CIS*4450 Term Project

Pylot

Jason Lapp 0652137

03 December 2009

Table of Contents

Pylot....................1
- Pylot Proposal....................2
  - Architecture....................2
  - Features....................2
  - Limitations....................3
- Experience....................3
  - Implementing Pylot....................3
  - Failure to Complete Demonstration Program....................3
  - Future Development....................4
- Running Pylot....................4
- Proposed Grade....................4

Pylot Proposal

Pylot was intended to be a Python wrapper layer over Pilot. I was to hand in a Python library which exposed Pilot, a series of unit tests to prove that the Python layer behaved correctly, and if there was time, a sample program to demonstrate the library in action. The first two items are complete, the third is not.

Pylot as Implemented

Architecture

The definitions for the functions exposed to Python are written in `pylot.i`. SWIG uses this file to automatically generate the code to bind Python and C. SWIG generates the interop file `pylot.c`, and the Python header `pylot.py`. Because `pylot.py` is automatically generated, I can't make any changes to it, so I created a second Python file `__init__.py`, and placed both of these files into a Python package called `pylot`. On compile, `pylot.c`, `pylot_wrap.c`, and the Pilot source are compiled into a shared object, _pylot.so inside the `pylot` Python package.

So, there is a package `pylot`, which automatically includes a second, hidden module also called `pylot`, which loads a C extension called `_pylot.so`.

I had to compile the Pilot source code into my program so that Python could load the Pilot functions from it. Pilot, as a static library, does not expose any symbol names, and when I imported it in Python it failed. Compiling Pilot as a shared library worked, and _pylot.so would load this library when it was loaded, but I could not get this solution to work on amdi. I made `libpilot.so` a well-known library on my own system by placing it in `/usr/lib`, but I of course don't have write access to this folder on amdi, so this solution would not work.

Features

Pylot exposes every function in `pilot.h` to Python. It also adds a pair of variables, `mpi_rank`, and `mpi_worldsize`, which use raw MPI calls to describe the environment that the program is running in. I added these functions so I could configure the unit tests properly; Pilot doesn't expose the rank of a process except as a return value from `PI_StartAll`, and of course any child processes can't see this return value while they are running.

I removed all format string arguments to the channel I/O functions. Rather than specifying the format of each argument, I allow the user to simply provide a variable-sized list of arguments to the I/O functions, and I use Python's reflection ability to inspect each one and build the format string for Pilot myself. Reading works similarly to writing: there is no need to describe which data type is coming off the channel because Pylot encodes this information along with the data. You simply tell Pylot how many items you want to read off the channel.

`Int`s, float, bools, `string`s, the `None` type, and lists of each of these can be written, read, or broadcast on a channel. You may gather `int`s, `float`s and `bool`s.

I implement Pilot's macro recording of calling functions by decorating the Python functions with a class that grabs the call stack, and passes the previous frame's information to `PI_CallerLine` and `PI_CallerFile`.

Unit tests exist for all the Pylot functions I wrote: configuration, reading, writing, broadcasting, and gathering. I have not yet written unit tests for the other Pilot functions, but most of them are indirectly tested by the existing tests.

Limitations

I don't allow users to send hashtables or classes across channels. Also, because I prefix each channel element with its type, Pylot introduces some extra overhead for each channel write. Rather than check for repeated argument types in a list, I simply prefix each element with it, which adds an extra `char` for each item transmitted across a channel.

Experience

Implementing Pylot

Implementing the Python API and unit tests took much longer than I expected. SWIG was able to wrap most of the Pilot functions, but I was forced to write and debug all of the channel I/O functions myself. While not the most pleasant experience, it was actually kind of neat to attach gdb to three processes at once and watch how they interacted with each other.

SWIG was a great help in creating the Python bindings for the Pilot API. While I did have to write some code by myself, the majority was created by SWIG, and I mostly had to define how arguments were to be converted to and from Python objects: these are all of the `%typemap` declarations in `pylot.i`.

The existing CUnit tests that came with the Pilot source code were extremely helpful in writing my own tests. By examining how those tests set up the Pilot environment, I was able to configure PyUnit similarly. Without the CUnit tests, I wouldn't have understood the point or necessity of Pilot's “bench mode”, which is quite an interesting feature, allowing me to stop and start worker processes as I choose. I realize that the worker processes still exist, and are actually just waiting to synchronize with `PI_MAIN`, but the impression from Pilot is that each of these is a subordinate worker that `PI_MAIN` spawned.

I found this to be a fascinating program to write. It greatly improved my understanding of MPI, and I feel much more confident in my parallel programming abilities.

Failure to Complete Demonstration Program

I'm very disappointed by my failure to complete a demonstration application to show off Pylot in a real environment. While the unit tests were extremely useful in proving that each function does what it says it will, and gives a small example of how to use the API, a real program is much more interesting to look at and play with.
Implementing something that would calculate the MD5 sum of a series of files ought to be simple, and if I had another few days I should be able to have it completed. The problem is implementing the algorithm correctly, and based on my performance on each of the assignments, my ability to implement a parallel algorithm correctly seems fairly impaired.
Getting a skeleton of an application written should not take more than an hour or two. Making it correct, and proving it it correct, would probably take two or three days, and I simply don't have time to do that before the semester ends.

As I explained above, more of my time than I expected was taken by writing the API and the tests. In my original proposal, I said that I would complete a suite of unit tests before writing any application code, and for this reason I have not attempted to write a demo application at all.

Future Development

I am still interested in working with Pylot and MPI in the future. Now that I understand how MPI works, I think it's a fascinating subject, and I'd like to know more about it. I don't know how much time I'll have to work on this in the next semester, so I don't want to make a concrete agreement about anything, but I'm open to continuing to develop this library.
I'm particularly interested in implementing some of the features Ben Kelly mentioned in his presentation: sending byte-code across channels, and an interactive MPI interpreter. These two features would make Pylot an excellent tool to use to expose beginners to parallel programming, because the user can see the effects of the statement as they occur, rather than having to reconstruct the environment from a series of `printf` statements.

Running Pylot

Pylot can be downloaded from my svn repository, under the tag `cis4450-project`.

Pylot can be loaded into the interactive shell using `import pylot`, provided that the pylot package directory is under the working directory, or somewhere in the path.

The unit tests can be run from the tests directory via `mpiexec -np 6 python ./testrunner.py`. The testrunner script will grab every file in the folder that begins with test_, and pull the test methods from each. Altogether there are about 75 tests, which is just enough to cover channel I/O, and, indirectly, Pilot configuration, bench mode, and process and channel creation.

Proposed Grade

I didn't complete my demo application, which is a big disappointment, but I also agreed to knock it off the project if I ran out time. The Python API for Pilot is complete, with all functions behaving correctly. I had plans to attach the `PI_Read` and `PI_Write` functions to the channels objects, but never got around to it. My suite of unit tests does not test every Python wrapped function, but it does test all the code I wrote by hand: PI_Configure, PI_Read, PI_Write, PI_Broadcast, and PI_Gather. The tests also indirectly prove that process, channel and bundle creation work, or else all the tests would fail.
For these reasons, that the majority of what I agreed to complete has been done, with the exception of the demo app, I propose that I receive a grade of 70% on my project.