Python based applications on Red Storm

Porting a Python based application to the Lightweight Kernel

John Greenfield, *SNLA and* **Daniel Sands**, *SNLA*

ABSTRACT: It is desirable to be able to run analysis and validation applications as pre and post-processing stages on large systems like Red Storm. Many analysis and validation applications are based on Python and porting Python based applications to the Lightweight Kernel presents some challenges. The example of porting the Data Services Toolkit (DSTK) Python-based application to Red Storm is used to illustrate the challenges and their solutions.

KEYWORDS: Python, Porting, Red Storm, XT3, Catamount

1. Introduction

It is desirable to be able to run analysis and validation applications as pre and post-processing stages on large systems like Red Storm. Often these applications are specific to a given problem and it is therefore desirable to be able to run these applications without imposing a great deal of additional effort on the user. Many analysis and validation applications are based on Python and porting Python based applications to the Lightweight Kernel presents some challenges.

The first motivation for porting Python to Red Storm was the desire to be able to use the Data Services Toolkit (DSTK) there. DSTK is a Sandia developed application that uses a Python interface to provide access to a set of post processing tools for the Exodus simulation file format. Since the first porting, additional uses have arisen for Python including the most recent requirement of adding an externally developed Python-based simulation code to the available tools on Red Storm.

The challenges of porting Python to the light weight kernel include the lack of dynamic loading, cross-compilation requirements, and a difference between service and compute nodes. The limited kernel does not allow dynamic loading, but the Python build process assumes that it will always be available. Cross-compilation requirements involve overriding the assumption that the build process environment will match the Python environment on the compute nodes.

This paper is based on work that was done to port Python to lightweight kernels, Red Storm's Catamount nodes in this case. This is based on Python 2.5. The example of porting the Data Services Toolkit (DSTK) Python-based application to Red Storm is used to illustrate the challenges and their solutions.

2. Difficulties and their solutions

Lack of dynamic loading.

Python is designed to load libraries and modules dynamically. The limited kernel does not allow this, therefore, Python must be modified to statically link libraries and modules.

Issues with Cross Compiling.

The Python build system does not handle cross-compiling well. For the limited kernel, not only is cross-compiling required, but launching of tasks is done via yod, which imposes further constraints on the way the build works. Therefore the build system must be modified and wrapper functions provided to handle return conditions that the build system is expecting. Since the build process for many third-party Python modules uses Python as a driver for their build, it is necessary to build a version of Python that runs on the service nodes but uses the flags and methods necessary for the compute nodes.

Other difficulties.

Some other adjustments need to be made to the Python build system, such as setting certain system variables, eliminating functions related to user and group information gathering and disabling explicit large file support (to avoid a bug in the 64-bit libraries).

There are some system variable settings on Red Storm that do not fit well with what the Python build is expecting them to be. In particular NGROUP_MAX and TMP_MAX are set to 0, but the Python build is expecting these to be non-zero. Since the Python code has built in acceptable defaults that are used if these variables are undefined, the simple solution is to add instructions to the Python configuration header file that undefines these variables before the build starts, so that the defaults will be used. Functions related to user and group information gathering such as getpwuid and getpwent require features which are missing in the catamount environment, so these functions are removed from the Python build. Finally, the Python build is edited so that Large File system support is explicitly disabled. If this is not done the Python build will call the xxxx64 versions of file IO functions instead of the standard versions. This should be avoided at this time since the 64-bit versions contain bugs, and the standard versions are bug-free and adequate for the Python requirements.

Parallel Performance Enhancements.

Python is designed to load modules from disk at run time. This presents a scalability issue for executing Python on large numbers of nodes simultaneously. In this case all the nodes would be requesting the same file transfer from the service nodes. In order to avoid this, the Python module loader is changed to allow only a single rank of nodes to load the module from disk and then broadcast the files to the other nodes via MPI. This does require that the Python code to be executed has all nodes load the same modules at the same time. This is generally not too much of a restriction for parallel pre- or post-processing Python codes.

3. Step by step installation

3.1 Basic Steps

The basic steps:

- Build and install the Python module builder (service node Python)
- Build and install basic Python itself
- Build the Third-party modules
- Add the modules to the Python static link list
- Rebuild Python with the final modules list.

3.2 Build and install the Python Module Builder

The Python module builder is a Python executable that runs on the service nodes but uses the flags necessary to build modules for inclusion in the Catamount Python. It will be installed in the same directory space as the Catamount version. Basically it's just a native Python implementation on the service node with adjusted flag settings. Therefore the steps to build this are simple:

```
➢module unload PrgEnv-pgi
➢./configure –prefix=<install path>
MACHDEP=redstorm ac_sys_system=Linux
➢make
➢make install
➢cd <install path>/bin
➢mv python python.sn
```

This creates a standard Python environment which is also the basis for the Catamount Python. Notice the MACHDEP-redstorm line above. This overrides the sys.platform variable, which would normally default to "Linux", setting it to "redstorm". This enables codes that use the variable to decide if or how to run, or what features to make available.

3.3 Build and Install Basic Python

Restore Catamount environment

Building basic Python requires some changes. First, restore the Catamount build environment and return to the Python source:

```
➢module load PrgEnv-pgi
➢cd <python source path>
```

Address cross-compiling issues

Next, it is necessary to address the deficiencies in the Python build system, which was not designed around cross-compiling, and to work around the limitations of Catamount. First the cross-compiling issue. In addition to compiling on the service nodes for execution on the compute nodes, the "yod" loader is used. Instead of specifying a cross-compile environment, which the configure script almost, but doesn't quite, support and which would not support the "yod" loader, some macros are added to tell configure how to run its compiled code. This is done by creating a file called aclocal.m4. The contents of this file rewrite the macro for running compiled programs. Once this file is in place, the program "autoconfig" is run to transform "configure.in" and "aclocal.m4" into a new "configure" script.

Contents of aclocal.m4:

```
    ➢# _AC_RUN_IFELSE(PROGRAM, [ACTION-IF-TRUE],
[ACTION-IF-FALSE])
    ➢# ---------------------------------------
    ➢# Compile, link, and run.
    ➢# This macro can be used during the
    ➢# selection of a compiler.
    ➢# We also remove conftest.o as if the
    ➢# compilation fails, some
    ➢# compilers don't remove it.  We remove
    ➢# gmon.out and bb.out, which may be created
    ➢# during the run if the program is built
    ➢# with profiling support.
    ➢m4_define([_AC_RUN_IFELSE],
    ➢[m4_ifvaln([$1],
[AC_LANG_CONFTEST([$1])])dnl
    ➢rm -f conftest$ac_exeext
    ➢AS_IF([AC_TRY_EVAL(ac_link writeresult.o)
&& AC_TRY_COMMAND(yod ./conftest$ac_exeext) &&
ac_status=`cat ./conftest$ac_exeext.result` &&
echo "Got result code "$ac_status && (exit
$ac_status)],
    ➢[$2],
    ➢[echo "$as_me: program exited with status
$ac_status" >&AS_MESSAGE_LOG_FD
    ➢_AC_MSG_LOG_CONFTEST
    ➢m4_ifvaln([$3],
    ➢    [( exit $ac_status )
    ➢$3])dnl])[ ]dnl
    ➢rm -f core *.core gmon.out bb.out
conftest$ac_exeext conftest.$ac_objext
m4_ifval([$1],
    ➢[conftest.$ac_ext])[]dnl
    ➢])# _AC_RUN_IFELSE
    ➢
    ➢# _AC_COMPILER_EXEEXT_WORKS
    ➢# -------------------------
    ➢m4_define([_AC_COMPILER_EXEEXT_WORKS],
    ➢[# Check the compiler produces executables
    ➢# we can run.  If not,
    ➢# either the compiler is broken, or we
    ➢# cross compile.
    ➢AC_MSG_CHECKING([if the _AC_LANG compiler
works])
    ➢# FIXME: These cross compiler hacks should
    ➢# be removed for Autoconf 3.0
    ➢# If not cross compiling, check that we can
    ➢# run a simple program.
    ➢
    ➢if test "$cross_compiling" != yes; then
    ➢if AC_TRY_COMMAND([yod ./$ac_file]); then
    ➢cross_compiling=no
    ➢else
    ➢if test "$cross_compiling" = maybe; then
    ➢  cross_compiling=yes
    ➢else
    ➢  AC_MSG_FAILURE([cannot run _AC_LANG
compiled programs.  If you meant to cross
compile, use `--host'.])
    ➢fi
    ➢fi
    ➢fi
    ➢AC_MSG_RESULT([yes])
    ➢])# _AC_COMPILER_EXEEXT_WORKS
```

Address Yod return code issue

There is one more requirement to address. Several of configure's tests rely on a program's return code. Yod, which has multiple process launches in mind, only returns whether it was successful in running the program on all of the requested nodes, not the program result code itself. The file "writeresult.c" works around this by providing an exit hook that writes the return result to <program name>.result upon exit. The macro in aclocal.m4 reads the ".result" file and uses its content (a single number) as the final result for a test.

```
    ➢writeresult.c:
    ➢#include <sys/types.h>
    ➢#include <sys/stat.h>
    ➢#include <fcntl.h>
    ➢#include <unistd.h>
    ➢#include <stdlib.h>
    ➢extern char* __progname_full;
    ➢extern char* __progname;
    ➢
    ➢void myexit(int code, void *arg);
    ➢
    ➢void __attribute((constructor))
writeexit(){
    ➢on_exit(myexit, NULL);
    ➢}
    ➢
    ➢void myexit(int code, void *arg) {
    ➢int fd;
    ➢char ofile[2048];
    ➢
    ➢printf( "Exiting with code %d\n", code); /*
for debugging */
    ➢sprintf( ofile, "%s.result",
__progname_full);
    ➢
    ➢fd = creat( ofile, 0700);
    ➢sprintf( ofile, "%d", code);
    ➢write( fd, ofile, strlen( ofile));
    ➢close(fd);
    ➢}
```

This program must be compiled with GCC since the "constructor" attribute is a GCC extension to C:

```
    ➢gcc -c writeresult.c
```

Address dynamic loading issues

Compile the replacement dynamic loader.

```
➢dynload_redstorm.c
➢/* This module provides simulation of
➢   dynamic loading for statically linked
➢   modules on Red Storm */
➢
➢#include "Python.h"
➢#include "importdl.h"
➢
➢
```

```
➢const struct filedescr
➢_PyImport_DynLoadFiletab[] = {
➢       {".a", "rb", C_EXTENSION},
➢       {0, 0}
➢};
➢
➢/* This table is defined in config.c:
➢*/ extern struct _inittab
➢_PyImport_Inittab[];
➢
➢/* Basically, just search through the
➢list of compiled-in init functions */
➢dl_funcptr
➢_PyImport_GetDynLoadFunc(const char
➢*fqname, const char *shortname,
➢const char *pathname, FILE *fp)
➢{
➢       struct _inittab *tab =
➢               _PyImport_Inittab;
➢       while (tab->name && strcmp(
➢               shortname, tab->name))
➢               tab++;
➢
➢       return tab->initfunc;
➢}
```

This does not actually implement dynamic loading since all modules are statically linked into the final Python executable. This is only here to aid in namespace resolution of modules which appear in sub-packages. When a package is imported within Python, say foo.bar._mumble, Python will search for _mumble.py within the foo/bar package space. Failing that, it will try the chosen dynamic loader to find _mumble.so. If the loader finds and loads the library, it will return the pointer to the module's init function. Dynload_redstorm applies this behaviour to static libraries by looking for _mumble.a, and if it finds this it will return _mumble's compiled-in init function.

Makefile changes

Finally, some simple modifications are made to Makefile.pre.in. The first change is to put "yod" in front of the line that runs $(PGEN). This change is not actually necessary since the Python build will continue even if this stage fails, but for completeness this change is made. During the build process the Python build generates and runs a grammar parser which (re)writes a part of Python's parsing code; this change allows that process to run correctly. The second change is to remove the "sharedmods" dependency from the "all" target. There are no shared modules in this build, and the build fails when it tries to build them. The third change is to add "yod" to the install targets that run $(BUILDPYTHON): scriptsinstall and libinstall. Finally, remove the sharedinstall dependency from the altinstall target.

Now run the configure script.

```
    ➢setenv ac_cv_file__dev_ptmx no
    ➢# Override detection to prevent getty build
    ➢./configure --without-gcc --prefix=<install
path> BASECFLAGS=-c9x
DYNLOADFILE=dynload_redstorm.o CXX=CC
MACHDEP=redstorm SO=.a
```

Address system variable problems

This should succeed in producing a makefile. However there is now an issue with Catamount to work around. Several key system-defined limits are supplied as 0. The two that are of concern here are NGROUPS_MAX and TMP_MAX. NGROUPS_MAX must be non-zero because the Python POSIX module (presented to the end user as the "os" module, which provides an OS independent interface to OS-dependent functions) defines an array of NGROUPS_MAX size for one of its functions. TMP_MAX must be non-zero because Python's home-rolled mkstemp function uses it as the maximum number of iterations for randomly creating unique temporary file names. Fortunately, Python uses defaults for both if left undefined, so two lines are added to the end of pyconfig.h, outside of the #ifdef / #endif wrapper.

```
    ➢#undef NGROUPS_MAX
    ➢#undef TMP_MAX
```

Address large file support issues

In addition, it is necessary to override the Python build's attempts at large file support. Certain file functions, most notably readdir, do not work correctly when the 64-bit version is called. Since file sizes are already 64 bits, the *64 versions are not needed. These defines should be undefined:

```
    ➢#undef _LARGEFILE_SOURCE
    ➢#undef _FILE_OFFSET_BITS
```

Parallel performance

Another change, while not technically necessary, should help to ease the load on the service node. Python loads external .py modules from disk while running. If over a thousand nodes were to read the same .py file at once, as is likely for a large MPI job, this could strain the system. Fortunately, all external module loading is handled by a single Python module, called the data marshal. So Modules/marshal.c is modified so that only one rank will actually load the data and communicate it to the other ranks through MPI_Bcast calls. The one caveat here is that all Python scripts must import the same modules at the same times, or hangs and possibly confusion will abound. Note that this change is

independent of, and does not require, the pyMPI module discussed later.

Distutils modifications

The Python distutils package is used by many Python modules as a build framework. Since shared objects are not available, the module must be customized to handle static libraries. The following function is added to the unixccompiler class in Lib/distutils/unixccompiler.py:

```
➢def link_shared_object (self,
                         objects,
                         output_filename,
                         output_dir=None,
                         libraries=None,
                         library_dirs=None,

➢runtime_library_dirs=None,
                         export_symbols=None,
                         debug=0,
                         extra_preargs=None,
                         extra_postargs=None,
                         build_temp=None,
                         target_lang=None):
➢  if output_dir is None:
➢     (output_dir, output_filename) =
os.path.split( output_filename)
➢
➢  linkline = "%s %s" % (output_filename[:-
2], output_filename)
➢
➢  for l in library_dirs:
➢     linkline = linkline + " -L" + l
➢
➢  for l in libraries:
➢     linkline = linkline + " -l" + l
➢
➢  old_fmt = self.static_lib_format
➢  self.static_lib_format = "%s%.0s"
➢  self.create_static_lib(objects,
                          output_filename,
                          output_dir, debug,
                          target_lang)
➢ self.static_lib_format = old_fmt
➢
➢ print "Append to Setup: ", linkline
```

This causes a module build to create <module>.a, and also to print a line that is suitable to be included in Modules/Setup. Then simply add that line to Modules/Setup.

Add to static modules list

The last change for basic Python is to add a list of Python native modules to the list of static inclusion. This is done in the file Modules/Setup. It has a simple format for each line: <module name> <list of included files defines and link flags>. Some modules are already statically bound, but many more need to be added. These are already listed, so it is necessary only to uncomment them. The current list of static Python modules includes:

- array
  - # array objects
- cmath
  - # complex math library functions
- math
  - # math library functions, e.g. sin()
- _struct
  - # binary structure packing/unpacking
- time timemodule.c
  - # time operations and variables
- operator
  - # operator.add() and similar goodies
- _weakref
  - # basic weak reference support
- _random
  - # Random number generator
- collections
  - # Container types
- itertools
  - # Functions creating iterators for efficient looping
- Strop
  - # String manipulations
- unicodedata
  - # static Unicode character database
- binascii
  - # Helper module for various ASCII-encoders
- cStringIO
  - # C implementation of string-based IO emulation
- cPickle
  - # C implementation of object pickling

Now complete the build

```
➢make
➢make install
```

3.5 Build the Third-Party Modules

The next step is to build any desired third-party modules. This can be done easily for modules which use the distutils framework, which is typically run through setup.py.

```
➢cd <module source path>
➢<install path>/python.sn setup.py build
➢<install path>/python.sn setup.py install
```

This of course assumes you have write privileges to the installed Python directory, which you will need anyway to link Python. In fact, the next step is to add the

built modules to the statically-linked list in <python source>/Modules/Setup:

```
➢mymodule <install path>/lib/python2.5/site-packages/mymodule.a
```

3.6 Add Modules to the Python Static Link List

One third-party module that is often needed is PyMPI, with some customization to build mpi as a module instead of building a new executable. Prior to running "configure", add pyMPI_softload.c to the list of objects in Makefile.in, and remove the redefinition of Py_GetVersion in pyMPI_sysmods.c since it conflicts with Python main's definition.

```
[skip unchanged parts]
➢pyMPI_user_startup.$(OBJEXT) pyMPI_util.$(OBJEXT) \
➢pyMPI_softload.$(OBJEXT)
```

Once these changes are made, run the build and then manually install.

```
➢cd pyMPI-2.4b4
➢./configure --with-python=<python install path>/bin/python.sn
➢make
➢cp libpyMPI.a <python install path>/lib/python2.5/site-packages/mpi.a
```

This will also build a pyMPI executable, but since Python will need to be recompiled whenever one or more modules are added, we only use the module.

Now add pyMPI to the Setup file as discussed above.

```
➢mpi <python install path>/lib/python2.5/site-packages/mpi.a
```

Other third-party modules, such as the module for the Data Services Toolkit, can be added in a similar fashion.

3.7 Rebuild Python with the final modules list

Once all modules have been built and added, rerun make:

```
➢cd <python source path>
➢make
➢make install
```

The final result is a Catamount runnable Python

```
➢yod -sz 4 <install path>/bin/python HelloWorld.py
```

4. Conclusion

4.1 Performance

Performance seems to be mostly identical to Python running on a standard linux system. This makes sense since the code itself is the same. Dynamic loading of modules is the only significant difference between the architectures. The code is not sharing any CPU time with other processes. Only if it's small I/O intensive would there be a noticeable drop in performance. The memory footprint may be larger than for a dynamic executable, for code that only uses a small set of the Python functionality. But it may be comparable or even smaller for codes that use more of the Python system, since there are not only the modules loaded, but the dynamic library and structures related to handling that part also.

4.2 Extension to other codes

What works for porting Python should work for other scripting systems in a similar fashion. The solutions for lack of dynamic loading and parallel efficiency should be relatively easy to translate to other scripting systems and applications. The particular system variables that were a problem for Python may not be a problem in other applications, but looking for use of parameters related to number of processes, swap size, or resource limits may prove helpful in getting other applications to build.

For a Python-based simulation code that is being ported to Red Storm now, there is an additional requirement. The executable itself must be re-linked prior to each execution since the code has many modules and it is up to the user to decide which modules are necessary for a given run. So a build system is being developed to modify the Modules/Setup file, probably using a dependency analyser; re-run make; and then run the executable. This will likely be done through a Python script which the native Python will execute. Much of this is already present, but more details will become available when the port is finished.

4.3 Future Work

The functionality of DSTK is being reimplemented in the ParaView scientific visualization application. The next task will be to attempt to port all or part of ParaView to Red Storm. The plan is to start work on this porting process next year.

Acknowledgments

The authors would like to thank Rena Haynes and Jim Holten for the development of DSTK, without which we would never have gotten involved in this area. We would also like to thank all of the Red Storm system staff for their assistance in this project. Finally we would like to thank our colleagues in the visualization support teams for their helpful advice and support.

Funding was provided by the Advanced Simulation and Computing (ASC) program's Data Visualization Science (DVS) Program. The work was performed at Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

About the Authors

John Greenfield, PhD is a post-processing and visualization support software engineer for ASAP under contract to Sandia National Laboratories in Albuquerque. He can be reached at Sandia National Laboratories, P.O. Box 5800, MS 0822, Albuquerque, NM, 87185-0822, USA E-Mail: jagreen@sandia.gov. Daniel Sands is a Software Developer for SAIC under contract to Sandia National Laboratories in Albuquerque. Daniel can be reached at Sandia National Laboratories, P.O. Box 5800, MS 0822, Albuquerque, NM, 87185-0822, USA E-mail: dnsands@sandia.gov.