Note on the Efficacy of Multi-threading in Obit

W. D. Cotton, October 1, 2008

Abstract—**This memo describes the implementation of multi-threading for increased efficiency of processing astronomical data. Tests on a high dynamic range, wide field data set problem using the Obit task Squint are presented. A variable number of threads are enabled with a good increase in performance up to at least 8 processors.**

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

I. Introduction

THE upcoming generation of radio interferometers (e.g. EVLA, ALMA, LOFAR, MeerKAT) will produce vastly more data than the current generation of interferometers. The speed of single processor computers appears to be nearing fundamental limits and the needed computing power will require harnessing multiple processors. One of the currently viable techniques for doing this is using multiple threads running on multiple processors/cores in a shared memory environment. This "multi-threading" technique allows using multiple processing units on a given problem. The following discusses an implementation of multi-threading of several compute intensive operations in Obit ([1], http://www.cv.nrao.edu/$\sim$bcotton/Obit.html).

II. Multi-threaded Software

The use of multiple asynchronous threads in a shared memory environment places fairly stringent requirements on the design of a software library. Care must be taken that memory potentially being accessed by other threads is not modified. When such modification is necessary, a locking mechanism such as mutexes is needed. Some libraries such as FFTW[1] for doing FFTs can be built to use multiple threads on multiprocessor/multi-core systems. Older libraries written before the widespread use of multiple threads may not be "thread-safe" and their usage must be limited to a single thread. Cfitsio[2] is an example of a widely used library for reading FITS files which is not thread-safe.

III. Multi-threading in Obit

Obit uses multiple native data types for the disk representation of data, including FITS. Since cfitsio is used to access FITS data, I/O to disk is limited to a single master thread. Most data processing in Obit uses the "strip mining" technique for access to potentially large data sets. This involves reading blocks of data into memory and performing the operation on the data file one block at a time. An example of this is forming an image from an interferometry UV dataset or a single disk OTF dataset. Each block of data is read, each data sample is multiplied by a griding function and accumulated onto a grid. Since operations of blocks of data already in memory do not involve I/O, they are candidates for multi-threaded processing.

In many operations using the strip mining technique, the operation over individual data samples can be independent of each other making it possible to divide up a buffer of data to be fed to multiple processing threads. Since processing a large dataset will involve many buffers of data, the overhead of starting and stopping many threads can be a significant overhead. The glib library adopted by Obit includes the gthreads library which has a solution to this problem. This involves "pools" of threads which are started and remain active until the pool is shut down. During this time, they may be used to execute a given routine multiple times with different data.

The model used is a master control thread which does the I/O and divides up the work into the pool of processing threads. Since a processing thread does not terminate when the routine it is executing returns, a "join" on the thread will hang forever and another means of indicating that the operation is completed is needed. For this, Obit uses a glib asynchronous message queue.

In spite of the effort to limit use of cfitsio to a single thread, it still may interfere with use of threads. The mechanism is not understood but under some circumstances, the combination of multi-threading and using cfitsio to access the data from FITS files causes the whole process to run VERY slowly as well as causing subtle (and probably erroneous) differences in the results.

In the initial implementation of threading in Obit, two compute intensive operations following the strip-mining model are converted to use multiple threads and these are discussed below. Use of multiple threads for the the inner loop of the "Clark" CLEAN was explored but no benefit was found for typical size operations. The two types of operations for which multi-threaded operation were implemented are described in the following.

A. Griding data for imaging

The griding of randomly sampled interferometric UV data or single disk OTF has the additional complication that the data are accumulated onto a grid. This explicitly adds a potential dependency between threads. In this case, each thread is given a grid(s) onto which to accumulate the data and when all data have been processed, the various thread grids are summed. This breaks the dependency between threads and is illustrated by the software fragments shown in the appendix. The griding of both interferometer and single dish data were implemented.

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

Manuscript received ; revised

[1]http://www.fftw.org

[2]http://heasarc.nasa.gov/docs/software.html

TABLE I
TIMING TESTS WITH 1,000 VIS PER THREAD

# threads	Run time (min)	Rel. Speed
1	42.45	1.00
2	22.62	1.87
4	13.18	3.22
6	9.33	4.55
8	8.40	5.05

B. Modeling the instrumental response to a sky model.

The operations in question here are calculating the instrumental response to a given model and either subtracting it from the data or dividing it into the data. These operations are used in a number of circumstances such as deconvolution and self–calibration. The potential dependencies in this case involve time variable models which may require thread specific models. One example of this is the VLA beam squint correction routines in which the gain in RR and LL for each CLEAN component is modified by the parallactic angle (hence time) variable antenna gain. For multi-threaded operations independent lists of CLEAN component gains are needed for individual threads. Threaded operations were implemented for normal interferometric sky model calculations (both "DFT" and "GRID" methods), time and space variable antenna gain and/or calibration, and sky model calculations of single dish data.

IV. TIMING TEST

To test the effectiveness of these techniques, a high dynamic range VLA test data set was used. This data is at 20 cm wavelength and has 3C84 (24 Jy) at the half power point of the antenna pattern. The VLA beam squint correction in Obit task Squint [2] was used in timing tests with variable numbers of threads. In these test, the griding of the data onto the 22 imaging facets used and the DFT model calculation for deconvolution and self–calibration were allowed the use of multiple threads. In addition to beam squint corrections, this test also involved auto–windowing, auto–centering of strong sources, phase and amplitude and phase self-calibration. Details of the data and processing are givenin [3]

The tests were performed on Brian Mason's computer, idl64, in Charlottesville. This machine has two quad core processors for a total of 8 processing elements and runs a 64–bit Red Hat Linux OS. The initial set of tests used buffer sizes of 1000 times the number of threads used. Input UV data, output images and scratch files used AIPS format to avoid the cfitio problem. The results of the timing tests are shown in Table I and Figure 1. This table gives the number of threads used, the total wall clock execution time and the ratio of the time used to that for a single thread.

The relative overhead cost of calling functions in threads depends on the amount of work done in each call. In order to test this, the size of the buffers used were increased to 10,000 times the number of threads and the tests rerun. The results are shown in Table II and Figure 1.

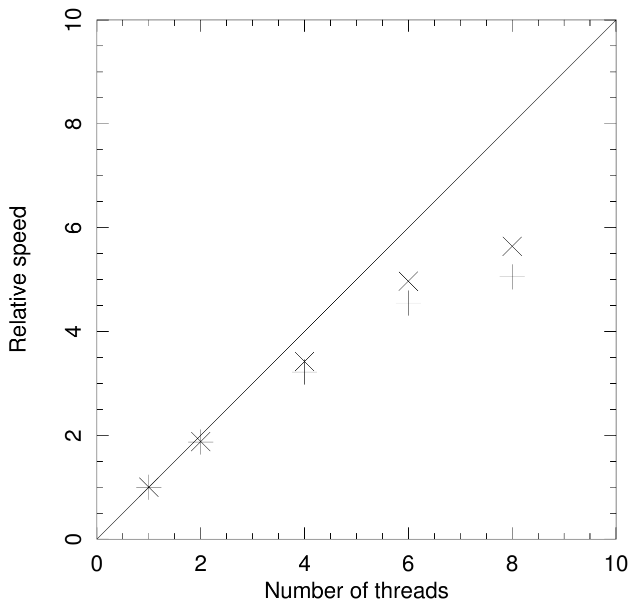

Fig. 1. Processing speed relative to a single processor. "+" symbols are for tests with 1,000 visibilities per thread; "x" symbols are for tests with 10,000 visibilities per thread. The line indicates complete utilization of the processors.

TABLE II
TIMING TESTS WITH 10,000 VIS PER THREAD

# threads	Run time (min)	Rel. Speed
1	42.78	1.00
2	22.73	1.88
4	12.51	3.42
6	8.60	4.97
8	7.58	5.64

V. DISCUSSION

The results given above demonstrate a significant improvement in the speed of processing of a fairly compute intensive problem up to at least 8 threads. With 8 threads the run time was reduced by a factor of 5.6 over the single thread run time. Increasing the buffer size made minor improvements to the performance for the larger number of threads but have no impact when small numbers of threads were used. The use of gthread thread pools seems to add little processing overhead and be good value for the slight additional complexity. This test demonstrates the potential to use multi-threaded processors for large radio astronomy computing problems.

The author would like to thank Brian Mason for the use of his computer for these tests and Darrell Schiebel for assistance on system related issues.

```
/* FT threaded function argument */
typedef struct {
  /* SkyModel with model components loaded (ObitSkyModelLoad) */
  ObitUVGrid *in;
  /* UV data set to model and subtract from current buffer */
  ObitUV       *UVin;
  /* First (1-rel) vis in uvdata buffer to process this thread */
  olong        first;
  /* Highest (1-rel) vis in uvdata buffer to process this thread  */
  olong        last;
  /* thread number, >0 -> no threading   */
  olong        ithread;
  /* Temporary gridding array for thread */
  ObitCArray  *grid;
} UVGridFuncArg;
```

Fig. 2. Thread call function argument structure

Appendix

The following software fragments are a detailed description of UV data griding from class ObitUVGrid.

References

[1] W. D. Cotton, “Obit: A Development Environment for Astronomical Algorithms,” *PASP*, vol. 120, pp. 439–448, 2008.
[2] J. Uson and W. D. Cotton, “Beam Squint and Stokes V with Off–axis Feeds,” *A&A*, vol. in press, 2008.
[3] W. D. Cotton and J. Uson, “Pixelization and Dynamic Range in Radio Interferometry,” *A&A*, vol. in press, 2008.

```
/**
 * Read a UV data object, applying any shift and accumulating to grid.
 * Buffering of data will use the buffers as defined on UVin
 * ("nVisPIO" in info member).
 * The UVin object will be closed at the termination of this routine.
 * Requires setup by #ObitUVGridCreate.
 * The gridding information should have been stored in the ObitInfoList on in:
 * \li "Guardband" OBIT_float scalar = maximum fraction of U or v range allowed in grid.
 *             Default = 0.4.
 * \li "MaxBaseline" OBIT_float scalar = maximum baseline length in wavelengths.
 *             Default = 1.0e15.
 * \li "startChann" OBIT_long scalar = first channel (1-rel) in uv data to grid.
 *             Default = 1.
 * \li "numberChann" OBIT_long scalar = number of channels in uv data to grid.
 *             Default = all.
 * \param in      Object to initialize
 * \param UVin    Uv data object to be gridded.
 *                Should be the same as passed to previous call to
 *                #ObitUVGridSetup for input in.
 * \param err     ObitErr stack for reporting problems.
 */
void ObitUVGridReadUV (ObitUVGrid *in, ObitUV *UVin, ObitErr *err)
{
  ObitIOCode retCode = OBIT_IO_OK;
  ObitInfoType type;
  gint32 dim[MAXINFOELEMDIM];
  ofloat temp, czero[2] = {0.0,0.0};
  olong   itemp;
  olong i, nvis, lovis, hivis, nvisPerThread, nThreads;
  UVGridFuncArg *args=NULL;
  ObitThreadFunc func=(ObitThreadFunc)ThreadUVGridBuffer ;
  gboolean doCalSelect, OK;
  gchar *routine="ObitUVGridReadUV";

  /* error checks */
  g_assert (ObitErrIsA(err));
  if (err->error) return;
  g_assert (ObitUVGridIsA(in));
  g_assert (ObitUVIsA(UVin));
  g_assert (ObitUVDescIsA(UVin->myDesc));
  g_assert (UVin->myDesc->fscale!=NULL); /* frequency scaling table */

   /* If more than one Stokes issue warning */
  if ((UVin->myDesc->jlocs>=0) &&
      (UVin->myDesc->inaxes[UVin->myDesc->jlocs]>1)) {
      Obit_log_error(err, OBIT_InfoWarn,
    "%s: More than one Stokes  ( %d) in data, ONLY USING FIRST",
     routine, UVin->myDesc->inaxes[UVin->myDesc->jlocs]);
  }

  /* get gridding information */
  /* guardband */
  temp = 0.4;
  /* temp = 0.1; debug */
  ObitInfoListGetTest(in->info, "Guardband", &type, dim, &temp);
  in->guardband = temp;
```

Fig. 3. Read and grid UV data

```
  /* baseline range */
  temp = 1.0e15;
  ObitInfoListGetTest(in->info, "MaxBaseline", &type, dim, &temp);
  in->blmax = temp;
  temp = 0.0;
  ObitInfoListGetTest(in->info, "MinBaseline", &type, dim, &temp);
  in->blmin = temp;

 /* Spectral channels to grid */
  itemp = 1;
  ObitInfoListGetTest(in->info, "startChann", &type, dim, &itemp);
  in->startChann = itemp;
  itemp = 0; /* all */
  ObitInfoListGetTest(in->info, "numberChann", &type, dim, &itemp);
  in->numberChann = itemp;

  /* Calibrating or selecting? */
  doCalSelect = FALSE;
  ObitInfoListGetTest(UVin->info, "doCalSelect", &type, (gint32*)dim, &doCalSelect);

  /* UVin should have been opened in  ObitUVGridSetup */

  /* How many threads? */
  in->nThreads = MAX (1, ObitThreadNumProc(in->thread));

  /* Initialize threadArg array  */
  if (in->threadArgs==NULL) {
    in->threadArgs = g_malloc0(in->nThreads*sizeof(UVGridFuncArg*));
    for (i=0; i<in->nThreads; i++)
      in->threadArgs[i] = g_malloc0(sizeof(UVGridFuncArg));
  }

  /* Set up thread arguments */
  for (i=0; i<in->nThreads; i++) {
    args = (UVGridFuncArg*)in->threadArgs[i];
    args->in   = in;
    args->UVin = UVin;
    if (i>0) {
      /* Need new zeroed array */
      args->grid = ObitCArrayCreate("Temp grid", in->grid->ndim,  in->grid->naxis);
      ObitCArrayFill (args->grid, czero);
    } else {
      args->grid = ObitCArrayRef(in->grid);
    }
  }
  /* end initialize */
```

Fig. 4.   Read and grid UV data, initialization

```
  /* loop gridding data */
  while (retCode == OBIT_IO_OK) {

    /* read buffer */
    if (doCalSelect) retCode = ObitUVReadSelect (UVin, NULL, err);
    else retCode = ObitUVRead (UVin, NULL, err);
    if (err->error) Obit_traceback_msg (err, routine, in->name);

    /* Divide up work */
    nvis = UVin->myDesc->numVisBuff;
    if (nvis<1000) nThreads = 1;
    else nThreads = in->nThreads;
    nvisPerThread = nvis/nThreads;
    lovis = 1;
    hivis = nvisPerThread;
    hivis = MIN (hivis, nvis);

    /* Set up thread arguments */
    for (i=0; i<nThreads; i++) {
      if (i==(nThreads-1)) hivis = nvis;  /* Make sure do all */
      args = (UVGridFuncArg*)in->threadArgs[i];
      args->first  = lovis;
      args->last   = hivis;
      if (nThreads>1) args->ithread = i;
      else args->ithread = -1;
     /* Update which vis */
      lovis += nvisPerThread;
      hivis += nvisPerThread;
      hivis = MIN (hivis, nvis);
    }

    /* Do operation on buffer possibly with threads */
    OK = ObitThreadIterator (in->thread, nThreads, func, in->threadArgs);

    /* Check for problems */
    if (!OK) {
      Obit_log_error(err, OBIT_Error,"%s: Problem in threading", routine);
      break;
    }
  } /* end loop reading/gridding data */
```

Fig. 5. Read and grid UV data, loop over data gridding

```
 /* Accumulate thread grids if more than one */
  if (in->nThreads>1) {
    for (i=1; i<in->nThreads; i++) {
      args = (UVGridFuncArg*)in->threadArgs[i];
      ObitCArrayAdd(in->grid, args->grid, in->grid);
    }
  } /* end accumulating grids */

  /* Shut down any threading */
  ObitThreadPoolFree (in->thread);
  if (in->threadArgs) {
    for (i=0; i<nThreads; i++) {
      args = (UVGridFuncArg*)in->threadArgs[i];
      if (args->grid) ObitCArrayUnref(args->grid);
      g_free(in->threadArgs[i]);
    }
    g_free(in->threadArgs);
  }
  in->threadArgs = NULL;
  in->nThreads   = 0;

  /* Close data */
  retCode = ObitUVClose (UVin, err);
  if (err->error) Obit_traceback_msg (err, routine, in->name);

} /* end ObitUVGridReadUV  */
```

Fig. 6. Read and grid UV data, accumulate thread grids

```
/**
 * Prepare and Grid a portion of the data buffer
 * Arguments are given in the structure passed as arg
 * \param arg  Pointer to UVGridFuncArg argument with elements
 * \li in      ObitUVGrid object
 * \li UVin    UV data set to grid from current buffer
 * \li first   First (1-rel) vis in UVin buffer to process this thread
 * \li last    Highest (1-rel) vis in UVin buffer to process this thread
 * \li ithread thread number, >0 -> no threading
 */
static gpointer ThreadUVGridBuffer (gpointer arg)
{
  /* Get arguments from structure */
  UVGridFuncArg *largs = (UVGridFuncArg*)arg;
  ObitUVGrid *in   = largs->in;
  ObitUV *UVin     = largs->UVin;
  olong loVis      = largs->first-1;
  olong hiVis      = largs->last;
  ObitCArray *grid = largs->grid;

  /* prepare data */
  PrepBuffer (in, UVin, loVis, hiVis);

  /* grid */
  GridBuffer (in, UVin, loVis, hiVis, grid);

  /* Indicate completion */
  if (largs->ithread>=0)
    ObitThreadPoolDone (in->thread, (gpointer)&largs->ithread);

  return NULL;
} /* end ThreadUVGridBuffer */
```

Fig. 7. Function called for each thread