Parallel and Distributed Processing with glish and AIPS++

A.G. Willis
National Research Council of Canada
Dominion Radio Astrophysical Observatory
Penticton, BC, Canada V2A 6K3
tony.willis@nrc.ca

August 16, 2000

Abstract

We discuss why parallel and distributed computer processing will be important for the SKA. Some examples of such processing that are applicable to the SKA are given using AIPS++ and glish.

1 Introduction

One of the SKA science goals (Taylor and Braun, 1999) is to image one square degree of sky at 21 cm wavelength with a half-power beamwidth of 0.1 arcsec. To avoid under-sampling artifacts during image restoration, we will have to sample the sky at about one-third the required resolution, or 0.03 arcsec, so we need to make an image of 120000 x 120000 pixels.

The SKA is unlikely to generate a single image. Rather, it will most likely have some kind of multiple feed system with overlapping fields of view. Let us assume that each feed accurately images a 2 arcmin area to half-power. 2 arcmin equals 4096 pixels at 0.03 arcsec resolution. To overlap adjacent images, allow joint deconvolution and avoid aliasing, the image size should be extended to 8192 pixels. If this form of imaging is used, we will require about 900 such fields.

Also, in order to avoid significant bandwidth smearing to the edge of a 2 arcmin field, we will need to sample the passband in increments of about 2 MHz if the array size is about 400 km or so. The SKA design requires a total bandwidth of about 600 MHz at 21cm (Taylor and Braun, 1999) to detect nanoJy sources, so we will need some 300 channels for continuum imaging. Some form of multi-frequency synthesis will allow groups of channels to be gridded into a single image, but lets assume that we may still need to process some 50 separate channel images for each field. The net result is that we might need up to 45000 images of size 8192 x 8192 to correctly image 1 sq degree.

Today a standard PC can usually handle a single continuum image of size about 4096 x 4096 pixels (relatively) easily. Moore's Law predicts about a factor 90 gain in computing power during the next decade, The net result is that a 2010 PC will probably be about a factor 2000 under-powered for SKA data processing, and this discussion does not even begin to include 3-d off-axis imaging requirements! (Note that Moore's Law does not obviously apply to aperture synthesis image processing. When the Westerbork telescope first came on-line, one could produce 512 x 512 images. 3 decades later the image size that the 'typical' astronomer can process is only about a factor 64 larger.)

Although the above calculations are clearly open to debate, they do suggest that the SKA will have gigantic data processing requirements that will be difficult for the computers of the day to handle. Further, during the next decade we will already need powerful computing systems in order to simulate different SKA designs and to develop new algorithms to process SKA data influenced by SKA-specific imaging artifacts such as time-variable primary beams.

Since many of us do not have access to state-of-the-art supercomputers we must place more emphasis on the development of parallel and distributed processing applications. To date, the majority of work in radio astronomy image processing has centred around the user with access to a single workstation - the classic AIPS model. However COTS (commodity off the shelf) PCs allow us to easily build inexpensive clusters of computers. These 'Beowulf' systems are best suited for applications in which the amount of time devoted to computation exceeds that devoted to message passing between computers in the cluster.

2 Master - Slave Examples

Master - slave parallel computing involves one task (the master) farming out similar computation tasks to a number of slave tasks. When the slaves have finished processing they return the results of their work to the master, who accumulates their contribution into some total. Depending on the problem to be solved, the master then may assign new work out to the slaves. The master-slave paradigm is directly relevant to at least two potential SKA simulator applications: computation of artificial sky models and calculations of model visibilities based on SKA simulated observations of the artificial sky models.

The glish programming language and library supplied with AIPS++ has excellent facilities for building parallel and distributed applications. (See Schiebel and Paxson for a complete description of the glish programming language.) Hopkins el al(2000) have written glish scripts to generate artificial sky images. Their scripts create the artificial sky sequentially from four separate source populations. It is easy to get almost a factor of four speedup in computation by farming out the calculations of the model sky associated with each source population to a separate slave processor, and then having a master script add each model sky to a running total.

Here is a greatly simplified version of the 'master' glish script that controls the parallel processing (# signs precede comments in glish):

```
#################
# We start by defining the glish subsequence 'runsky'. A
# subsequence is a special kind of object which can respond to
# glish messages. The 'reflect=T' option allows the
# subsequence to post messages to itself.
# 'maxpop' is a glish parameter containing an integer number
# that tells us how many source populations we wish to model.
# It is also the number of slave processors that will need to be activated.
runsky := subsequence(maxpop) : [reflect=T] {

# Inside runsky, create 512 x 512 array 'final_image' to store running
# total of images. 'final_image' has initial value of zero.
array_pixels := 512
final_image := array(0, array_pixels, array_pixels)

# start 'slave' glish client scripts running on computers with host names
# slave1 through (a maximum of) slave4. As each client script is launched,
# it gets added to an array of clients called 'image_generator'.
image_generator := [=]
for (popn in 1:maxpop) {
   host_name := spaste("slave",as_string(popn))
   image_generator[popn] := client("image_generator.g",host=host_name)
} # end of for loop
```

```
# the glish variable 'pop_counter' will be used to keep track of
# how many images we have received back from the slave clients
pop_counter := 0

# the glish record 'popn_info' is used to send two pieces of in-
formation to
# each client: 'image_pixels' - the size of the image to be cre-
ated, and
# 'popn' - the number of the source population that the client is to
# use for source modeling
popn_info := [=]
popn_info.image_pixels := array_pixels

for (popn in 1:maxpop) {
   popn_info.popn := popn

# send each 'image_generator' client a 'generate_image' message containing
# the 'popn_info' data
   image_generator[popn]->generate_image(popn_info)

# when each 'image_generator' client finishes it produces a 'post_image'
# message. When the master script catches this message, it adds the
# contents of the message, an image array, to the total image
# and increments the 'pop_counter' variable by 1
   whenever image_generator[popn]->post_image do {
      pop_counter := pop_counter + 1
      final_image := final_image + $value
# if the master has received all images from the slaves
# it sends a message to itself that it has finished getting all data
      if (pop_counter == maxpop)
         self->finished_imaging()
   } # end of whenever block
} # end of for loop

# the master script waits at this point until it gets the 'finished_imaging'
# message
await self->finished_imaging
print "created artificial sky image!"
```

```
} # end of subsequence definition

# we have finished defining the subsequence. So we will now in-
stantiate one
# and tell it to generate a model sky from four different source populations
example := runsky(4)
#################
```

Here is the corresponding glish script that runs on each slave:

```
#################
# image_generator.g - glish slave script to produce artificial sky images

# the script basically works as follows:
# whenever the script is sent a 'generate_image' message it
#   1) first calls the function 'generate_source_list', with the appropriate
#      population number. 'generate_source_list' performs its calculations
#      and eventually returns a list of sources in the glish record
#      'source_list'.
#   2) then calls the 'make_image' function with the list of sources
#      and the image size as parameters. 'make_image' performs its
#      calculations and returns an artificial sky for this population
#      in the image array variable 'artificial_image'.
#   3) finally creates a glish message 'post_image' with the
#      message contents being the array 'artificial_image'.

# include glish code that gererates a list of sources for specified
# population
include 'generate_source_list.g'

# include glish code that converts a list of sources into a model image
include 'make_image.g'

# the following line just makes sure this script is running as a
# glish client script capable of receiving messages from another script
if (system.is_script_client) {
   whenever script->generate_image do {
      source_list := generate_source_list($value.popn}
      artificial_image := make_image(source_list, $value.image_pixels)
      script->post_image(artificial_image)
```

```
  } # end of whenever block
} # end of if block
#################
```

You should be aware that, while glish scripts are useful to try out ideas for parallel processing, once one has a working concept that needs to be turned into an industrial strength system, it is a good idea to convert the glish scripts into compiled C++ glish clients.

Another master-slave type application that can benefit extensively from parallelization is the direct Fourier transform computation of model visibilities from an artificial sky image or list of source components. Direct, rather than fast Fourier transforms, will be required to simulate the influence on an observed visibility of instrumental effects such as a time variable primary beam.

In this case the master first sends a list of source components or a model image to each of the available slaves. It then sends each slave the necessary parameters to describe a specific interferometer baseline. Using the model data and the baseline parameters the slave computes model visibilities for the specified baseline, and returns the computed visibilities back to the maaster. If there are still baselines that require visibility computation, the master then sends the slave parameters for another baseline. This cycle continues until visibilities have been computed for all baselines. One can get a significant speed up by doing this simple parallelization. A DFT inversion of a 4000 pixel model of Cygnus A that took 32 hours to complete as a sequential application on a Sun ultraSparc 200 was finished in a hour when the application was ported to a Beowulf cluster of 16 450 MHz PIIs.

3 Pipeline Processing of Data

The eventual SKA will almost certainly have streams of data coming from multiple sources, such as focal plane arrays, which have to go through similar calibration and reduction processes. In such cases we can process the data in parallel pipelines (Figure 1).

Using glish and the AIPS++ class library, we are implementing such a data reduction system for the AutoCorrelation Spectrometer Imaging System (ACSIS) that DRAO is building for the JCMT (see Hovey et al., 2000 for details). At the highest data rate, the ACSIS system must handle 16 data streams, each generating 8192 channel spectra every 50 millisec. A very general schematic of this system is shown in Figure 2.

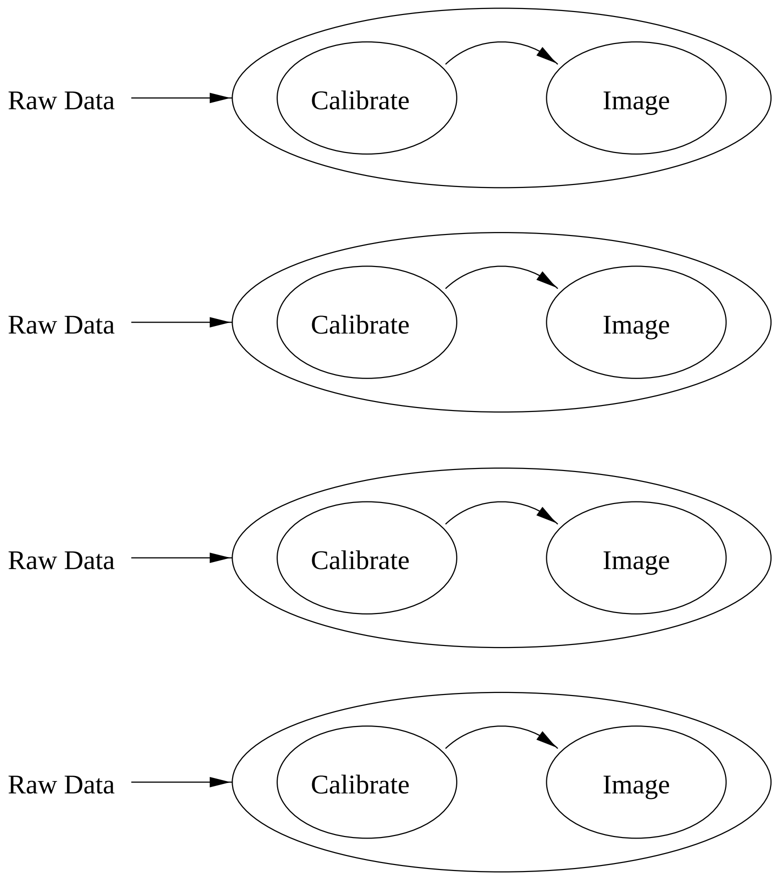

Figure 1: pipeline processing in parallel. A number of similar raw data sources can sometimes be processed in parallel.

The system basically works as follows: at start time a controlling glish script reads a configuration file similar to the following one.

```
sync_task slave1
sync_task slave2
sync_task slave3
reducer slave4
reducer slave5
reducer slave6
gridder slave7
gridder slave8
gridder slave9
```

This configuration file tells the glish script which glish C++ clients to start and which machine the clients should run on. So in our case the system would start separate sync_task clients on slaves 1, 2 and 3, reducers on slaves 4, 5 and 6, and gridders on slaves 7, 8 and 9. The function of a sync_task is to synchronize raw correlator lag data together with other telescope and antenna data by means of sequence numbers. Reducers calibrate the data according to a recipe; they are essentially programmable calculators. Finally, the gridders grid the data into 3-d data cubes (usually with RA, DEC, frequency axes). When the C++ clients are started they open direct socket or pipe connections to the next task in the pipeline. So, for instance, the first sync_task opens a direct connection to the first reducer, etc. These connections ensure that data can be sent between clients at the maximum possible rate.

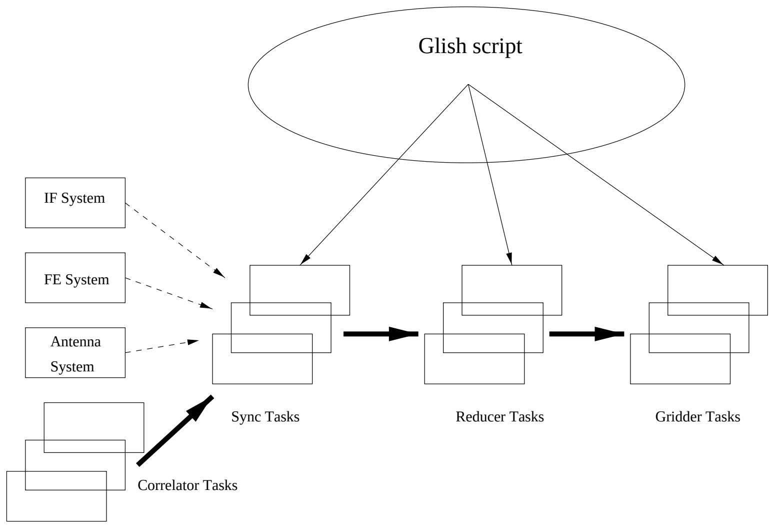

Figure 2: A general schematic of the ACSIS correlator reduction system.

An advanced prototype version of this reduction system is currently running successfully on a 16 PC Beowulf cluster at DRAO.

We have used the AIPS++ glish client library to implement message passing within this system. However, other messaging systems, such as CORBA, could have been used to implement a similar system. Additional information on doing parallel processing specifically with AIPS++ and glish may be found in AIPS++ Note 230 (Willis, 2000).

4 Conclusions

Parallel and distributed processing can provide a great increase in speed over normal sequential applications in many areas related to radio astronomy, including application areas that will be directly relevant to the SKA. If we are to successfully run SKA simulations and develop image processing algorithms for the SKA during the next decade we will almost certainly need to make extensive use of parallel processing in order to obtain the necessary computing power.

References

[1] Hopkins, A., Windhorst, R., Cram, L., & Ekers, R. 2000, Exp. Ast., (in press) (astro-ph/9906469)

[2] Hovey, G. et al. ”A New Spectral Line, Multi-beam Correlator System for the James Clerk Maxwell Telescope”, Proceedings of SPIE Meeting, March 2000, Munich (in press)

[3] Schiebel, D., & Paxson, V., ”The Glish 2.7 User Manual”, http://aips2.nrao.edu/docs/reference/Glish/Glish.html

[4] Taylor, A.R., & Braun, R. (eds) ”Science with the Square Kilometer Array”, March 1999

[5] Willis, A.G., ”AIPS++ NOTE 230 - Advanced Programming with AIPS++ and Glish”, March 2000, http://aips2.nrao.edu/docs/notes/230/230.html