The Implementation of Threaded Image Interpolation in Obit

W. D. Cotton, November 11, 2008

***Abstract*—This memo describes an implementation of multi-threaded image interpolation. Results of a test interpolating the pixel values in an image from one geometry to another geometry are presented. There is a nearly linear improvement in performance with up to 8 parallel threads.**

***Index Terms*—Computing efficiency, astronomical images**

I. INTRODUCTION

ONE of the common relative compute intensive operations in astronomy is interpolating the pixels in an image onto another geometry. For large images or large numbers of images, this can be an expensive operation. This operation is almost always needed for the comparison of images at various wavebands or for the "flattening" of a Fly's eye mosaic of images onto a single plane. The following discusses an implementation of multi-threading of image interpolation in Obit ([1], http://www.cv.nrao.edu/∼bcotton/Obit.html). This implementation follows the general method of multi–threading in Obit as described in [2].

II. IMAGE INTERPOLATION

The process of image interpolation is relatively straightforward. First, for each pixel in the output image, the corresponding pixel in the input image on the sky must be determined; then the value in the input image at that pixel must be interpolated. The translation of pixel locations is complicated by the various nonlinear geometries in common use and the possibility of a precession of the equinox of the celestial coordinates. In the general case, pixels in the input image will not be aligned on the same grid as the output image and the pixel values must be determined by interpolation. Edge effects and pixels with undefined values further complicate this. These two operations are described further in the following.

A. Coordinate Conversion

Astronomical images have the difficulty that they are flat but the sky is not; therefore a number of different geometries have been developed for projecting the curved sky onto a flat image. Different projections are appropriate for different applications. A further complication of celestial coordinates is that the coordinate system rotates with time to stay aligned with the earth's rotational axis. Images are generally referred to the mean coordinate system at standard equinoxes (e.g. 1900, 1950, 2000).

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

Manuscript received ; revised

TABLE I
PROJECTIVE GEOMETRIES IN OBIT

Geometry code	Geometry name
-SIN	Sin projection
-TAN	Tan projection
-ARC	Arc projection
-NCP	NCP (WSRT) projection
-GLS	Global sinusoid projection
-MER	Mercator projection
-AIT	Aitoff projection
-STG	Stereographic projection

The general method for determining corresponding pixels between two images is to convert the first pixel location to a celestial coordinate using the projective geometry of that image. Then it may be necessary to precess the equinox of the coordinate to that of the second image. The pixel location in the second image at the desired celestial coordinate then can be determined from the celestial coordinate and the projective geometry of that image. Projective geometries in Obit are implemented in the ObitSkyGeom utility module which supports translations between pixels and celestial coordinates. Obit supports the AIPS projective geometries shown in Table I.

A further complication is that the coordinate translation may need to also incorporate a distortion of the geometry. This is implemented in Obit using a Zernike polynomial representation. This is needed to incorporate the effects of the ionosphere on the apparent positions of celestial locations.

B. Pixel Interpolation

In the general case, locations desired in the input image will lie between pixels so an interpolation is necessary. This operation in Obit is performed using the ObitFInterpolate class which interpolates values in an ObitFArray containing the pixel values. Interpolation is by the Lagrange method and gives support of a range of interpolation kernel sizes. Pixels with undefined values ("blanked") must also be accommodated.

For the combination of overlapping images as in a linear mosaic or flattening a Fly's eye, it is desirable to taper the images toward the edge to eliminate the "seams" between images. In Obit, this is accomplished by creating a "weighted" image and the corresponding "weight" image to be used in the weighted combination of the overlapping images. The weight

TABLE II
TIMING TESTS OF HGEOM

# threads	Avg. Run time (sec)	Rel. Speed
1	10.35	1.00
2	5.94	1.74
3	4.23	2.45
4	3.31	3.13
5	2.68	3.86
6	2.22	4.66
7	1.90	5.95
8	1.65	6.28

used is unity inside a given radius and then rapidly tapers to zero.

III. MULTI–THREAD IMAGE INTERPOLATION

The general method of image interpolation is to loop over the pixels in the output image, calculate the corresponding pixel in the input image and interpolate its value. Except for the internal state of the interpolator, there are no dependencies for this operation amoung different output pixels. This allows dividing the work among various parallel processors/cores when available. In the Obit implementation, different sets of image rows are divided among different threads; some of the details are given in the Appendix. (See [2] for a description of the general technique in Obit for multi-threaded operation.) Each thread has an independent interpolator but each uses the same ObitFArray containing the input pixel values.

IV. TIMING TEST

To evaluate the effectiveness of this technique, a test was constructed interpolating a 2500×2500 pixel image to one with a slightly different pixel spacing but with the same equinox and projective geometry. This interpolation was done with Obit task HGeom.

The tests were performed on the Obit development computer, mortibus, in Charlottesville. This machine has two quad core processors for a total of 8 processing elements and running a 32–bit Red Hat Linux OS. The Unix "time" facility was used to determine the wall clock execution time of each trial. A set of 5 trials was run using each of 1 to 8 threads. The averaged results of the timing tests are shown in Table II 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.

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 6.28 over the single thread run time. The relatively linear increase in performance with increasing number of threads shown in Figure 1 indicates that

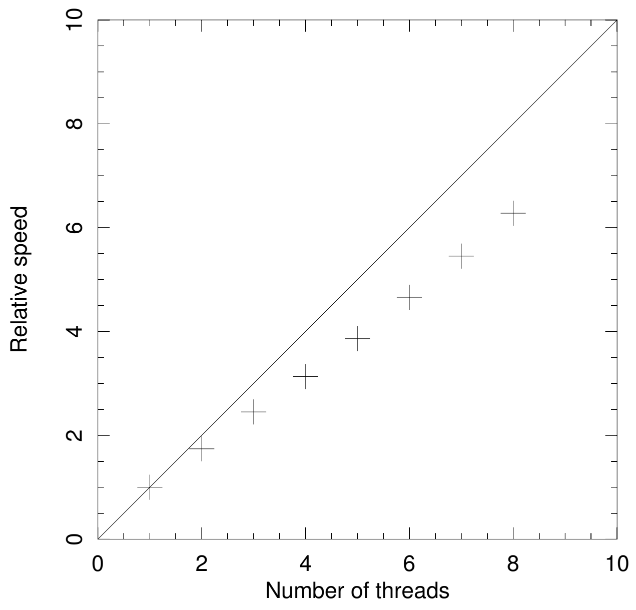

Fig. 1. Processing speed relative to a single processor. The solid line indicates complete utilization of the processors.

the fraction of the work which was run in parallel was very large and that performance would benefit from an increased number of processors. However, the slope of the points in Figure 1 is less than unity which suggests that the overhead of starting threads is a non trivial fraction of the total work performed, of order of 10%. This implementation used the glib thread pools which dramatically reduces the thread startup overhead whenever a given thread is reused many times. In this particular application each thread was only given a single package of work.

APPENDIX

The following software fragments are a detailed description of the implementation of threading of image interpolation. This implementation supports simple interpolation, weighted interpolation and interpolation given a Zernike polynomial representation of the distortion of the input geometry.

REFERENCES

[1] W. D. Cotton, "Obit: A Development Environment for Astronomical Algorithms," *PASP*, vol. 120, pp. 439–448, 2008.
[2] ——, "Note on the Efficacy of Multi-threading in Obit," *Obit Development Memo*, no. 1, 2008.

```
typedef struct {
  /* Input descriptor */
  ObitImageDesc *inDesc;
  /* Input plane pixel data */
  ObitFArray *inData;
  /* Output descriptor */
  ObitImageDesc *outDesc;
  /* Output plane pixel data */
  ObitFArray *outData;
  /* Also do Weights? */
  gboolean   doWeight;
  /* Output weight plane pixel data */
  ObitFArray *wtData;
  /* Radius in pixels of weighting circle */
  olong      radius;
  /* Number of Zernike corrections */
  olong      nZern;
  /* Zernike coefficients */
  ofloat     *ZCoef;
  /* First (1-rel) row in image to process this thread */
  olong      first;
  /* Highest (1-rel) row in image to process this thread  */
  olong      last;
  /* thread number, <0 -> no threading  */
  olong      ithread;
  /* Obit Thread object */
  ObitThread  *thread;
  /* Obit error stack object */
  ObitErr    *err;
  /* Input Image Interpolator */
  ObitFInterpolate *Interp;
} InterpFuncArg;
```

Fig. 2. Thread call function argument structure

```
/**
 * Interpolate selected rows from one image onto another,
 * possibly including weighting across the image.
 * If doWeight, outData will be filled with the pixel values
 * interpolated from inData multiplied by a weight based on a
 * circle defined by radius from the center; this is 1.0
 * in the center and tapers with distance^2 to 0.0 outside
 * and the weights are written into wtData.
 * Distortion of the input image geometry by a Zernike polynomial
 * is also supported.
 * Magic value blanking is supported.
 * Callable as thread
 * \param arg Pointer to InterpFuncArg argument with elements:
 * \li inDesc   Image Descriptor for input image
 * \li inData   ObitFArray with input plane pixel data
 * \li outDesc  Image Descriptor for output image
 * \li outData  ObitFArray for output plane pixel data
 * \li doWeight gboolean if TRUE, also do primary beam weighting
 * \li wtData   ObitFArray for output weight plane pixel data
 *              only used if doWeight
 * \li radius   Radius in pixels of weighting circle
 *              only used if doWeight
 * \li nZern    If>0 apply cernike corrections to position
 *              nZern is the number of terms in ZCoef
 * \li zCoef    Zernike correction coefficients
 *              only used if zCoef>0
 * \li first    First (1-rel) row in image to process this thread
 * \li last     Highest (1-rel) row in image to process this thread
 * \li ithread  thread number, <0 -> no threading
 * \li thread   thread Object
 * \li err      ObitErr Obit error stack object
 * \li Interp   ObitFInterpolate Input Image Interpolator
 * \return NULL
 */
static gpointer ThreadImageInterp (gpointer args)
{
  /* Get arguments from structure */
  InterpFuncArg *largs = (InterpFuncArg*)args;
  ObitImageDesc *inDesc = largs->inDesc;
  /* ObitFArray *inData    = largs->inData;*/
  ObitImageDesc *outDesc= largs->outDesc;
  ObitFArray *outData   = largs->outData;
  gboolean   doWeight   = largs->doWeight;
  ObitFArray *wtData    = largs->wtData;
  olong      radius     = largs->radius;
  olong      nZern      = largs->nZern;
  ofloat     *ZCoef     = largs->ZCoef;
  olong      loRow      = largs->first;
  olong      hiRow      = largs->last;
  ObitErr    *err       = largs->err;
  ObitThread *thread    = largs->thread;
  ObitFInterpolate *interp = largs->Interp;
  /* local */
  olong ix, iy, indx, pos[2];
  ofloat inPixel[2], outPixel[2], *out, *outWt, rad2, dist2, irad2;
  ofloat *crpix, wt, val, fblank =  ObitMagicF();
  gboolean OK;
  gchar *routine = "ThreadImageInterp";
```

```
/* Get output aray pointer */
pos[0] = pos[1] = 0;
out   = ObitFArrayIndex (outData, pos);

/* Coordinate reference pixel of input */
crpix = inDesc->crpix;

/* if weighting */
if (doWeight) {
  /* Working version of radius */
  rad2 = radius * radius;
  irad2 = 1.0 / rad2;
  outWt = ObitFArrayIndex (wtData, pos);
}

/* Loop over image interpolating */
for (iy = loRow; iy<=hiRow; iy++) { /* loop in y */
  outPixel[1] = (ofloat)iy;
  for (ix = 1; ix<=outDesc->inaxes[0]; ix++) {/* loop in x */
    outPixel[0] = (ofloat)ix;

   /* Get pixel in input image - Zernike correction?*/
    if (nZern>0) { /* yes */
      OK = ObitImageDescCvtZern (outDesc, inDesc, nZern, ZCoef,
                                 outPixel, inPixel, err);
    } else {       /* no */
      OK = ObitImageDescCvtPixel (outDesc, inDesc, outPixel, inPixel, err);
    }
    if (err->error) {
      ObitThreadLock(thread);  /* Lock against other threads */
      Obit_log_error(err, OBIT_Error,"%s: Error projecting pixel",
                     routine);
      ObitThreadUnlock(thread);
      goto finish;
    }

    if (doWeight) { /* weighting? */
      if (OK) { /* In image? */
        /* weight based on distance from center of inImage */
        dist2 = (crpix[0]-inPixel[0])*(crpix[0]-inPixel[0]) +
          (crpix[1]-inPixel[1])*(crpix[1]-inPixel[1]);
        /*dist2 = (crpix[0]-xyzi[0])**2 + (crpix[1]-xyzi[1])**2;*/
        if (dist2 <= rad2) {
          wt = 1.0 - dist2 * irad2;
          wt = MAX (0.001, wt);
        } else {
          wt = fblank;
        }
      } else {
        wt = fblank;  /* don't bother */
      }
    } else wt = 1.0;
```

```
      /* interpolate */
      /* array index in out for this pixel */
      indx = (iy-1) * outDesc->inaxes[0] + (ix-1);
      if (wt != fblank ) {
        val = ObitFInterpolatePixel (interp, inPixel, err);
        if (doWeight & (val != fblank )) val *= wt;
        out[indx] = val;
        if (err->error) {
          ObitThreadLock(thread);  /* Lock against other threads */
          Obit_log_error(err, OBIT_Error,"%s: Error interpolating pixel in %s",
                         routine, interp->name);
          ObitThreadUnlock(thread);
          goto finish;
        }
        if (doWeight) outWt[indx] = wt;
      } else {
        out[indx]   = fblank;
        if (doWeight) outWt[indx] = fblank;
      }

    } /* end loop over x */
  } /* end loop over y */

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

  return NULL;
} /* ThreadImageInterp */
```

Fig. 3. Thread call function for image interpolation