NOTE 230 – Advanced Programming with AIPS++ and *Glish*

Anthony G. Willis
National Research Council of Canada
Herzberg Institute of Astrophysics
Penticton, BC V2A 6K3
Canada

Version 1.1/12 Sep 2001

Contents

0.1 Introduction

This tutorial attempts to bridge the gap between the introductory *Glish* programming material in *Getting Started with* Glish by Rick Fisher and the very advanced material in *How to Write an AIPS++ Application* by Brian Glendenning and Tim Cornwell

The first section of the tutorial covers object-oriented programming techniques in the AIPS++ environment. In the programming community object-oriented programming is presently considered to be a Good Thing. An object can be defined in many ways, but a practical definition might be that an object acts as a container and processor of data. You interact with an object by sending it messages. These messages might contain data items that you wish stored inside the object, they might be instructions to the object to process the data stored inside it in some way, or they can be messages to the object telling it that you want to retrieve and see some or all of the data that it contains. Messages generally have an appearance similar to function calls in older programming languages such as FORTRAN or C.

Although the basic functionality of *Glish* is covered in the *Glish Manual* and *Getting Started with Glish*, neither of these documents show you a methodology for developing object-oriented applications in *Glish* and c++. In particular they do not describe the ideas and techniques that lie behind the Distributed Object (DO) paradigm used to develop AIPS++ applications. The note by Glendenning and Cornwell assumes a fairly high level of awareness about the DO methodology and merely tells you the recipe to be used to create DOs in the AIPS++ environment.

This tutorial does not make specific use of the AIPS++ programming environment, but after reading it you should be in a fairly good position to understand the AIPS++DO system and be able to understand the design concepts behind an AIPS++ *Glish* script. You should also be able to use *Glish* and the AIPS++ class libraries to develop independent applications.

The section on object-oriented programming may also be of some interest to end users of AIPS++. After reading that section you will have a better understanding why AIPS++ commands look and feel the way they do.

The second section will focus on parallel programming in the AIPS++ environment. However, the examples shown there will illustrate some properties of *Glish*that any *Glish* programmer should be aware of.

To a large extent the contents of the tutorial reflect the parts of AIPS++ and *Glish* that I have personally had to learn about or had experience with during the past year or so.

0.2 Object Oriented Programming with *Glish*

There are two things you should know about *Glish* records and *Glish*functions in order to do object-oriented programming with *Glish*. Firstly, you should be aware of the enhanced data types that can be stored in a *Glish* record. Some of you may be familiar with structures from the C programming languages. A *Glish*record is somewhat similar to a C structure, but in addition to holding expected data items of a numeric or string type, *Glish* records can also contain functions, as functions are just another data type in *Glish*. *Glish* records can also contain a mysterious data type called an agent which we shall describe in detail later.

Let's look at a *Glish* record that contains a mixture of traditional data types and *Glish* functions:

Running the following script

---

demorecord.g

```
demo := [=]
demo.demofunction := function(a,b) {
        print "a = ",a
        print "b = ",b
        return a-b
}
demo.intnumber := 3
demo.floatnumber := 5.5
c := demo.demofunction(10,6)
print "c = ", c
print demo
print "intnumber = ",demo.intnumber
print "floatnumber = ",demo.floatnumber
```

yields the result

```
a = 10
b = 6
c = 4
[demofunction=<function>, intnumber=3, floatnumber=5.5]
intnumber = 3
floatnumber = 5.5
```

So we have created a *Glish* record that holds both an integer and floating point number as well as a function!

The second thing you need to know about *Glish* is that *Glish* functions can contain embedded definitions of other functions.

running the script

demofunction.g

```
function demo(x) {

   function multiply2(z) {
      result := 2 * z
      return result
   }
```

```
   y := x * x
   print "y = ", y
   y1 := multiply2(y)
   print "y1 = ",y1
   return
}
demo(5)
```

yields the result

```
y = 25
y1 = 50
```

Armed with these pieces of knowledge about *Glish* records and *Glish*functions, we can quite easily generate an object-oriented paradigm within *Glish*. Here is a sample *Glish*script that closely resembles a c++ class

demoobject

```
# define a function called demoobject
const demoobject := function()
{
# Here we will use a glish record called 'private' to store private data items
# and a glish record called 'public' that will provide the publicly
# visible methods to access the private data items. Note that 'public' and
# 'private' are not reserved works as they are in the object-oriented
# language c++. Here they are used as normal glish record names, but I have
# emulated the way private data and public access methods would be used
# in c++. Here, we initialize the two glish records
   private :=[=]
   public := [=]

# a function that will assign some data type to the 'private' glish record
   const public.insertdata := function(x) {
      wider private
      private.x := x
      return
   }
```

```
# a function to add some amount, default value is 5, to the
# internal data contents
   const public.addtodata := function(amount=5) {
      wider private
      if (is_record(private.x))
         print 'unable to add a numeric amount to a record'
      if (is_numeric(private.x))
         private.x := private.x + amount
      return T
   }

# a function to extract the data contents of the 'private' glish record
   const public.getdata := function() {
      wider private
      return private.x;
   }

# a function to delete internal data elements when the closure object
# goes out of scope
   const public.done := function() {
      wider private, public
      private := F
      public := F
   }


# the record 'public' which just contains data access functions
# is the item returned to the outside world when a glish variable having
# the 'demoobject' function as its value is created
   return public
}

# The object has been defined. The remainder of the script shows a simple
# example of its usage

#run the demoobject 'function' and assign the returned publicly
#visible access methods to a glish variable a
a := demoobject()
```

```
# create a simple glish 1-d vector with 10 elements
examplein := [1:10]
print "examplein = ",examplein

# store, or hide, the 1-d vector, examplein, inside a. The glish variable 'a'
# uses the visible access function 'insertdata' to assign the contents
# of examplein to the hidden record member private.x.
a.insertdata(examplein)

# modify the internal contents of the object
# by adding a default amount of 5 to numeric contents
a.addtodata()

# retrieve the internal data contents and assign them to the
# variable exampleout
exampleout := a.getdata()

# print contents of what is now stored in exampleout
print "exampleout = ",exampleout
```

running this script gives

```
examplein = [1 2 3 4 5 6 7 8 9 10]
exampleout = [6 7 8 9 10 11 12 13 14 15]
```

A detailed explanation of what's happening with the quasi-object "a" follows. When "a" is created it contains the four functions insertdata, getdata, addtodata, and done. These functions themselves contain a reference, through use of the *Glish* keyword "wider" to the entire call frame (all the local variables) of "demoobject" By means of the "wider" keyword both "getdata" and "insertdata" can access these variables as they both point to the same call frame. Thus getdata, addtodata, and insertdata have shared state, and therefore we have essentially been able to create an object.

The "wider" keyword says is that a function wants to use the "wider" variable at a wider scope (though not global) if it is modified, i.e. modify the wider scoped variable instead of making the variable local to the function (the *Glish* default).

Also note the presence of the "done" function. Before the closure object, a, that has been created goes out of scope one should have a command

```
a.done()
```

This requirement is due to a bug in *Glish* which should hopefully be fixed by the end of 1999.The problem is that the record (public) which is returned holds a (the only) reference to each of the functions, but each of the functions hold a reference to the record (a bit difficult to comprehend, but thats what Darrell says).

Given

```
a := demoobject()
```

when the variable "a" is re-assigned:

```
a := F
```

"a"s reference count goes from 3 to 2, but the 2 is the result of the functions internal to "a" holding a reference to "a"

0.3 Subsequences

Subsequences are a kind of super function. They allow you to create an object which you can communicate with by means of events as well as function calls. Let's rewrite our demoobject script as a subsequence.

---

demosubsequence.g

```
demosubseq := subsequence()
{
   private :=[=]

# store some data internally
   whenever self->insertdata do
      private.x := $value

# retrieve the stored data by responding to an event
   whenever self->getdata do
      self->getdata(private.x);

# retrieve the stored data by responding to a function call
   self.getdata := function() {
      wider private
```

```
      return private.x
   }
}
# The subsequence has been defined. The  remainder of the script shows
# how it might be used

# instantiate the subsequence
a := demosubseq()
print "a = ",a

# create a data vector
examplein := [1:10]
print "examplein = ",examplein

# store the data  in the subsequence
a->insertdata(examplein)

# retrieve the data by sending a message (function call) to the subsequence
exampleout := a.getdata()
print "exampleout = ",exampleout

# retrieve the data by sending an event to the subsequence
exampleout1 := a->getdata()
print "exampleout1 = ",exampleout1
```

---

running this script gives the result

```
a = [*agent*=<agent>, getdata=<function>]
examplein = [1 2 3 4 5 6 7 8 9 10]
exampleout = [1 2 3 4 5 6 7 8 9 10]
exampleout1 = [1 2 3 4 5 6 7 8 9 10]
```

A subsequence is an example of an agent. You can communicate with an agent via events. One thing that differs with a function is that a subsequence contains a predefined variable, “self” that refers to the value or identity of the agent. So when we create a function called self.getdata, this function is attached to the agent we create, a. Once we have stored some data inside the subsequence we have created, we can retrieve the contents of the stored data by either sending the subsequence a "getdata" event, or by calling the attached "getdata" function. (Note that many AIPS++ scripts simply use "self" as a record similar to the way we have used "private" and "public" in the above examples. This can be confusing.)

When we send the subsequence a getdata event as in

```
exampleout := a->getdata()
```

we await a reply from the subsequence. When the subsequence posts an event with the same name as the one it received, the value associated with the posted event is taken to be the "reply" value. So in our example, the contents of private.x are retrieved and assigned to exampleout.

You should be aware that in the current version of *Glish* there is no way to destruct a subsequence once you have created it. In many cases this is not necessarily a problem.

The "whenever" statement used in the above example is an extremely powerful feature of *Glish*. However, used incorrectly, it can cause some unexpected behavior in a *Glish* script. As pointed out on p. 90 of the manual, especially avoid creating *Glish* scripts where the interpreter may end up processing a given whenever statement more than once. This can easily happen if you have a whenever clause inside a function which could be called more than once. Especially avoid use of whenevers in DO methods that could be called repeatedly by an end user from the command line or the AIPS++ GUI. (In scripts associated with the actual AIPS++ project, 'whenevers' seem to be mostly used to handle events generated in GUI programming, so they're unlikely to be present in functions or methods called directly by a user).

To avoid such a situation, the function shown on p. 90 of the manual could have been written as part of a *Glish* object in the style

```
...
private.whenevercalled := F
...
public.reportfoo := function(x) {
  y:= 3
  if (!private.whenevercalled) {
    whenever x->foo do {
         print y
         y +:= 1
    }
```

```
    private.whenevercalled := T
  }
  y := 7
}
```

We shall show an example of what can happen if a function allows whenevers to be processed more than once in the section on parallel processing.

Also, as we shall see in a later example, the behaviour of a subsequence can sometimes be different from what one might intuitively expect because the *Glish* interpreter essentially behaves like a multi-threaded application when it handles events. In our above demo script we sent two events to the subsequence in rapid succession, one to store the data and one to retrieve it. When the interpreter receives an event it opens a quasi-thread to handle it. That means that in our above example the thread to retrieve the data might have attempted to retrieve nothing if the thread to insert the data had not not yet completed its activity. Luckily, in the above example, the insertion thread completed its activity, before retrieval was attempted.

0.4 Distributed Objects

Glish scripts execute whole array operations quite quickly. However, examine the following script and then try running it.

demoarray.g

```
const demoarray := function(m=32, n=32)
{
   private :=[=]
   public := [=]
   private.xsize := m
   private.ysize := n
   print 'starting constructor'
   private.demoarr := array(0,m,n)

   for (i in 1:m) {
      for (j in 1:n) {
         r := sqrt ( i * i + j * j)
             private.demoarr[i,j] := cos(0.6*sqrt(i*80./m)-16.0*j/(3.*n)) *
                 cos(16.0*i/(3.*m)) +(i/m-j/n) + 0.05*sin(r)
```

```
      }
   }
   print 'ending constructor'

# return the maximum value of the constructed array using built in glish
# 'max' function
   const public.getmax := function() {
      return max(private.demoarr);
   }

# return the maximum value of the constructed array the slow way
   const public.getslowmax := function() {
      localmax := -20000.0
      for (i in 1:private.xsize) {
         for (j in 1:private.ysize) {
            if (private.demoarr[i,j]  > localmax)
               localmax := private.demoarr[i,j]
         }
      }
      return localmax
   }

# a function to delete internal data elements when the closure object
# goes out of scope
   const public.done := function() {
      wider private, public
      private := F
      public := F
   }
   return public
}
# object has been defined, now use it

a := demoarray(256,256)
fastmax := a.getmax()
print "fast max = ",fastmax
slowmax := a.getslowmax()
print "slow max = ",slowmax
```

If you run this *Glish* script on a 450 MHz PII computer, you can pretty well go for lunch while it is executing. (On a trivia note, the really alert reader might recognize that this is the array computed in the fourth demo program of the PGPLOT package.)

Glish performs well on whole array arithmetic, but does not perform very well when it is required to examine individual array elements on a one by one basis.

How to proceed? You can farm out a computationally intensive task that does not perform well in *Glish*, to a dedicated c++ *Glish* client that does the processing. To the external user of the script, nothing will appear to have changed, except the script will run a lot faster. This concept forms the basis of the AIPS++ distributed object paradigm. The user sees what appears to be a single object, whose methods, in reality are distributed between the *Glish* script and a c++ based *Glish* client.

We take the above script and rewrite it as

---

distributed.g

```
const demoarray := function(m=32, n=32)
{
   private :=[=]
   public := [=]
   private.xsize := m
   private.ysize := n

# launch c++ client that will do the intensive calculations
# a client is an agent, so you can communicate with it by sending it events
   private.arraygenerator := client("./arraydemo")

# create record with information for the client
   arrayinfo :=[=]
   arrayinfo.xlength := private.xsize
   arrayinfo.ylength := private.ysize

# send client a createarray event, and assign the data contents of
# the event that the client sends back to the script to the variable
# private.demoarr
   private.demoarr := private.arraygenerator->createarray(arrayinfo)
```

```
# return the maximum value of the constructed array using built in glish
# 'max' function
   const public.getmax := function() {
      return max(private.demoarr);
   }

# Return the maximum value of the constructed array by using the client
# Send the client an event findmax. Store the contents of the client's
# reply in the variable localmax
   const public.getslowmax := function() {
      localmax := private.arraygenerator->findmax(private.demoarr)
      return localmax
   }

# a function to delete internal data elements when the closure object
# goes out of scope
   const public.done := function() {
      wider private, public
      private := F
      public := F
   }
   return public
}

# The object has been defined. Here is an example of its usage.

# create an instantiation
a := demoarray(256,256)

# find the maximum using the built-in glish method

fastmax := a.getmax()
print "fast max = ",fastmax

# find the maximum using the glish client
slowmax := a.getslowmax()
print "slow max = ",slowmax
```

This script is similar to the previous one, but we have farmed off the two computationally intensive tasks to a *Glish* client “arraydemo” To the user of the script, however, things still appear the same, except that when it runs there will be a slight difference in speed to say the least.

Note the following: if I ran the above script, and then created a new ’demoarray’ object with, say

```
newdemo := demoarray(300,300)
```

the system would start to run a second version of the computation client “arraydemo” This is obviously wasteful. In the actual AIPS++Distributed Object (DO) environment, you start up clients by means of a “default-servers” method which checks if an executable of the requested client is already running. If one is, the method just returns a reference to that client instead of starting up a new one.

In the above script the *Glish* script communicated with the *Glish* client via the event

```
private.demoarr := private.arraygenerator->createarray(arrayinfo)
```

Here the client, identified by the name “private.arraygenerator” inside the script, receives an event, createarray, with contents “arrayinfo” How does the client respond to such events? There are two ways to write clients that can respond to *Glish* message passing, the painful way and the AIPS++way. The painful method makes use of the message-passing interface that comes with the basic release of *Glish*. However, the AIPS++ group has developed some c++ class libraries that are wrapped the basic *Glish*message-passing interface that make message passing a whole lot easier; I suggest that you use these classes if they are available to you.

To show the difference between the two approaches I first show the c++ code for a version of the client that just makes use of the basic *Glish*message passing functions. The basic message passing functions are summarized on pages 260 to 271 of the *Glish* manual. The example shows how they would be used in practice. The example does make use of the AIPS++Array classes, but I assume that you could replace this component of the code by an array class of your own choosing if you do not have the AIPS++ class library available.

---

arraydemo.cc

```
// code for arraydemo.cc using basic glish calls
#include <string.h>
#include <Glish/Client.h>
#include <aips/Arrays/Matrix.h>

// function to create and assign data values to a matrix
Value createarray(GlishEvent *evt) {
// retrieve array dimensions
   Value* val = evt->value;
   Int xxlen, yylen;
   Int *xlen = (Int *)val->FieldIntPtr("xlength",xxlen);
   Int *ylen = (Int *)val->FieldIntPtr("ylength",yylen);
   Int m = *xlen;
   Int n = *ylen;
// create and compute array
   Matrix<Float> a;
   a.resize(m,n);
   for (Int i = 0; i<m; i++) {
      for (Int j = 0; j<n; j++) {
// the following line converts things to equivalent glish 1-based indexing
         Int ii = i+1; Int jj = j+1;
         Float tmp = ii * ii + jj *jj;
         Float r = sqrt (tmp) ;
         a(i,j) = cos(0.6*sqrt(ii*80./m)-16.0*jj/(3.*n)) *
            cos(16.0*ii/(3.*m)) +(Float(ii)/m-Float(jj)/n) +
            0.05*sin(r) ;
      }
   }

// Now put computed array into a form that can be sent back
// to calling glish script.
// The code is gratefully adapted from the example in
// the aips++ GlishValue class.
//
// Using the aips++ wrapper classes, all the code from here to the
// end of the function can be replaced by one line of code.

// Get a pointer to the values in the aips++ array
   Bool del;
   Float *aipsFloatPtr = a.getStorage(del);
```

```
// Create a receptor array -  note that float != Float!!
   float *glishfloatPtr = new float[a.nelements()];

// Copy the values from the aips++ array into the glish array
   for (uInt i=0; i < a.nelements(); i++)
      glishfloatPtr[i] = float(aipsFloatPtr[i]);

// Delete aips++ pointer, if necessary.
   a.putStorage(aipsFloatPtr, del);

// OK; create the glish Value, with variable name 'native'
// from the initialized storage
   Value native(glishfloatPtr, a.nelements());

// Now we need to attach the shape to the glish Value as an attribute
   IPosition aShape = a.shape();
   int shapeArray[a.ndim()];
   for (uInt i=0; i < a.ndim(); i++)
      shapeArray[i] = aShape(i);

   Value *shapeValuePtr = new Value(shapeArray, a.ndim(), COPY_ARRAY);
   native.AssignAttribute("shape", shapeValuePtr);
   Unref(shapeValuePtr);
// Done! the variable 'native' now contains a copy of aips++ array a,
// with both data and attributes.
   return native;

}

// function to find the maximum element of an array
Value findarraymax(GlishEvent *evt) {
   Value* val = evt->value;

// Using the aips++ wrapper classes, all the code from here to the
// 'find maximum' comment can be replaced by one line of code.

// make sure the value's type is float
   val->Polymorph(TYPE_FLOAT);
```

```
// how many elements coming in
   int numarrayelem = val->Length();

// pointer to the low-level array
   float* glishfloatPtr = val->FloatPtr();

// what is its shape? Get it from the attribute associated with a glish array.
   const Value *shapeValuePtr = val->HasAttribute("shape");
   int* shapeArr = shapeValuePtr->IntPtr();

// create an internal array
   Matrix<Float> a(shapeArr[0],shapeArr[1]);

// Get a pointer into aips++ array
   Bool del;
   Float *aipsFloatPtr = a.getStorage(del);

// Copy the values from the glish array into the aips++ array
   for (int i=0; i < numarrayelem; i++)
         aipsFloatPtr[i] = Float(glishfloatPtr[i]);

// Delete pointer, if necessary.
   a.putStorage(aipsFloatPtr, del);

// find maximum
   Float localmax = -100000.0;
   for (Int i = 0; i <shapeArr[0];i++) {
      for (Int j = 0; j < shapeArr[1]; j++)
         if (a(i,j) > localmax)
            localmax = a(i,j);
   }

// turn localmax into a Value
   Value native((float)localmax);

   return native;
}

int main(int argc, char *argv[])
{
```

```
  Client c(argc, argv);
  GlishEvent *evt;

// Create the equivalent of an event loop
// and define functions to be called with a particular event
// is received.
  while ((evt = c.NextEvent()))
    {
      if (!strcmp(evt->name, "createarray") ) {
      Value nativevalue = createarray(evt);
      if (c.ReplyPending())
         c.Reply(&nativevalue);
   }
      else
      if (!strcmp(evt->name, "findmax") ) {
      Value nativevalue = findarraymax(evt);
      if (c.ReplyPending())
         c.Reply(&nativevalue);
   }
      else
   c.Unrecognized();
    }
  return(0);
}
```

---

and here is the code for a version of the client, arraydemo.cc, that makes use of the wrapper classes supplied with the AIPS++ programming environment:

AIPS++ wrapper.cc

```
// code for arraydemo.cc using aips++ wrapper classes
#include <aips/Glish.h>
#include <aips/Arrays/Matrix.h>

// function to create and assign data values to a matrix
Bool createarray(GlishSysEvent &event, void *) {
   GlishSysEventSource *glishBus = event.glishSource();
   GlishValue glishVal = event.val();
```

```
// check that incoming argument is a glish record
   if (glishVal.type() != GlishValue::RECORD) {
      if (glishBus->replyPending())
         glishBus->reply(GlishArray(False));
         return True;
   }
// OK - its a glish record so get contents
   GlishRecord glishRec = glishVal;
   Int xlen, ylen;
   if (glishRec.exists("xlength")) {
      GlishArray tmp;
      tmp = glishRec.get("xlength");
      tmp.get(xlen);
   }

   if (glishRec.exists("ylength")) {
      GlishArray tmp;
      tmp = glishRec.get("ylength");
      tmp.get(ylen);
   }

// now compute array elements
   Matrix<Float> a;
   Int m = xlen;
   Int n = ylen;
   a.resize(xlen,ylen);
   for (Int i = 0; i<xlen; i++) {
      for (Int j = 0; j<ylen; j++) {
// the following line converts things to equivalent glish 1-based indexing
         Int ii = i+1; Int jj = j+1;
         Float tmp = ii * ii + jj *jj;
         Float r = sqrt (tmp) ;
// need to cast ii and jj to Float in the following line to emulate
// glish style of doing arithmetic
         a(i,j) = cos(0.6*sqrt(ii*80./m)-16.0*jj/(3.*n)) *
            cos(16.0*ii/(3.*m)) +(Float(ii)/m-Float(jj)/n) +
            0.05*sin(r) ;
      }
   }
// The replyPending() method will return true for our example, because
```

```
// the parent glish script is waiting for a reply from the client.
   if (glishBus->replyPending()) {
// Convert from aips++ array to a glish array
// conversion is much easier when one can use the aips++ GlishArray class
      GlishArray demo = a;

// Here the client posts an event which is send back to the calling glish script
      glishBus->reply(demo);
   }
   return True;
}

// function to find the maximum element of an array
Bool findarraymax(GlishSysEvent &event, void *) {
   GlishSysEventSource *glishBus = event.glishSource();
   GlishArray tmp = event.val();

// convert from GlishArray to a aips++ matrix
   Matrix<Float> localarray;
   tmp.get(localarray);
   IPosition arraydims = localarray.shape();
   Float localmax = -100000.0;
   for (Int i = 0; i <arraydims(0);i++) {
      for (Int j = 0; j < arraydims(1); j++)
         if (localarray(i,j) > localmax)
            localmax = localarray(i,j);
   }

// convert the interval data type into a GlishArray that can be
// used by the external glish script
   if (glishBus->replyPending()) {
      GlishArray returnedvalue = localmax;
      glishBus->reply(returnedvalue);
   }
   return True;
}

int main(int argc, char *argv[])
{
   GlishSysEventSource glishStream(argc, argv);
```

```
//  Define callback event handlers.
//  The addTarget method of the aips++ GlishSysEventSource class
//  defines the functions that will handle particular incoming events
//  So, for example, when the client receives a 'findmax' event,
//  control will be transferred to the findarraymax function.
   glishStream.addTarget(findarraymax, "findmax");
   glishStream.addTarget(createarray, "createarray");

//  loop and handle incoming events as they are received
   glishStream.loop();
}
```

The advantages of using the second method should be obvious.

Note the following about clients like the one above. If you send it a "createarray" event immediately followed by a "findmax" event the client will first completely do the processing associated with the "createarray" event (in the createarray function) before it begins the processing associated with the "findmax" event. This single threaded behaviour is different from the (potentially) multi-threaded behaviour of a subsequence.

0.5 Multi threading *Glish* clients

Glish clients are single threaded. If a *Glish* client receives an event, the client will execute the function associated with that client to completion, before it responds to the next event.

However, AIPS++ is (generally) an interactive system. In interactive systems, scripts (and clients) should remain responsive to user requests.

Say you want a client to begin a massive computing job, but you still want the client to remain responsive to incoming events, such as a query for status of the computation, a query for retrieval of some temporary data associated with the computation, etc. There is a way to make the client simulate multi-threaded behavior, by breaking the computation task into segments that can be easily resumed after another event has been handled.

Here is a (silly) client which when sent a "compute" event will multiply a vector by 1.0 two billion times. If you tried to do this as one uninterrupted task, the client will remain unresponsive for a very long time. You can send it all the events you want and nothing will happen. Even if you exit from glish the client will continue running happily because it will be unable to respond to the exit command until a long time in the future.

In order to make the client remain responsive to events being sent to it we break the computation task down into subsections, each of which do the array multiplication 5000 times. After 5000 computations have been done, the subroutine updates a counter so the task knows where it is in the computation process, and then posts a "compute" event. The "threads.g" script forwards this event back to the client. The client catches the event and then proceeds 5000 iterations further.

However, if you want to send another type of event to the client, the client remains responsive. In the simple example, we send the client a "getstatus" event as we would like to know how it is progressing with the computation. The client will handle this getstatus event in a timely way, as it will be placed after the last posted "compute" event. After the "getstatus" event has been handled, the last thing the callback function that handled the 'getstatus' event does is post a new "compute" event, so the computation cycle will resume.

---

threads.cc

```
#include <aips/Glish.h>
#include <aips/Arrays/Vector.h>
#include <aips/iostream.h>

Int counter = 0;
Bool First = True;
Bool DoCompute = True;
Vector<Float> a(8192);

// a function to handle status requests
Bool handlestatus(GlishSysEvent &event, void *) {
   GlishSysEventSource *glishBus = event.glishSource();
   cout<<"caught getstatus event at computecounter "<< counter<<endl;
   GlishArray stat(counter);
   glishBus->postEvent("computestatus",stat);
   if (DoCompute)
      glishBus->postEvent("compute", "dummy");

   return True;
```

```
}

// a function to do heavy computing
Bool computearray(GlishSysEvent &event, void *) {
   GlishSysEventSource *glishBus = event.glishSource();
   cout <<"caught compute event at counter "<<counter<<endl;
   if (First) {
      a = 1.0;
      First = false;
   }

// compare behaviour of this loop with what would happen if you
// uncommented the five commented out lines below.
   if (DoCompute) {
      for (Int i = counter; i < counter+5000; i++)
            for (Int j = 0; j < 8192; j++)
                  a(j) = a(j) * 1.0;
   }
// if (DoCompute) {
//    for (Int i = 0; i < 2000000000; i++)
//       for (Int j = 0; j < 8192; j++)
//          a(j) = a(j) * 1.0;
// }
   counter = counter + 5000;
   if (counter > 2000000000)
      DoCompute = False;
   else
      glishBus->postEvent("compute", "dummy");
   return True;
}

int main(int argc, char **argv) {
   cout<<"threads client started"<<endl;
   GlishSysEventSource glishStream(argc, argv);

//  define callback event handlers
   glishStream.addTarget(computearray, "compute");
   glishStream.addTarget(handlestatus, "getstatus");

   glishStream.loop();
```

```
}
```

Here is the script which launches the above client

threads.g

```
a:= client("./threads")
#link a->continue to a->compute
whenever a->computestatus do
      print 'client has completed compute integration ',$value
whenever a->compute do
   a->[$name]($value)
```

after starting this script, sending the command

```
a->getstatus()
```

followed by a few additional getstatus commands
results in something like

```
threads client started
a->getstatus()
caught getstatus event at computecounter 0
client has completed compute iteration  0
caught compute event at counter 0
caught compute event at counter 5000
caught compute event at counter 10000
a->getstatus()
caught compute event at counter 15000
caught getstatus event at computecounter 20000
caught compute event at counter 20000
client has completed compute integration  20000
caught compute event at counter 25000
a->getstatus()
caught compute event at counter 30000
caught compute event at counter 35000
caught getstatus event at computecounter 40000
caught compute event at counter 40000
client has completed compute integration  40000
caught compute event at counter 45000
... etc ...
```

so we continue to receive a timely response from the client.

Another (and probably better) solution to this problem was provided by Darrell Schiebel. Instead of posting a message to itself every 5000th step in the computation, the following variant of the program calls a function

```
checkevents
```

every 5000th step. This function will process the next event waiting in the queue, and after that event has been processed, control is returned to the point in the compute cycle where

```
checkevents
```

was called.

---

secondthreads.cc

```
#include <aips/Glish.h>
#include <aips/Arrays/Vector.h>
#include <aips/iostream.h>

Int counter = 0;
Bool available = True;
Bool proceed = False;

void checkevents( GlishSysEventSource &s, int count=1 ) {
//    cout << "---------- starting check #" << count << " ----------" << endl;
    while ( s.waitingEvent( ) )
        {
        GlishSysEvent e = s.nextGlishEvent();
        s.processEvent( e );
        }
//    cout << "---------- ending check #" << count << "   ----------" << endl;
}

Bool handlestatus(GlishSysEvent &event, void *) {
        GlishSysEventSource *glishBus = event.glishSource();
        cout<<"caught getstatus event at computecounter "<< counter<<endl;
//      GlishArray stat(counter);
//      glishBus->postEvent("computestatus",stat);
```

```
        return True;
}

Bool handlehalt(GlishSysEvent &event, void *) {
        cout <<"caught halt command - stopping computation at loop "<<counter<<endl;
        proceed  = False;
        return True;
        }

Bool computearray(GlishSysEvent &event, void *) {
        cout <<"caught compute event "<<endl;
        if (available) {
                available = false;
                proceed = True;
        }
        else {
                cout<<"but computearray routine already active!!"<<endl;
                return True;
        }
        Vector<Float> a(8192);
        a = 1.0;
// start a very looooooong computation ...
        for (Int i = 0; i < 2000000000; i++) {
                if (proceed) {
                        counter = i;
                        for (Int j = 0; j < 8192; j++)
                                        a(j) = a(j) * 1.0;

// but check for other incoming events every 5000th step in the cycle
                        if (i % 5000 == 0) {
                                GlishSysEventSource *glishBus = event.glishSource();
                                checkevents( *glishBus, 1 );
                        }
                }
                else {
                        cout<<"halting computation"<<endl;
                        break;
                }
        }
```

```
// callback is finished so reset available flag
        cout<<"resetting available"<<endl;
        available = True;
        return True;
}

int main(int argc, char **argv) {
        cout<<"simulated threads client started"<<endl;
        GlishSysEventSource glishStream(argc, argv);

//  define callback event handlers
        glishStream.addTarget(computearray, "compute");
        glishStream.addTarget(handlehalt, "halt");
        glishStream.addTarget(handlestatus, "getstatus");

        glishStream.loop();
}
```

Note that the

`computearray`

callback includes a test that prevents a user from trying to have two versions of this function in action simultaneously. Otherwise things could get rather confusing!

A sample of interaction taking place when running this variant of the program follows.

```
a:=client("./threads")
glish version 2.6.
- simulated threads client started
a->compute()
- caught compute event
a->getstatus()
- caught getstatus event at computecounter 50000
- a->compute()
- caught compute event
but computearray routine already active!!
- a->halt()
- caught halt command - stopping computation at loop 165000
```

```
halting computation
resetting available
- a->getstatus()
- caught getstatus event at computecounter 165000
- a->compute()
- caught compute event
- a->halt()
- caught halt command - stopping computation at loop 50000
halting computation
resetting available
- exit
```

0.6 Parallel Processing with AIPS++ and glish

With the advent of cheap PC-based clusters running the Linux operating system (Beowulfs) parallel computing is available for the masses. Two message passing systems that have been used to implement parallel processing on Linux clusters are PVM and MPI. The 'Linux Parallel Processing HOWTO' by Hank Dietz contains examples of how the following piece of sequential c code, which computes the value of PI by summing the area under x squared, can be converted into an 'embarrassingly parallel' application by using the PVM or MPI message passing systems.

```
/************************************/
#include <stdlib.h>
#include <stdio.h>
main(int argc, char**argv)
{
   register double width, sum;
   register int   intervals, i;

   /* get the number of intervals */
   intervals = atoi(argv[1]);
   width = 1.0 / intervals;

   /* do the computation */
   sum = 0;
   for (i = 0; i < intervals; ++i) {
      register double x = (i + 0.5) * width;
      sum += 4.0 /(1.0 + x * x);
```

```
   }
   sum *= width;
   printf("estimate of pi is %f\n",sum);
   return(0);
}
/**********************************/
```

This algorithm easily yields an "embarrassingly parallel" application as the number of intervals can be split up and assigned to different processors.

We shall show how a combination of a *Glish* script acting as a master controller together with a *Glish* client that does the computations, yields exactly the same result as does a PVM or MPI based program. Consequently, if you have learned the *Glish* message passing system, you can do most things that PVM or MPI can do. In fact, in many ways it is easier to handle and decode *Glish* messages, especially if you use the AIPS++ wrapper classes to decode the contents of messages.

Here is the resulting *Glish* client:

---

parallelclient.cc

```
#include <aips/Glish.h>
#include <aips/iostream.h>

// callback function to decode the computation parameters and do the calculation
Bool computeinterval(GlishSysEvent &event, void *) {
   GlishSysEventSource *glishBus = event.glishSource();
   GlishValue glishVal = event.val();
// check that argument is a record
   if (glishVal.type() != GlishValue::RECORD) {
      if (glishBus->replyPending())
         glishBus->reply(GlishArray(False));
         return True;
   }

// get parameters - starting position, number of intervals and
// number of processors - from the data record associated with the
// incoming event
   GlishRecord glishRec = glishVal;
   Int startpos;
```

```
   if (glishRec.exists("start")) {
      GlishArray tmp;
      tmp = glishRec.get("start");
      tmp.get(startpos);
   }
   Int intervals;
   if (glishRec.exists("intervals")) {
      GlishArray tmp;
      tmp = glishRec.get("intervals");
      tmp.get(intervals);
   }

   Int processors;
   if (glishRec.exists("numprocessors")) {
      GlishArray tmp;
      tmp = glishRec.get("numprocessors");
      tmp.get(processors);
   }

// now do the computation
   register double width, sum;
   width = 1.0 / intervals;
   sum = 0;
   for (Int i = startpos; i < intervals; i+=processors) {
      register double x = (i + 0.5) * width;
      sum += 4.0 /(1.0 + x * x);
   }
   sum *= width;

// post the result back to the controlling glish script
        GlishRecord record;
        record.add("pisum",sum);
        glishBus->postEvent("integration",record);

   return True;
}


int main(int argc, char **argv)
{
```

```
   GlishSysEventSource glishStream(argc, argv);
   cout<<"client starting"<<endl;

//  define callback event handler
   glishStream.addTarget(computeinterval, "compute");

// send message that the client started OK
        glishStream.postEvent("initialized","dummy");

// loop
   glishStream.loop();
}
```

---

and here is an initial example of a simple *Glish* DO script that farms out processing to multiple versions of the above client running on different machines in a cluster.

---

parallel.g

```
const demoparallel := function() {

# initialize necessary variables
   private := [=]
   public := [=]

   private.parallelclient := [=]
   private.numparallelclients := 0
   private.wheneverset := F
   private.numdone := 0
   private.pisum := 0

# open 'hosts' file; then read in and start clients
   hosts := ["slave1", "slave2", "slave3", "slave4", "slave5", "slave6"]
   numhosts := len(hosts)
   taskname := "./parallelclient"
   for (i in 1:numhosts) {
      hostname := hosts[i]
      private.numparallelclients := private.numparallelclients + 1
      print 'launching client on host ', hostname
```

```
      private.parallelclient[private.numparallelclients] :=
             client(taskname, host=hostname)
      await private.parallelclient[private.numparallelclients]->initialized
      print 'parallel client started'
   }

   const public.computeparallel := function(numintervals=40) {
# send each client information needed to do computation
      wider private
      private.pisum := 0
      private.numdone := 0
      parallelinfo := [=]
      parallelinfo.intervals := numintervals
      parallelinfo.numprocessors := private.numparallelclients

      for (i in 1:private.numparallelclients) {
         whenever private.parallelclient[i]->integration do {
            print $name, $value
            private.pisum := private.pisum + $value.pisum
            private.numdone := private.numdone + 1
            print 'numdone pisum = ', private.numdone, private.pisum
            if (private.numdone == private.numparallelclients)
               print 'computed value of pi ', private.pisum
         }
         parallelinfo.start := i - 1
         private.parallelclient[i]->compute(parallelinfo)
      }
      private.wheneverset := T
      return T
   }
   return public
}


a := demoparallel()
a.computeparallel()
```

The result of running this script (the

```
waiting for daemon ...
```

messages appear when the system is waiting for a glishd daemon to get started on the foreign machine) is

```
launching client on host  slave1
waiting for daemon ...
client starting
parallel client started
launching client on host  slave2
waiting for daemon ...
client starting
parallel client started
launching client on host  slave3
waiting for daemon ...
client starting
parallel client started
launching client on host  slave4
waiting for daemon ...
client starting
parallel client started
launching client on host  slave5
waiting for daemon ...
client starting
parallel client started
launching client on host  slave6
waiting for daemon ...
client starting
parallel client started
glish version 2.6.
- integration [pisum=0.561413]
numdone pisum =  1 0.561413
integration [pisum=0.553083]
numdone pisum =  2 1.1145
integration [pisum=0.544517]
numdone pisum =  3 1.65901
integration [pisum=0.535739]
numdone pisum =  4 2.19475
integration [pisum=0.477397]
numdone pisum =  5 2.67215
integration [pisum=0.469495]
numdone pisum =  6 3.14164
```

```
computed value of pi  3.14164
```

Of course, in our toy example, the latency involved in starting up the client processing tasks, far exceeds any benefits gained by doing parallel processing. So in actual parallel tasks, the amount of time spent doing computational work should exceed the amount of time spent doing task instantiation and message passing.

The above example still contains a REALLY BAD bug. When doing parallel and distributed processing we must often handle events coming back from distributed clients by means of a whenever statement rather than the request/reply mechanism used in the matrix computation example in which the *Glish* script communicated with only one client.

The public.computeparallel function contains a whenever statement that will get executed every time public.computeparallel is called. As was pointed out earlier, this is a Bad Thing. So if, after one call of a.computeparallel(). I decide to reissue this call using a non-default number of intervals, .e.g

```
a.computeparallel(200)
```

I get:

```
- a.computeparallel(200)
integration [pisum=0.534435]
numdone pisum =  1 0.534435
integration [pisum=0.534435]
numdone pisum =  2 1.06887
integration [pisum=0.532752]
numdone pisum =  3 1.60162
integration [pisum=0.532752]
numdone pisum =  4 2.13437
integration [pisum=0.521085]
numdone pisum =  5 2.65546
integration [pisum=0.521085]
numdone pisum =  6 3.17655
computed value of pi  3.17655
integration [pisum=0.519435]
numdone pisum =  7 3.69598
integration [pisum=0.519435]
numdone pisum =  8 4.21542
integration [pisum=0.517777]
```

```
numdone pisum =  9 4.73319
integration [pisum=0.517777]
numdone pisum =  10 5.25097
integration [pisum=0.51611]
numdone pisum =  11 5.76708
integration [pisum=0.51611]
numdone pisum =  12 6.28319
```

!!!!

You want to avoid having a whenever executed twice. So we recast the above script in the form:

---

goodparallel.g

```
const demoparallel := function() {

# initialize necessary variables
   private := [=]
   public := [=]

   private.parallelclient := [=]
   private.numparallelclients := 0
   private.pisum := 0
   private.numdone := 0

# open 'hosts' file; then read in and start clients
   hosts := ["slave1", "slave2", "slave3", "slave4", "slave5", "slave6"]
   numhosts := len(hosts)
   taskname := "./parallelclient"
   for (i in 1:numhosts) {
      hostname := hosts[i]
      private.numparallelclients := private.numparallelclients + 1
      print 'launching client on host ', hostname
      private.parallelclient[private.numparallelclients] :=
             client(taskname, host=hostname)
      await private.parallelclient[private.numparallelclients]->initialized
      print 'parallel client started'
      whenever private.parallelclient[private.numparallelclients]->integration do {
         print $name, $value
```

```
         private.pisum := private.pisum + $value.pisum
         private.numdone := private.numdone + 1
         if (private.numdone == private.numparallelclients)
            print 'computed value of pi ', private.pisum
      }
   }

   const public.computeparallel := function(numintervals=40) {
# send each client information needed to do computation
      wider private
      private.pisum := 0
      private.numdone := 0
      parallelinfo := [=]
      parallelinfo.intervals := numintervals
      parallelinfo.numprocessors := private.numparallelclients
      for (i in 1:private.numparallelclients) {
         parallelinfo.start := i - 1
         private.parallelclient[i]->compute(parallelinfo)
      }
      return T
   }
   return public

}

a := demoparallel()
a.computeparallel()
```

The "whenever" block has been migrated up into the constructor section of the distributed object, which can only be processed once, when the object is first created.

The latency mentioned above means that the startparallel function takes quite a long time to complete. This allows us to point out an interesting behaviour of subsequences.

We recast the above sequential *Glish* script into the style of a subsequence, as follows:

---

parallelsubsequence.g

```
demoparallel := subsequence()
{

# initialize necessary variables

   private.pisum := 0
   private.numdone := 0

   private := [=]
   private.parallelclient := [=]
   private.numparallelclients := 0
   private.ready := F
   private.docomputations := function(numintervals) {
      parallelinfo := [=]
      parallelinfo.intervals := numintervals
      parallelinfo.numprocessors := private.numparallelclients
      for (i in 1:private.numparallelclients) {
         parallelinfo.start := i - 1
         private.parallelclient[i]->compute(parallelinfo)
      }
   }

   const startparallel := function() {
      wider private
      print 'please wait for system has been initialized message'
      print 'before sending compute event'

      hosts := ["slave1", "slave2", "slave3", "slave4", "slave5", "slave6"]
      numhosts := len(hosts)
      taskname := "/home/beowulf1/twillis/bin/parallel"
      for (i in 1:numhosts) {
# use split function to delete endline
         private.numparallelclients := private.numparallelclients + 1
         hostname := hosts[i]
         print 'launching client on host ', hostname
         private.parallelclient[private.numparallelclients] :=
                client(taskname, host=hostname)
         await private.parallelclient[private.numparallelclients]->initialized
         whenever private.parallelclient[private.numparallelclients]->integration do {
            print $name, $value
```

```
            private.pisum := private.pisum + $value.pisum
            private.numdone := private.numdone + 1
            if (private.numdone == private.numparallelclients)
               print 'computed value of pi ', private.pisum
         }
         print 'parallel client started'
      }
      print 'system has been initialized - ready to compute'
      return T
   }

   const computeparallel := function(numintervals=40) {
# send each client information needed to do computation
      wider private
      private.pisum := 0
      private.numdone := 0
      print 'in computeparallel numintervals = ',numintervals
      private.docomputations(numintervals)
   }

# define some event handlers
      whenever self->start do {
      print 'received start'
      startparallel()
      }
      whenever self->compute do {
      print 'received compute command'
      computeparallel($value)
      }

}
# end of subsequence definition
# now, use it

a:= demoparallel()
print a
a->start()
#print 'a sent start event'
a->compute(50)
#print 'a sent compute event'
```

As it turns out this is also an exceptionally bad piece of code. If you run it you will get something like:

```
[*agent*=<agent>]
glish version 2.6.
- received start
please wait for system has been initialized message
before sending compute event
launching client on host  slave1
waiting for daemon ...
received compute command
in computeparallel numintervals =  50
client starting
parallel client started
launching client on host  slave2
waiting for daemon ...
integration [pisum=3.14163]                 !!!!!!!!!!!!!
client starting
parallel client started
launching client on host  slave3
waiting for daemon ...
client starting
parallel client started
launching client on host  slave4
waiting for daemon ...
client starting
parallel client started
launching client on host  slave5
waiting for daemon ...
client starting
parallel client started
launching client on host  slave6
waiting for daemon ...
client starting
parallel client started
system has been initialized - ready to compute
```

and that's it. What has happened is the following. When the script is sent the first event

```
a->start()
```

, the *Glish* interpreter starts to process the

```
startparallel()
```

function and launch the clients. The subsequence is then sent the event

```
a->compute()
```

and the glish interpreter then begins to respond to that event even though it has not completed doing all the work in response to the

```
a->start()
```

event. At the moment the

```
a->compute()
```

event is received, the interpreter has only completed the launch of 1 client, so it thinks there is only one client available to do the actual computing. Here we really see the *Glish* interpreter behaving like a multi-threaded application.

In the above case, we really only want the compute phase of the subsequence to start after all clients have been launched and we know how many are available. We do that by just making the startparallel function part of the basic constructor part of the object and not doing the construction in response to a start event. Having the possibility of a separate start event is a bad idea in any case as there's nothing to stop a user from sending two start events, and that could wreck havoc with multiple processing of those whenevers. Here is a better subsequence:

---

goodparallelsubsequence.g

```
demoparallel := subsequence() : [reflect=T]
{

# constructor
   private := [=]

   private.pisum := 0
   private.numdone := 0
```

```
   private.parallelclient := [=]
   private.numparallelclients := 0

   print 'please wait for system has been initialized message'
   print 'before sending compute event'

   hosts := ["slave1", "slave2", "slave3", "slave4", "slave5", "slave6"]
   numhosts := len(hosts)
   taskname := "./parallelclient"
   for (i in 1:numhosts) {
      private.numparallelclients := private.numparallelclients + 1
      hostname := hosts[i]
      print 'launching client on host ', hostname
      private.parallelclient[private.numparallelclients] := client(taskname, host=hostname)
      await private.parallelclient[private.numparallelclients]->initialized
      whenever private.parallelclient[private.numparallelclients]->integration do {
         print $name, $value
         private.pisum := private.pisum + $value.pisum
         private.numdone := private.numdone + 1
         if (private.numdone == private.numparallelclients)
            print 'computed value of pi ', private.pisum
      }
      print 'parallel client started'
   }

   private.docomputations := function(numintervals) {
      parallelinfo := [=]
      parallelinfo.intervals := numintervals
      parallelinfo.numprocessors := private.numparallelclients
      for (i in 1:private.numparallelclients) {
         parallelinfo.start := i - 1
         private.parallelclient[i]->compute(parallelinfo)
      }
   }
   private.ready := T
   print 'system has been initialized - ready to compute'

   const computeparallel := function(numintervals=40) {
# send each client information needed to do computation
```

```
      wider private
      private.pisum := 0
      private.numdone := 0
      print 'in computeparallel numintervals = ',numintervals
      if (private.ready) {
         private.docomputations(numintervals)
      }
      else
         print 'system not yet initialized -- waiting'
      return T
   }

      whenever self->compute do {
      print 'received compute command'
      computeparallel($value)
      }

}
# end of subsequence definition
# now, use it

a:= demoparallel()
print a
a->compute(50)
#print 'a sent compute event'
```

Running this version of the script will give correct behaviour.

```
...
system has been initialized - ready to compute
[*agent*=<agent>]
glish version 2.6.
- received compute command
in computeparallel numintervals =  50
integration [pisum=0.56699]
integration [pisum=0.560064]
integration [pisum=0.513393]
integration [pisum=0.506979]
integration [pisum=0.500433]
integration [pisum=0.493768]
```

```
computed value of pi  3.14163
```

0.7 Pipeline parallel processing

Another type of parallelism is pipeline processing. Here a number of tasks do partial processing of data and then forward the partially processed result to another processing client down the pipeline for further processing.

A task familiar to radio astronomers is the processing of spectra from correlator lags to spectra. Three steps in the processing are

1) multiply the observed lags by a window function 2) fft the lags 3) subtract off a baseline to obtain the computed spectrum.

We shall show how these tasks could be joined together in a data processing pipeline.

Here is a *Glish* DO script that sets up a pipeline of three clients running on three different computers to do the processing. "pipe1" does the windowing, "pipe2" does the fft, and "pipe3" does a simple baseline subtraction:

---

pipeline.g

```
const demopipeline := function() {

# initialize necessary variables
   private := [=]
   public := [=]

   private.parallelclient := [=]
   private.numparallelclients := 0
   private.pipelineconstructed := F

# open 'hosts' file; then read in and start clients
   hosts := ["slave1", "slave2", "slave3"]
   numhosts := len(hosts)
   tasks := ["pipe1", "pipe2", "pipe3"]
   taskpreface := "/home/beowulf1/twillis/aips++/code/hia/apps/pipeline/"
   for (i in 1:numhosts) {
# use split function to delete endline
      hostname := hosts[i]
      taskend := tasks[i]
```

```
      taskname := spaste(taskpreface,taskend)
      private.numparallelclients := private.numparallelclients + 1
      private.parallelclient[private.numparallelclients] :=
           client(taskname, host=hostname)
      await private.parallelclient[private.numparallelclients]->initialized
   }

# handle event associated with final output by printing it.
   whenever private.parallelclient[3]->finalspectrum do {
      outputspectrum := $value.reducedspectrum
      print 'outputspectrum ',outputspectrum
   }
# instead of using these 'whenevers' to pass events, use the 'link' command
# below
#               whenever private.parallelclient[1]->tostep2 do
#                       private.parallelclient[2]->[$name]($value)
#               whenever private.parallelclient[2]->tostep3 do
#                       private.parallelclient[3]->[$name]($value)

# set up pipeline links
   link private.parallelclient[1]->tostep2 to private.parallelclient[2]->*
   link private.parallelclient[2]->tostep3 to private.parallelclient[3]->*

# pause here to make sure links get set before events start flowing through them
   d := client("timer", "-oneshot 2.0")
   await d->ready

   const public.dopipeline := function(speclength = 1024) {
      wider private

# create a block spectrum
      spectrumarray := array(0.0,speclength)
      specbegin := speclength / 8
      specend := speclength - speclength / 8
      spectrumarray[specbegin:specend] := 1.0

# send 5 identical block spectra through the pipeline
      specrecord := [=]
      specrecord.spectrum := spectrumarray
      for (i in 1:5) {
```

```
      print 'sending event ',i
      private.parallelclient[1]->inputdata(specrecord)
    }
    return T
  }
  return public

}

a := demopipeline()
a.dopipeline(8)
```

There are a couple of things to note here. First we want to transfer data between the pipeline processing clients as fast as possible. We do this by setting up direct channels (pipes or sockets) between the clients by use of the glish "link" command as ...

```
# set up pipeline links
link private.parallelclient[1]->tofft to private.parallelclient[2]->*
link private.parallelclient[2]->toremove to private.parallelclient[3]->*
# pause here to make sure links get set before events start flowing
# through them
   d := client("timer", "-oneshot 2.0")
   await d->ready
```

Secondly, note the use of the *Glish* timer client to inject a 2 sec pause after the link command has been issued. This pause allows the *Glish* system to set up the link connections before any events flow through them. If you do not force a pause, but start sending events right away, it is possible that the link will not be set up in time and you will get dropped events.

Here is the code for the three clients making up the pipeline.

---

pipe1.cc

```
#include <aips/Glish.h>
#include <aips/Arrays/Vector.h>
#include <aips/iostream.h>

Bool applyweight(GlishSysEvent &event, void *) {
```

```
   cout <<"in pipeline 1 applyweight"<<endl;
   GlishSysEventSource *glishBus = event.glishSource();
   GlishValue glishVal = event.val();
// check that argument is a record
   if (glishVal.type() != GlishValue::RECORD) {
      if (glishBus->replyPending())
         glishBus->reply(GlishArray(False));
         return True;
   }
   GlishRecord glishRec = glishVal;
   Vector <Float> spectrum;
   if (glishRec.exists("spectrum")) {
      GlishArray tmp;
      tmp = glishRec.get("spectrum");
      tmp.get(spectrum);
   }

   IPosition speclength = spectrum.shape();
   Int nelem = speclength(0);


// very simple - apply a  uniform weight of 1 to all points
   Vector <Float> weight(nelem);
   weight = 1.0;
   for (Int i = 0; i < nelem; i ++)
      spectrum(i) = spectrum(i) * weight(i);

   GlishRecord record;
   record.add("weightedspectrum",spectrum);
   glishBus->postEvent("tostep2", record);
   cout <<"in pipeline 1 posted event tostep2"<<endl;

   return True;
}

int main(int argc, char **argv) {
   cout<<"pipe 1 started"<<endl;
   GlishSysEventSource glishStream(argc, argv);

//  define callback event handlers
```

```
   glishStream.addTarget(applyweight, "inputdata");

// send message that the client started OK
        glishStream.postEvent("initialized","dummy");

   glishStream.loop();
}
//////////////////////// end  pipe1.cc ////////////////////

//////////////////////// begin  pipe2.cc ////////////////////
#include <aips/Glish.h>
#include <aips/Arrays/Vector.h>
#include <aips/Mathematics/FFTPack.h>
#include <aips/iostream.h>

Bool dofft(GlishSysEvent &event, void *) {
   cout<<"in pipeline 2 dofft"<<endl;
   static Bool First = True;
   static Vector<Float> wsave;
   Float* Wsave;

   GlishSysEventSource *glishBus = event.glishSource();
   GlishValue glishVal = event.val();
// check that argument is a record
   if (glishVal.type() != GlishValue::RECORD) {
      if (glishBus->replyPending())
         glishBus->reply(GlishArray(False));
         return True;
   }
   GlishRecord glishRec = glishVal;
   Vector <Float> spectrum;
   if (glishRec.exists("weightedspectrum")) {
      GlishArray tmp;
      tmp = glishRec.get("weightedspectrum");
      tmp.get(spectrum);
   }

   IPosition speclength = spectrum.shape();
   Int nelem = speclength(0);
   if (First) {
```

```
// initialize stuff for FFTPACK fft
      uInt wsavelen = 4 * nelem + 104;
      wsave.resize(wsavelen);
      Wsave = &wsave(0);
      FFTPack::costi(nelem, Wsave);
      First = False;
   }
   Float *Spectrum = &spectrum(0);
// cout << "before dofft: spectrum "<< spectrum <<endl;
// do real-to-real cosine transform
   Wsave = &wsave(0);
   FFTPack::cost(nelem, Spectrum, Wsave);

// normalize
   for (Int i = 0; i <nelem; i++)
            spectrum(i) = spectrum(i) / (2 * (nelem - 1));

   GlishRecord record;
   record.add("fftdspectrum",spectrum);
   glishBus->postEvent("tostep3", record);
   cout<<"in pipeline 2 posted tostep3"<<endl;

   return True;
}

int main(int argc, char **argv) {
   cout<<"pipe 2 started"<<endl;
   GlishSysEventSource glishStream(argc, argv);

//  define callback event handlers
   glishStream.addTarget(dofft, "tostep2");

// send message that the client started OK
        glishStream.postEvent("initialized","dummy");


   glishStream.loop();
}
/////////////////////// end pipe2.cc ////////////////////
```

```
/////////////////////// begin  pipe3.cc ////////////////////
#include <aips/Glish.h>
#include <aips/Arrays/Vector.h>
#include <aips/Mathematics/FFTPack.h>
#include <aips/iostream.h>

Bool removebaseline(GlishSysEvent &event, void *) {
   cout <<"in pipeline 3 remove baseline "<<endl;

   GlishSysEventSource *glishBus = event.glishSource();
   GlishValue glishVal = event.val();
// check that argument is a record
   if (glishVal.type() != GlishValue::RECORD) {
      if (glishBus->replyPending())
         glishBus->reply(GlishArray(False));
         return True;
   }
   GlishRecord glishRec = glishVal;
   Vector <Float> spectrum;
   if (glishRec.exists("fftdspectrum")) {
      GlishArray tmp;
      tmp = glishRec.get("fftdspectrum");
      tmp.get(spectrum);
   }

   IPosition speclength = spectrum.shape();
   Int nelem = speclength(0);

// In this exercise, average first and last 1/8 of spectrum and
// remove this averaged value

   Int toaverage = nelem / 8;
   Float sum = 0;
   for (Int i = 0; i < toaverage; i++ )
      sum = sum + spectrum(i);
   for (Int i = nelem-1; i > nelem - toaverage; i-- )
      sum = sum + spectrum(i);
   Float baseline = sum / (2 * toaverage);
   for (Int i = 0; i < nelem; i++ )
      spectrum(i) = spectrum(i) - baseline;
```

```
   GlishRecord record;
   record.add("reducedspectrum",spectrum);
   glishBus->postEvent("finalspectrum", record);
   cout <<"in pipeline 3 posted finalspectrum "<<endl;

   return True;
}

int main(int argc, char **argv) {
   cout<<"pipe 3 started"<<endl;
   GlishSysEventSource glishStream(argc, argv);

//  define callback event handlers
   glishStream.addTarget(removebaseline, "tostep3");

// send message that the client started OK
        glishStream.postEvent("initialized","dummy");


   glishStream.loop();
}
```

0.8 A Link is not just a Faster Whenever

In our discussion of the pipelining technique we replaced

```
whenever private.parallelclient[1]->tostep2 do
     private.parallelclient[2]->[$name]($value)
whenever private.parallelclient[2]->tostep3 do
     private.parallelclient[3]->[$name]($value)
```

by

```
link private.parallelclient[1]->tostep2 to private.parallelclient[2]->*
link private.parallelclient[2]->tostep3 to private.parallelclient[3]->*
```

in order to bypass the *Glish* interpreter and send events quickly and directly between clients. You should be aware that replacing a whenever by a link in the above way can have a subtle impact on the way a large scale distributed application behaves.

This happens because direct communication between *Glish* clients is essentially synchronous. There is essentially no buffer space associated with a link (some tests show that there would appear to be a buffer size of 8192 words, but this is pretty well useless for applications sending a lot of data.) *Glish* clients also have one thread of control so they can only respond to incoming messages one at a time.

You will notice synchronized message passing in the following situation. If client a sends an event containing a large amount of directly to client b, client b transfers control to the callback function associated with this event and begins handling the data. If client b is finishes processing this data before client a sends the next data event, there is no problem. However if client a produces and sends the next data event before client b is finished processing the first one there is a problem. In this case client b is still processing data in the callback function, and it will not be able to immediately handle the event posted by client a at the moment client a posts the event. In this case client a blocks at the point it is trying to post the event. It cannot proceed further until client b finishes processing the data from the first event and returns to the event loop. So the rate at which client a can post messages to client b is limited by the rate at which client b can handle the messages. In a case where you might have four clients sending data along a pipeline, the rate at which each client in the pipeline will handle the data is directly dependent on the rate at which the slowest client can work.

The *Glish* interpreter behaves differently. If the *Glish* interpreter sends a message to a client, but that client is too busy to respond to the message, the *Glish* interpreter does not block, but remains responsive to other incoming events. It will attempt to do the write at some later time when the client becomes responsive. The interpreter behaves as a multi-threaded application stuck in a single thread.

So what this means is that if client a sends its events via a "whenever" to client b, the interpreter will store up the events it receives from a and will send them to client b at the rate client b can digest them. In this case client a may run to completion long before client b does. However if client a sends too many large data events to client b via the interpreter, the interpreter can essentially "go out to lunch" if it cannot get the events forwarded to and digested by the second client fast enough.

You can also recreate an illusion of asynchronous communication between two *Glish* clients that process data at different rates by putting a buffer client between the two tasks. The buffer client just contains a list.

When the first client posts data it sends an event to the buffer task which just adds the data to one end of the list.

At the same time this buffer task could be observing the second task. When that task sends an event to the buffer task that it is available to process data the buffer task gets a data record off the other end of the list, deletes the element from the list, and posts the data to the second client. The list acts as a buffer between the two tasks. Of course, at some point, the faster task must stop generating data or you will run out of memory.

0.9 Distributed and Parallel Programming with Glish Scripts

The previous sections have focussed on the use of compiled c++ *Glish* clients to do the bulk of the work in parallel and distributed processing. *Glish* scripts have been used to coordinate and pass out the work to compiled clients. However, a little known property of *Glish* scripts is that they themselves can be used to do parallel or distributed processing. We return to our example of section 0.6 and show how the computation of PI can be reformulated into a parallel processing application that makes use of *Glish* scripts alone.

We first rewrite our 'parallelclient.cc' c++ client code as a *Glish* client script that can be started remotely by the controlling *Glish* script 'goodparallel.g'. To do this we must make use of the `is_script_client` field associated with the `system` record.

Here is the resulting *Glish* script client:

---

parallelclient.g

```
# glish function to extract the computation parameters and do the calculation
computeinterval := function (parameters) {

# first obtain parameters - starting position, number of intervals and
# number of processors
  startpos := parameters.start
  intervals := parameters.intervals
  processors := parameters.numprocessors
# now do the computation
  width := 1.0 / intervals
```

```
  sum := 0
  i := startpos
  while (i <= intervals-1) {
     x := (i + 0.5) * width
     sum := sum + 4.0 /(1.0 + x * x)
     i := i + processors
  }
  sum := sum * width;
  return sum
} # end of function definition

# here is the equivalent of the main function in the c++ client
if (system.is_script_client) {
  print "script client starting"
  script->initialized("dummy")
  whenever script->compute do {
    record := [=]
    record.pisum := computeinterval($value)
    script->integration(record);
  }
}
```

In our example, running this script as a script client causes the global `script` variable to become an `agent` capable of receiving or sending *Glish* messages. We can start up the client *Glish* scripts by replacing the definition of the client task to be started in the following line of code from goodparallel.g

```
taskname := "./parallelclient"
```

by

```
taskname := "glish ./parallelclient.g"
```

It is important to note that the first parameter fed to the `client` function call in this case must `glish`. Otherwise the client scripts will not run.

0.10 Further Reading

There is a lot of further introductory material available.

- *Getting Started with Glish* by Rick Fisher describes the most important features of Glish. For definitive details, see *Glish 2.7 Manual* located on the *Glish* Web pages
- Reference material on all tools and functions is found in the (massive!) *AIPS++* User Reference Manual