Processing Radio Astronomy Data from the Arecibo Observatory

John Zollweg
Senior Research Associate
Cornell Center for Advanced Computing
zollweg@cac.cornell.edu
The problem

• One observation session (3 hr) generates about 500 GB (0.5 TB) of data

• Data are very noisy - pulsar signals rarely seen in raw data

• Signals are subject to dispersion - different frequency components arrive at different times.

• Some pulsars are best identified by their periodicity

• Others show up better as single pulses

• Many observations are contaminated by periodic Radio Frequency Interference (RFI)

• Final analysis is done visually -> plots need to be produced automatically
The data

• The telescope has 7 apertures - the signals are stored pairwise interleaved in 4 files.

• Files are limited to 2 GB, so there are 4, 8, or 12 files in a pointing

• Data size for a pointing is typically 4, 8, or 16 GB.

• Raw data are archived in TSM, so they have to be restored before use.

• I typically process a whole day of observations in one batch.
Dispersion

Dynamic spectrum is FFT of raw data.

See that phase of pulses changes with frequency - this is due to dispersion
Dedispersion

• Time of Arrival of frequency channels are adjusted according to a postulated dispersion measure. In our processing we use 1272 different values.
Periodicity search

- A sequence of sharp pulses will have many harmonics.
Processing pipeline

PULSAR SEARCHING

MATCHED FILTERS

THRESHOLD TESTS

SINGLE PULSE CANDIDATES

DEDISPERSE

FFT

HARMONIC SUM

THRESHOLD TESTS

FOLD

CANDIDATE LISTS & PROFILES

RAW DATA

TSM

DATA PROCESSING

CLUSTER

SQL SERVER DATABASE

DATA ACCESS ARCHITECTURE

WEBSERVER

GRIDFTP

CLIENT

RAW DATA ARCHIVE

www.cac.cornell.edu
Pipeline detail
Assembling the Pipeline

• Used Python
  – Cross-platform
  – Available as built-in on linux systems
  – Xml configuration file for pipeline management
  – Easily fork new processes
  – Manage pipes
Configuration file

```xml
<?xml version="1.0" encoding="iso-8859-1"?>

<marshal>
  <dictionary id="i2">
    <string>band_norm</string>
    <string>base_dir</string>
    <string>/home/gfs01/jaz4/Arecibo/pulsar.search.code.2004</string>
    <string>bin_dir</string>
    <string>lambda d: "%s/bin/%s" % (d["base_dir"], d["ostype"])</string>
    <string>bins_input</string>
    <int>6500</int>
    <string>do_dedisp</string>
    <bool>True</bool>
  </dictionary>
</marshal>
```

Usage

```python
import ffxml, ldict
parfile = open(process_dir+os.sep+'defpar.xml')
parmdict = ffxml.load(parfile)
parfile.close()
pm.update(parmdict)
```
Subprocesses

from subprocess import *

outname = "run.out.%s.%d" % (wappno, i)
fd.append(open(outname, 'w'))
if beam<7 or pm["do_dedisp"] or pm["do_dynspec"]:  
    beamproc.append(Popen([sys.executable, 
        pm.workdir+os.sep+'run_cornell_search.py', 
        pm.filename,ntimesamples_s], stdout=fd[i],stderr=PIPE))
......
for i in [0, 1]:
    sys.stderr.write(beamproc[i].communicate()[1])
Managing pipes - 1

Connect to named pipe:

    while True:
        try:
            if os.name is 'nt':
                WaitNamedPipe(pipename, 0)
            else:
                sinfo = os.stat(pipename)
                print "Named pipe %s found" % pipename
                break
        except:
            time.sleep(1)
            fdpipe = os.open(pipename, os.O_RDONLY)
if do_dedisp and do_dynspec:
    (dyinr, dyinw) = os.pipe()
    fbpipe = os.fdopen(fdpipe, "rb", 65536)
    dyin = os.fdopen(dyinr, "rb", 65536)
    dypipe = os.fdopen(dyinw, "wb", 65536)
    (deinr, deinw) = os.pipe()
    dein = os.fdopen(deinr, "rb", 65536)
    depipe = os.fdopen(deinw, "wb", 65536)
elif do_dedisp:
    dein = os.fdopen(fdpipe, "rb", 65536)
elif do_dynspec:
    dyin = os.fdopen(fdpipe, "rb", 65536)
if do_dynspec:
    decimate = Popen([pm["sigproc_bin"]+os.sep+'decimate', '-c', nfdecimate_s, '-t', ntdecimate_s, '-headerless'], stdin=dyin, stdout=PIPE, stderr=decimerr)
    reader = Popen([pm["sigproc_bin"]+os.sep+'reader', '-noindex'], stdin=decimate.stdout, stdout=fileout, stderr=readerr)

if do_dedisp:
    dedisp = Popen([pm['dedisp_bin']+os.sep+'dedisp.float', '-c', nchans_s, '-s', sample_time_s, '-b', bins_input_s, '-f', frequency_s, '-w', bandwidth_s, '-d', ntimesamples, '-o', 'f', '-n', nsubbands_s, '-t', dedisp_type, '-z', tree_shift_s, unflip, band_norm], stdin=dein, stdout=dedispout, stderr=dedisperr)
if do_dedisp and do_dynspec:
    buff2=[]
    while True:
        buff = fbpipe.read(65536)
        if len(buff) > 0:
            buff2.append(buff)
        if len(buff2) >= 64:
            dypipe.write("".join(buff2))
            depipe.write("".join(buff2))
            buff2=[]
            continue
        dypipe.write("".join(buff2))
        depipe.write("".join(buff2))
        break
    dypipe.close()
    depipe.close()
Exploit multiple cores

• OpenMP
  – Insert directives

• Pthreads (or Windows Threads)
  – Use separate threads for I/O
  – Manage threads with semaphores (or Windows events)
OpenMP

#pragma omp parallel for private(delayp, outputp, datap, nf, nt)
for(idm=0; idm<ndm; idm++) {
    delayp = &delays[idm][ns*m];
    outputp = &output[idm][done-tstart];
    for(nf=0;nf<m;nf++) {
        datap = &ddata[nf][delayp[nf]];
        for(nt=tstart;nt<n;nt++) {
            outputp[nt] += datap[nt];
        }
    }
}
void *iothread(void *dummy) {
    int bytes_out_reqt,jt,outpt=0,offsett=0;
    jt = (int)dummy;
    bytes_out_reqt=bytes_out_block;
    while (1) {
        pthread_mutex_lock(&doIO[outpt]);
        pthread_mutex_lock(&IOdone[outpt]);
        offsett+=bytes_out_reqt;
        #pragma omp parallel for private(jt)
        for (jt=0; jt<n_dm_channels_per_subband; jt++) {
            bytes_out = pwrite(out[n_subbands-1], output[outpt][jt], bytes_out_reqt, *ovl[n_subbands-1][jt]);
            *ovl[n_subbands-1][jt] = jt*file_size + offsett;
        }
        IOdon[outpt]=1;
        pthread_mutex_unlock(&IOdone[outpt]);
        outpt = (++outpt)%2;
    }
}
Output buffers

In order to make use of a write thread, there must be two output buffers - while one is being filled, the other is being written - and *vice versa*.

Mutex *doIO* is locked until a buffer is full; output thread waits for it to be unlocked. When a buffer is filled by main thread, it unlocks *doIO* so that write thread can go ahead with writing it.

A second mutex *IODone* is locked by the write thread while it is writing and unlocked when it is done. The main thread doesn’t start filling a buffer unless it can lock (and immediately unlock) that mutex.
Conclusion

• Arecibo Data Processing on Ranger:
  – Multicore processors
  – Large, fast scratch space