Note 188: Proposed Initial Standard Computer Configuration for AIPS++

B.E. Glendenning
bglenden@nrao.edu

March 25, 1996

1 Introduction

Now that AIPS++ is within about a year of initial β-release, it is time to describe a standard hardware (mostly) and software system that we expect end-users to have to be able to effectively run AIPS++. It must be emphasized that this paper describes the first machines to run AIPS++, not to preclude future ones.

I have chosen to adopt the simple expedient of declaring that AIPS++ should perform well for problems of normal size on a modern Unix-based workstation costing about $5,000. The exemplar of such a machine is a:

1. Sun Sparcstation 4
2. 64M of main memory
3. A few gigabytes of disk storage

A standard problem is considered to be a few hundred thousand visibility samples which result in $\sim 1024 \times 1024$ continuum images, or $\sim 512 \times 512 \times 128$ spectral line cubes. Larger problems should run, but standard sized problems should fit comfortably on machines of this size.

This paper is couched in synthesis processing terms, it would be interesting to make a similar analysis for the single-dish case.

2 Operating System and Other Software

External committments require that AIPS++ run on HP and Dec alpha hardware. In addition, development takes place largely on Sun Sparc machines running Solaris.[1] These three platforms will be represented in our initial releases.

[1] It is my opinion that we need not run on SunOS 4.x.

While those platforms must have adequate host libraries (X-Windows and the like), the dominant problem is apt to be in the available C++ compilers. Our first choice in each case is to use the vendor's native compiler. The fall-back in each case is to attempt to use g++ or a third-party CFront compiler.

Danger 1 We cannot find a compiler capable of compiling and linking AIPS++ on one of our target machines.

3 Processor Speed

There is little to be said about this item: AIPS++ must run standard sorts of problems in a time comparable to other packages on the *same* hardware. Users are apt to be more tolerant of slow execution of unique functionality, but not forever.

Danger 2 The software must run the standard problems in the standard amount of time.

4 Disk Space

4.1 Software Installation

Disk space is relatively inexpensive, so the amount of disk space required to hold an AIPS++ installation should not be a serious obstacle. (The effect on memory use is serious and is discussed later). It will be embarrassing if it takes too much disk space however (more than, say, a few hundred Megabytes).

4.2 Data Sizes

Images should not be intrinsically much larger in AIPS++ than in any other system since the size is so dominated by the pixel data. (That is, the ratio of data to "book-keeping" is inherently high). Moreover, the scaled storage manager allows us to transparently use scaled 2-byte integers if we choose to.

The main danger comes about because our Measurement-Equation based processing is inherently for all polarizations at once. The software must be able to store on disk only those polarizations which are required, not insist on always writing 4-polarization images. Similar arguments occur for visibility data.

Danger 3 Only store the desired (or required) polarizations in persistent data.

The raw MeasurementSet (visibility data) is more susceptible to general data bloat since the ratio of book-keeping (indices into subtables, for example) to correlation data might be high, particularly for continuum data. Moreover optimizations such as assuming that a single number can represent the weight

of an entire visibility spectrum are made in other packages and will be expected in AIPS++[2]. These sorts of problems should be solvable by using and creating appropriate storage managers.

Danger 4 Generality in the MeasurementSet might cause data bloat for the "usual" cases.

We certainly can't afford an integral factor bloat in the raw data size. Even a 20% factor would be painful for large experiments that might already barely fit on a single disk.

5 Memory Use

Memory use can generally be split up between that required for dynamic allocations within the application, and that required for the binary executable itself.[3].

For the sake of discussion, suppose the user is doing some interactive fiddling and image display while a standard-sized background continuum imaging task is computing.

5.1 Memory Allocations

It seems likely that the workhorse imaging programs will require access to all polarizations at once. In "image" space, then the standard problem size is:

$$(1024 \times 1024) pixels/image \times 5 polns/pixel \times 4 bytes/poln = 20MB$$

The factor of 5 comes from requring 4 polarizations in the output (and input) images, but assuming that the beam is scalar. I also assume that we take advantage of symmetry, i.e. the Complex images can be 1/2 the size and transform into full-sized real images.

The size of the cache required to go through the MeasurementSet should be fairly small (equivalent to a few rows). The subtables are also generally small. So, the continuum case memory use is dominated by the image sizes. Inflating by 25% on general principles to allow for various other application work areas brings the size to 25M.

Danger 5 The Images must transform in-place (including the Complex to Real transformation). No in-memory Image "temporaries" can be required.

[2] One should be able to turn this assumption on and off.

[3] Space required for static allocations should be insignificant in a system with dynamic memory allocation

The spectral line problem is considerably cheaper, assuming only one plane needs to be kept in memory at a time.

Of course applications which do not require an entire image plane in memory at once could and should be coded to use a "sliding window." The Image iteration model should make this straightforward.

AipsView uses memory mapped I/O when the underlying FITS file is floating point, and the host architecture is big-endian. When this is not the case it will pull the entire image into memory, which is unacceptable for medium to large spectral line cubes. Normally however *AipsView* uses memory mapped I/O for the data, which should result in a buffer which is sized to be "nice" in the current processing environment. Pixmaps are however probably kept in local memory, so a couple of 1024×1024 images will result in about 2M of memory allocation.

Danger 6 *AipsView* can require that entire cubes be in memory in some circumstances.

Glish users will use memory directly in local variables, and in its external clients (e.g., the tableclient). There is nothing restricting how much memory the user wishes to stick in local variables. Moreover the tendency of Glish to promote from single to double precision can exacerbate these space problems. In practice, I expect that users will learn that bringing in whole Images, or entire column of visibility data, is usually a bad thing to do. They will either set an "xinc," or will iterate through all the data. I'll arbitrarily set the amount of allocations in Glish and its clients to 3M. This brings the amount of allocated memory to about 30M.

5.2 Executable Sizes

AipsView is about 1.4M (shared X libraries). Glish and a client or two should be 5M or less (shared Glish libraries). Imager is 10M.

While the Glish size seems a little large (statics?), Imager is clearly the largest offender. Presumably this is largely caused by template instantiations of similar functions.

Moreover, unless common library functionality is in a shared library, this bloat will add up linearly with the number of processes.[4]. Moreover, large executable sizes imply slow startup of the executables if they must be fetched over the network (for example, via NFS).

5.3 Discussion

The sum of the allocated memory estimates and executable size estimates is about 46.5M. Adding about 16M for the operating system takes us to 62.5, just

[4]On the other hand, putting infrequently used functions in a shared library makes things worse

under the amount of memory in our canonical machine.

While we can conclude that this standard problem would run reasonably well in our standard machine, a larger problem (e.g., $2k \times 2k$) would cause the standard machine to swap heavily.

Danger 7 Larger than standard problems push us over the limit required by the "canonical $5,000 computer."

6 Reccomendations

My main reccomendations are as follows:

1. Decide on a policy for application memory use. Do we want to require that "core" applications use as little memory as possible, *i.e.*, trade of programmer convenience for allowing larger problems to be run on smaller machines. While it seems that Image access is the driver, we should attempt to think of applications that require, for example, access to a large number of visibilities at the same time.

2. That AIPS++ adopt the "Standard $5,000 machine" as its goal.

3. That an individual be appointed to track progress and extant "Dangers," and to report on progress and prospects on a monthly timescale until running on the standard machine is a reality.

4. That developers routinely run new applications on standard sized problems on standard machines[5], and immediately report problems, performance and memory related particularly.

5. Monitor the probable size of an end-user installation.

6. Occasionally read standard continuum (particularly) and spectral line data sets into AIPS++ and compare the resultant visibility data-set sizes and size of the resultant images with those in other packages. Do this for datasets with and without polarization measurements.

7. Plan to modify *AipsView* so that it does I/O on the underlying image data rather than requiring it to (logically) be entirely within memory.

8. Investigate weakening Glish's promotion to double (and dcomplex). Investigate a way for users to discover which variables are using up all their space.

9. Investigate Glish's executable size — is the problem static functions or something else.

[5]Degradation for larger problems is also of interest.

10. Investigate AIPS++ library size, particularly trying to factor out common code in template instantiations.

11. Complete the implementation of shared libraries for "common" code.