Note on Parallel Processing of Spectral Line Data in Obit

W. D. Cotton, October 1, 2008

***Abstract*—This memo describes a prototype implementation of parallel processing of spectral line data. The current test was run on a single, dual core machine but is extend-able to use on a cluster. Dividing the bulk of the processing into two streams on a dual core workstation reduced the processing time for a test case in half relative to the time required for a single stream of processing. This improvement should scale well to a larger number of nodes on a cluster.**

***Index Terms*—Computing efficiency, clusters, interferometry**

I. Introduction

THE data rates for interferometers currently under development (e.g. EVLA, ALMA, LOFAR, MeerKAT) will produce vastly more data than the current generation of interferometers and many cases will require parallel computing and especially clusters. Many of the most compute intensive problems are among the "embarrassingly simple" to run on a parallel system. While embarrassingly simple in principle, these problems involve substantial bookkeeping and data transfer to get the data to where the CPU cycles are available. The following discusses an initial attempt at running one of the "simple" cases, spectral line imaging and deconvolution in a parallel fashion in Obit ([1], http://www.cv.nrao.edu/∼bcotton/Obit.html). This test does not make use of a cluster but, with minor modifications, should be able to make use of Obit's existing cluster interface.

II. Spectral Line Imaging

The problem considered here is that of applying calibration to a spectral line data set and then imaging and deconvolving each image plane to produce a 3 dimensional image with RA and dec on two axes and frequency or velocity on the third. In a single stream processing, the channel images are simply produced one at a time and accumulated into an output cube. This scheme involves a substantial overhead in that the full data set must be accessed for each channel and the calibration process rerun. The processing of each channel is completely independent and is easily separated into a number of processing streams in a parallel system.

In the parallel scheme, there are three logical components, two of which are essentially serial and the third can be parallel. The two serial operations are the initial splitting of the data into a number of data files and the accumulations of partial cubes into the full image cube. The actual imaging of individual image planes can be highly parallel.

National Radio Astronomy Observatory, 520 Edgemont Rd., Charlottesville, VA, 22903 USA email: bcotton@nrao.edu

Manuscript received ; revised

The separation of the data into multiple files is needed to avoid the I/O collisions when multiple processing streams try accessing the main data file at the same time. On a cluster, the data files to be processed can be moved to the node on which they will be processed. The collection of the intermediate image cubes into the final cube must be serial to avoid the write conflicts if this is attempted by multiple threads or processors. Fortunately, the bulk of the work is in the imaging and deconvolution.

The implementation of these three steps in the Obit environment is described in the following.

A. Calibrating and splitting into multiple files

The initial step is to use task SplitCh which calibrates and divides the dataset into a specified number of output files. Thus, a single pass calibration is made through the main data file. On a cluster, the output files would be written on a set of disks visible to the processing nodes. An example script for this phase is in the appendix as Figure 3.

B. Imaging and deconvolution of individual channels

Multiple parallel processing streams are initiated, each given a set of the output files from SplitCh. For each channel to be processed, the data for that channel is extracted from the SplitCh output and then imaged, deconvolved and collected to create partial image cubes corresponding to the channels in the SplitCh files. These partial cubes are left where they are visible to the master, control script.

The Obit distributed computing model is implemented in the python scripting environment ObitTalk and is derived from its predecessor, ParselTongue [2]. In this model, each data directory known to the scripting engine has an associated URL, either local or a remote system such as a node on a cluster. To execute tasks (or remote scripts), the scripting engine checks which machine the data to be operated on resides and executes the task on that machine. This makes use of an xmlrpc protocol interface which operates over local area networks, clusters or wide area networks in addition to single computer applications. Using this remote execution capability, multiple parallel processing streams can be distributed over nodes of a cluster or network.

An example stream python script is in the appendix as Figure 4.

C. Collection into the full cube

After all the processing streams are finished, the partial image cubes are left where the master script can access them.

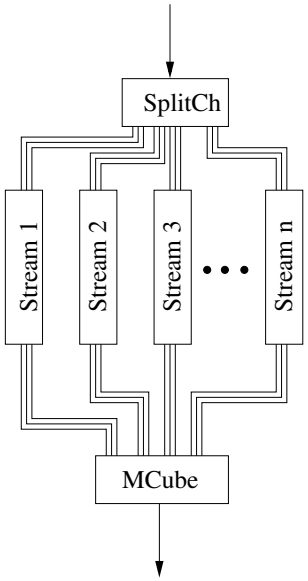

Fig. 1. General data flow and processing configuration. SplitCh divides a spectral line data set into multiple parts which are then fed to multiple, parallel processing streams. MCube accumulates the images into a single cube.

Task MCube then concatenates these partial cube into a total cube. The intermediate data products are then deleted leaving only the initial dataset and the output image cube.

The total processing scheme is represented in Figure 1. An example script for this phase is in the appendix as Figure 5.

III. Timing Test

In order to test this scheme on a real (if small) application, hard-coded scripts were written to perform these operations. These are given in the appendix. The test consisted of imaging 109 channels of a 128 channel VLBA observation of SiO masers in a circumstellar envelope. Each channel was imaged on a 1024 $\times$ 1024 pixel grid and channels containing significant emission were CLEANed. The data to be imaged were calibrated, extracted from the full datasets and stored in 8 intermediate files. Due to the relatively poor uv plane coverage of this multiple snapshot data, the autoWindow feature was used. The data imaged were Stokes I in one transition of a two transition, full Stokes data set which consisted of 168 MByte of "compressed" data.

The processing was on a dual core Linux workstation so the testing used two parallel processing streams, each processing 4 of the SplitCh output files. In order to compare the execution time to more traditional methods, the data were processed in three fashions, 1) single stream (traditional) processing using one thread, 2) single stream processing using two threads and 3) a two stream parallel processing. The timing results are given in Table I and a color coded velocity image of the resulting cube is given in Figure 2.

IV. Discussion

The timing test gave the expected result that dividing the data into two parallel processing paths required half the execution time of the traditional approach. In this case, the alternate parallel scheme, using two threads for the griding and model subtraction netted only minor improvement (8%). With only minor modifications to the scripts, this scheme can be adopted to use ObitTalk's ability to spread processing over a network or nodes of a cluster with subsequent reductions in the computation time. For modest size data sets, NFS cross mounting of disks may be an adequate means of moving data among nodes of a cluster. For larger data sets, data transport time may become a major factor and more efficient means may be necessary.

For a toy problem such as the one presented here, a set of hard coded scripts as were used is practical. For larger or more routine pipeline processing, either more intelligent scripts or, better, building much of the process into a single task will be necessary. The means to communicate among multiple compiled c processes using xmlrpc currently exists in Obit and a test proof of concept exists.

Imaging of spectral line datasets is probably the easiest of the large computing problems to break into parallel streams;

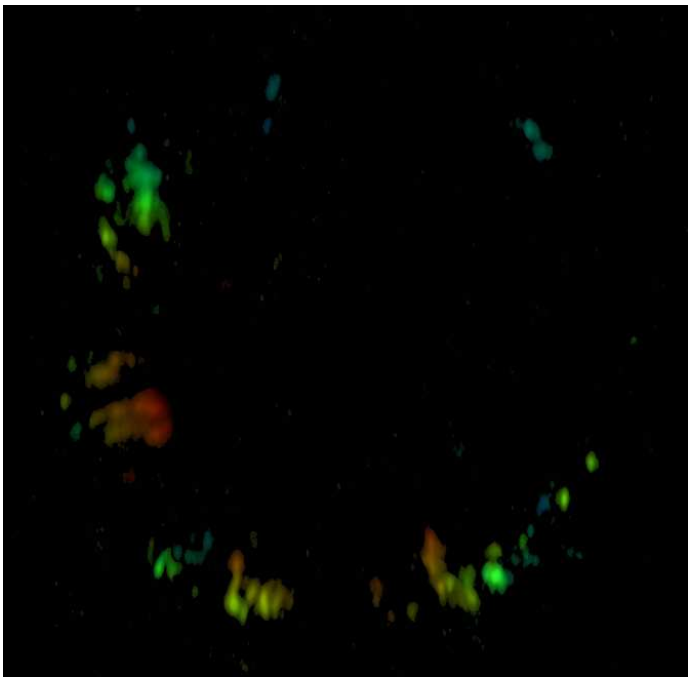

Fig. 2. Color coded velocity image resulting from test processing, blue is approaching, red receding.

TABLE I
Timing results

test	Wall clock run time (min)
Single stream, 1 thread	5.68
Single stream, 2 threads	5.23
Two stream total:	2.70
SplitCh	0.02
MCube	0.37
stream	2.22

so simple that script based systems as the one presented here will work. Other problems are not quite so straightforward. Calculating the multiple facets needed for "fly's eye" wide field imaging is another obvious case. In this case, the construction of the facets while inherently parallel, is a part of a much larger process and is best dealt with by embedding it in a compiled task rather than in a python script.

APPENDIX

The following python fragments give the details of breaking the spectral line imaging problem into parallel streams.

REFERENCES

[1] W. D. Cotton, "Obit: A Development Environment for Astronomical Algorithms," *PASP*, vol. 120, pp. 439–448, 2008.

[2] M. Kettenis, H. J. van Langevelde, C. Reynolds, and B. Cotton, "'ParselTongue: AIPS Talking Python," *Astronomical Data Analysis Software and Systems XV, ASP Conference Series, Vol. XXX, C. Gabriel, C. Arviset, D. Ponz and E. Solono, eds*, p. 497, 2005.

```
# Script to be executed from ObitTalk to process
# Split data
uv = UV.newPAUV ("input", "myStar","Full",1,1, True, err)
source = "star"

spl = ObitTask("SplitCh")
setname(uv,spl)
spl.outDType="AIPS"
nm = uv.Aname
spl.outNames=[nm, nm, nm, nm, nm, nm, nm, nm]
spl.outClass=["Part1","Part2","Part3","Part4", \
              "Part5","Part6","Part7","Part8"]
spl.outSeq  =[1,2,3,4,5,6,7,8]
spl.outDisk =[1,2,4,6,1,2,4,6]
# Selection
spl.BChan = 10
spl.EChan = 118
spl.BIF   = 1
spl.EIF   = 1
spl.Stokes = "I"

# Calibration
spl.Sources =[source]
spl.doCalib = 2
spl.gainUse = 0
spl.flagVer = 1
spl.doPol   = False   # Already done
spl.doBand  = -1      # Already done
spl.g  # Run SplitCh

# Two asynchronous processing streams
procSc1 = ObitScript("Stream1",file="ProcSc1.py")
tw1 = go(procSc1)

procSc2 = ObitScript("Stream2",file="ProcSc2.py")
tw2 = go(procSc2)

# Wait to finish
tw1.wait()
tw2.wait()
```

Fig. 3. Python script fragment to run SplitCh and run the asynchronous scripts procSc1 and ProcSc2.

```
# Proc script 1 for processCh
# Run Imager for stream 1

# Files from SplitCh for this stream
nm =  "myStar"
outNames =[nm, nm, nm, nm]
outClasss=["Part1","Part3","Part5","Part7"]
outSeqs  =[1,3,5,7]
outDisks =[1,4,1,4]

# Imaging parameters
source = "star"
cells = 0.00005
nx = 1024; ny=1024
niter = 1000
Beam=[0.00039,0.000170,0.0]
minFlux  = 0.05
RAShift  = 0.0067
DecShift = 0.0086

# Setup task, set parameters not defaulted
img = ObitTask("Imager")
img.Stokes  = 'I'
img.Sources = [source]
img.Beam    = Beam
img.minFlux = minFlux
img.RAShift = [RAShift];
img.DecShift= [DecShift]
img.NField  = 1
img.Niter   = niter
img.nx      = [nx];
img.ny      = [ny];
img.dispURL = "None"
img.chInc   = 1
img.chAvg   = 1
img.autoWindow = True

# Loop over files, imaging
for i in range(0,len(outNames)):
    tmpUV = UV.newPAUV("part", outNames[i], outClasss[i], \
                       outDisks[i], outSeqs[i], True, err)
    OErr.printErrMsg(err,"Error with split uv data")
    setname(tmpUV,img);
    img.outName  = img.inName
    img.outClass = "I"+img.inClass.strip()
    img.outDisk  = img.inDisk
    img.outSeq   = img.inSeq
    img.out2Disk = img.inDisk
    img.out2Seq  = img.inSeq
    img.g                        # Run Imager
# end
```

Fig. 4. Python script for one processing stream.

```
# Count output channels
count = 0
for i in range(0,len(spl.outNames)):
    if spl.outNames[i][1:5]!="    ":
        tst = (source+spl.outNames[i])[0:12]
        inImg = Image.newPAImage("part", tst, 'I'+spl.outClasss[i], \
                                 spl.outDisks[i], spl.outSeqs[i], True, err)
        OErr.printErrMsg(err,"Error counting image planes")
        count += inImg.Desc.Dict["inaxes"][2]

# Combine
mc = ObitTask("MCube")
mc.DataType = "AIPS"
mc.axNum = 3
mc.axDim = count    # Number of output channels
mc.axPix = 1        # first start plane
mc.outName = uv.Aname
mc.outClass="ICube"
mc.outSeq  = 1
mc.outDisk = 1
mc.copyCC  = True   # Copy Clean Component tables

# Loop over files, concatenating, cleaning temporary files
for i in range(0,len(spl.outNames)):
    if spl.outNames[i][1:5]!="    ":
        tst = (source+spl.outNames[i])[0:12]
        inImg = Image.newPAImage("partI", tst, 'I'+spl.outClasss[i].strip(), \
                                 spl.outDisks[i], spl.outSeqs[i], True, err)
        OErr.printErrMsg(err,"Error with temp image")
        setname(inImg, mc)
        mc.g                                        # Run MCube
        mc.axPix += inImg.Desc.Dict["inaxes"][2]  # next start plane
        # Cleanup
        inImg.Zap(err); del inImg
        tmpUv = UV.newPAUV("partUV",spl.outNames[i], spl.outClasss[i], \
                           spl.outDisks[i], spl.outSeqs[i], True, err)
        tmpUv.Zap(err); del tmpUv
        tmpUv2 = UV.newPAUV("Imager", source, "Imager", \
                            spl.outDisks[i], spl.outSeqs[i], True, err)
        tmpUv2.Zap(err); del tmpUv2

# end
```

Fig. 5. Python script fragment to accumulate the partial image cubes and cleanup intermediate files.