXXFLAGS     += $(ROOTCFLAGS)
LIBS          = $(ROOTLIBS)
GLIBS         = $(ROOTGLIBS)

OBJS          = cpptest.o

cpptest: $(OBJS)
        $(CXX) -o $@ $(OBJS) $(CXXFLAGS) $(LIBS)

# suffix rule
.cc.o:
        $(CXX) -c $(CXXFLAGS) $(GDBFLAGS) $<

# clean
clean:
        rm -f *~ *.o *.o~ core
```

Also you need a wrapper script which compiles source files.

```
$ cat cpptestmake.sh
```

```
#!/bin/bash
make
```

Now, you can submit the job

```
$ ls
cpptest.cc
cpptestmake.sh
Makefile
...
$ prun --exec "cpptest %IN" --bexec "cpptestmake.sh" --athenaTag=14.2.24 --inDS ...
```

buildGen job executes the argument of the `--bexec` option to produce binary files, and then runGen jobs run on top of the binary files. So each runGen doesn't have to compile.

BTW, you may realize that the above can be done without the wrapper script since it essentially invokes only the make command.

```
$ prun --exec "cpptest %IN" --bexec "make" --athenaTag=14.2.24 --inDS ...
```

In this case, buildGen directly executes make . You can execute multiple commands in `--bexec` (and also in `--exec`) by using ;. For example,

```
$ prun --bexec "make clean; make" --exec ...
```

Run Python job

This is an example to execute a completely general job. Here is a python script.

```
$ cat purepython.py

import sys
print sys.argv
f = open('out.dat','w')
f.write('hello')
f.close()
sys.exit(0)
```

Then

```
$ prun --exec "python purepython.py %IN" --inDS ...
```

Python is always available on remote WNs. So you don't need to setup Athena on WNs.

Run jobs without input files

If `--inDS` is not specified, `--nJobs` (default=1) subjobs will be instantiated. Perhaps you may set random seeds using `%RNDM` in the `--exec` option. e.g,

```
$ prun --exec "somescript %RNDM:123 %RNDM:456" --outDS user...
```

where %RNDM:basenumber (e.g., %RNDM:100) will be incremented per sub-job.

Run user containers jobs

You can use prun to run standalone containers by adding --containerImage option. A basic "Hello World!" works like that

```
$ prun --containerImage docker://alpine --exec "echo 'Hello World\!'" --outDS user.$RUCIO_ACCOUNT
```

If your image is unpacked in CVMFS you can still use it just replace the docker url with the path to the image. This has the advantage of avoiding downloading the image and building it on the WN for each pilot.

```
$ prun --containerImage /cvmfs/unpacked.cern.ch/registry.hub.docker.com/atlasml/ml-base:latest --
--outDS user.$RUCIO_ACCOUNT.test
```

If you require IO the prun options will continue to work. If you are accessing root files you don't need to worry about copying to scratch the system will handle direct IO from the grid storage but if you are not using root files you need to stage the input files on disk. Typically ML users will need these options `--forceStaged` and `--forceStagedSecondary`

```
$ prun --containerImage docker://atlasml/ml-base --exec "my ML command" --outputs my-output-file.
--outDS user.$RUCIO_ACCOUNT.test --inDS my-input-ds.h5 --forceStaged
```

FAQ

Use Athena Runtime on remote WNs

Sometimes you may need Athena Runtime on remote WNs. e.g.,

```
$ prun --athenaTag=14.2.24 ...
```

This will setup Athena-14.2.24 runtime on remote WNs. The syntax of `--athenaTag` may be familiar to Athena users. If you want to use caches, `--athenaTag=14.2.24.3,AtlasProduction`, for example.

Archive output files

When you have a lot of output files on WN, you may want to archive them.

```
$ prun --outputs "abc.data,JiveXML_*.xml" ...
```

this will produce `DatasetName_SerialNumber_abc.data` and `DatasetName_SerialNumber_JiveXML_XYZ.xml.tgz`. The latter will contain all JiveXML_*.xml files. You need " or \ to disable shell-globing when the wild-card is used.

Send jobs to a particular site

prun automatically chooses an appropriate site by using information about dataset location, site occupancy, and user's VOMS FQAN. But users can send jobs to a particular cloud/site using `--cloud=` or `--site` option. e.g.,

```
$ prun --cloud FR ...
$ prun --site TRIUMF ...
```

Use/skip selected files in the input dataset

prun has the `--match` option which allows user to choose files matching a given pattern. The argument is a comma-separated string.

```
$ prun --match "*AOD*" ...
$ prun --match "*r123*,*r345*" ...
```

If you need to skip specific files, use the `--antiMatch` option.

How to use multiple input datasets in a single job

If all you want to do is to call pathena/prun once and have it run over many input datasets, you can simply do this:

```
prun --inDS dsA,dsB,dsC,dsD ...
```

This will still only access a single dataset per job (on the worker node), but the entire jobset will process multiple input datasets.

However, if you want to access multiple datasets within each subjob, you need something more complicated:

prun has the `--secondaryDSs` option to specify secondary dataset names. The argument is a comma-separated list of `StreamName:nFilesPerJob:DatasetName[:MatchingPattern[:nSkipFiles]]` where

StreamName
: the name of stream in the `--exec` argument

nFilesPerJob
: the number of files per subjob

DatasetName
: the dataset name

MatchingPattern
: to use files matching a pattern (can be omitted)

nSkipFiles
: to skip files (can be omitted)

For example,

```
$ prun --exec "test %IN %IN2 %IN3" --secondaryDSs IN2:3:mc08.106017.gg2WW0240_JIMMY_WW_taunutaunu
```

When a subjob runs on a WN, `%IN2` and `%IN3` will be replaced with e.g. `AOD.029946._00001.pool.root.1,AOD.029946._00002.pool.root.1,AOD.029946._00003.pool.root.1` and `AOD.025888._00001.pool.root.1,AOD.025888._00002.pool.root.1`, respectively. Note that `IN` is replaced with files in `--inDS` and the job is split based on the number of files in `--inDS` and `--nFilesPerJob` even if `--secondaryDSs` is used.

Compile Athena packages

If the `--useAthenaPackages` option is set, Athena packages in your work area (`$TestArea`) will be sent to WN to compile. One can still use the `--bexec` option if additional build step is required. In this case, the argument of the `--bexec` option will be executed after Athena packages compile and environment variables are configured to use binary/python files in InstallArea on the WN. So you can link your program against Athena libraries in InstallArea if needed. Also, the argument of the `--exec` option will be executed after Athena runtime is setup to use the binary/python files on the WN. You need to setup Athena before running prun. e.g.,

```
$ source setup.sh -tag=15.3.0,32,setup
$ source .../etc/panda/panda_setup.sh
$ prun --useAthenaPackages --exec ...
```

Note that prun gathers only files with some special extensions from the current directory if `--useAthenaPackages` is used. So `--extFile` is required if you want to send other files to WNs.

Merge ROOT files

You may use hadd to merge ROOT files locally. You can do that on the grid as well. e.g.,

```
$ prun --exec "hadd newoutput.root \`echo %IN | sed 's/,/ /g'\`" \
--outputs newoutput.root \
--athenaTag=15.3.0 \
--inDS user09.CalDevens.Pythia_ZmumuJet.recon.AAN.v15300000 \
--outDS ...
```

where `sed` just replaces commas with white-spaces because hadd takes a list of source files separated by white-spaces instead of commas.

Of course, you can run more complicated macro to merge ROOT files if you want (see CINT example).

Merge output files

You can also use the `--mergeOutput` option when running prun to merge output files on the fly. By default, Merging_trf.py is used for pool files, hadd is used for hist and ntuple, gzip is used for log and text. The `--mergeScript` option allows you to use your own script/executable to merge output files if necessary. E.g.,

```
prun ... --mergeOutput --mergeScript="your_merger.py -o %OUT -i %IN"
```

At the runtime, the `%IN` will be replaced by a list of input file names separated by a comma and `%OUT` replaced by the output file name. If your job produces two output files, `your_merger.py` is executed for each output file separately, i.e.

```
your_merger.py -o out1.root -i in1.root,in2.root,in3.root,...
your_merger.py -o out1.xml -i in1.xml,in2.xml,in3.xml,...
```

so that `your_merger.py` needs to be able to work either output type.

Input files are unavailable in the 'current' directory on the WN at some sites

Some sites are configured to read input files using remote IO such as dcap, rfio and xrootd. In this case, `%IN` is converted to a list of TURLs instead of LFNs, e.g., `dacp://path1/file1,dcap://path2/file2`. As long as your application takes `%IN` via stdin/file and opens input files by using TFile, the difference between TURL and LFN is transparent. In other words, if you search the current directory for input files ignoring `%IN`, your jobs will fail at those sites.

Running pyROOT script at remote IO sites

As mentioned in the above section, as long as you parse `%IN` and open input files via TFile, you don't need to change your pyROOT script for remote IO sites. However, it is known that pyROOT captures command-line arguments when loading ROOT modules (see a roottalk topic). e.g., when you execute

```
$ python yourscript.py input1.root,input2.root
```

"input1.root,input2.root" is given to ROOT TApplication too although you intended to give the argument to yourscript.py only. In many cases, this doesn't cause a critical problem. However, when you run your job at remote IO sites, ROOT hangs up due to a very long argument (see another roottalk topic) and the job is eventually killed by the pilot or the batch system. The solution is to put "-" between your script and the argument, which prevents pyROOT from parsing command-line arguments . i.e.,

```
$ python yourscript.py - input1.root,input2.root
```

This means that you submit the job using

```
--exec "python yourscript.py - %IN"
```

instead of

```
--exec "python yourscript.py %IN"
```

You may need to modify yourscript.py accordingly. e.g., from

```
inputFiles = sys.argv[1].split(',')
```

to

```
inputFiles = sys.argv[-1].split(',')
```

Or you may put `ROOT.PyConfig.IgnoreCommandLineOptions=True` to your script if you use ROOT-5.24 or higher.

Using so many input files in a single subjob

The maximum number of input files per subjob is 200 by default. This protection is required since too many input files result in a too long command-line argument and some sites cannot run such jobs due to the length limit in a command-line shell. You can relax the constraint using the `--maxNFilesPerJob` option. Generally it is better to use this option together with the `--writeInputToTxt` option, so that your job can run even if the remote site has the limitation on the length of the command-line argument. The `--writeInputToTxt` option takes a comma separated list of `StreamName:FileName`. e.g., when `IN:input.txt`, the list of input files is written to input.txt. Essentially that is the same as `'echo %IN > input.txt'` in the `--exec` option, however, the point is that the list is directly written to the file (i.e., not through the shell) and thus it avoids the limitation on the argument length. If you have multiple input streams using `--secondaryDSs`, `--writeInputToTxt` would be something like `IN:input.txt,IN2:input2.txt,IN3:input3.txt`. Note that you would need to modify your application to read the list of input files from a file instead of stdin and `--exec` would be something like `"run your_app"` instead of `"echo %IN | run your_app"` or `"echo %IN > input.txt; run your_app"`.

Using custom root classes with SELinux

On certain sites (for example: ANALY_CERN) the compiling of custom root class libraries will succeed, but when trying to load said library, an error similar to this is produced: `"cannot restore segment prot after reloc: Permission denied"`. This error is caused by SELinux, which is enabled by default in SLC5. In order to fix this, you need to build your code using the -fPIC compiler flag. If recompiling is not an option, a workaround is to disable the SELinux check for relocations on your library alone, by doing `"chcon -t textrel_shlib_t"` on your .so-library file (you do not need root privileges to do this).

Use RootCore

The `--useRootCore` option is available since 0.3.51. You can use this option once you setup RootCore runtime like

```
$ cd RootCore
$ ./configure
$ source scripts/setup.sh
```

The option takes care of three things. First, it copies all packages into the current working directory before sending them to WN. This means that you don't need to have the packages in subdirectories of the working directory. Second, it compiles all packages in the build step. Third, it sets up the environment variables, updates paths in the package list and links in the run step before running your application. Note that you still need `--athenaTag` or `--useAthenaPackages` to have ROOT runtime on WN.

Error when --nGBPerJob is used

The option --nGBPerJob is useful when the dataset has files with different sizes but prun submission can fail if the value is less than the files. For example, if you define --nGBPerJob=5, the error message printed is:

```
ERROR : A subjob has 1 input files and requires 5151MB of disk space . It must be less than 5120M
```

So, several options can be used, or the --nGBPerJob should be higher or put as --nGBPerJob=MAX. To check the file size, the panda monitor can be used: check file size

Adding job metadata

Users can add metadata to each job in PanDA. If jobs produce json files `userJobMetadata.json` in the run directory it is uploaded to PanDA and you can see it in pandamon or pbook. This is typically useful if jobs have very small outputs, such as hyperparameter optimization for machine learning where each job could produce only one value. Users can get results directly from PanDA rather than uploading/downloading small files to/from storages. Note that the size of each metadata must be less than 1MB and metadata are available only for successfully finished jobs.

First you need to change your application to produce a json file, e.g.

```
$ cat a.py
# do something useful and then
import json
json.dump({'aaaaaa':'bbbbbb', 'ccc':[1,2,5]}, open('userJobMetadata.json', 'w'))
```

Then submit tasks as usual. You don't need any special option. E.g.,

```
$ prun --exec 'python a.py' --outDS user.hage.`uuidgen`
```

Once jobs have successfully finished you can see metadata in pandamon, e.g. link which shows

```
Job metadata
{
    "reportVersion": "1.0.0",
    "user_job_metadata": {
        "aaaaaa": "bbbbbb",
        "ccc": [
            1,
            2,
            5
        ]
    }
}
```

or in json dump

```
https://bigpanda.cern.ch/jobs/?jeditaskid=<taskID>&fields=metastruct&json
```

or in pbook

```
$ pbook
>>> getUserJobMetadata(taskID, output_json_filename)
```

or in python session

```
$ python
>>> from pandatools import PBookCore
>>> x = PBookCore.PBookCore()
>>> x.sync()
>>> x.getUserJobMetadata(taskID, output_json_filename)
```

Contact Email Address: `hn-atlas-dist-analysis-help@cern.ch`

Responsible:

Never reviewed

This topic: PanDA > PandaRun
Topic revision: r98 - 2020-01-08 - AlessandraForti