Computational Concurrency in X-Ray Image Restoration

Michael S. Noble
mnoble@cfa.harvard.edu
http://hea-www.harvard.edu/m̃noble

May 13, 1999

Abstract

This paper surveys the extent to which parallelism may be employed to speed up Richardson-Lucy (RL) deconvolution in astronomical image analysis. First a brief background of imaging in astronomy is given, followed by a presentation of the classic imaging equation and the attendant mathematical machinery leading to the RL method. We then discuss how concurrency can enhance the RL computation, and mention previous work that focused upon visible image restoration. Finally an outline of how RL can be applied to X-ray astronomy is presented, as well as plans for implementation of a concurrent solution using MPI, first on an SGI Origin2000 MPP, and later on a networked cluster of workstations.

1 Introduction

Image processing plays a central role in modern astronomy and astrophysics, with sophisticated mirror, camera, and detector assemblies having largely supplanted the technique of naked-eye observation used by astronomers in previous centuries.

Not only does the use of these devices automate and render more precise the collection of astrophysical data, it also widens the scope of inquiry by allowing scientists to peer beyond the merely visible into portions of the electromagnetic spectrum that simply cannot be perceived with the human eye (for example, the UV and X-ray bands).

The use of modern instrumentation, however, does not come without a price: no machine is perfectly constructed, which means no device can faithfully record (ie, with 100% accuracy) the phenomena under observation. Invariably the physical and electrical properties of the mirror and instruments contribute to the collected data, meaning that what is recorded is a somewhat distorted view of what was actually observed. In the scientific lexicon, the extent to which the instruments and mirrors faithfully reproduce the observed phenomena are referred to respectively as the *instrument response* and *point spread function* (PSF).

Over the years a wide range of engineering and numerical methods have been developed to enhance the instrument response and account for the PSF. In this paper we survey one of the more widely used numerical techniques, Richardson-Lucy (RL), and examine the extent to which RL can be parallelized for computational efficiency and improved accuracy. Previous work in this area by

Boden et al [2] focused upon parallel restoration of visible Hubble Space Telescope images, while herein we focus upon the X-ray spectrum. The author hopes that this work will eventually be utilized to deconvolve images generated by the Chandra X-Ray Observatory (scheduled for launch in July, 1999).

2 Theory

2.1 The Imaging Equation

The classic *imaging equation* is of seminal importance in image restoration, and specifies how an observed image I relates to an object O as seen through an optical system (instrument) with PSF P:

$$I(x) = \int O(x')P(x,x')dx' \tag{1}$$

Because CCD (charge-coupled device) detectors are commonly employed to count quantized photons, it is convenient to utilize the discrete form

$$I_q(x) = \sum O_q(x')P_q(x,x') \tag{2}$$

Here the PSF gives the probability that a photon from the true incident location x$'$ intersects the detector at pixel position x (ie, contributes to pixel x within the observed image), and each of I_q, O_q, and P_q are averaged over the detector grid intervals.

2.2 Convolution, Restoration, and Richardson-Lucy

From Fourier transform theory, the *convolution* of two functions f and g (each of which obey certain conditions) is *defined* as

$$h(x) = (f*g)(x) = \int_{-\infty}^{\infty} f(p)g(x-p)dp = \int_{-\infty}^{\infty} f(x-p)g(p)dp \tag{3}$$

The convolution is used when f(x) and g(x) are known to be the inverse transforms of some functions F(s) and G(s), and we would like to calculate the inverse h(x) of their product H(s) = F(s)G(s).

The convolution theorem [5] goes further to state that the Fourier transform $\mathcal{F}$ has the nice property

$$\mathcal{F}(f*g) = \sqrt{2\pi}\mathcal{F}(g)\mathcal{F}(f) \tag{4}$$

which is to say the Fourier transform of a convolution is product of the Fourier transforms of the functions to be convolved, multiplied by $\sqrt{2\pi}$. This is important because when the PSF response function P in equation 2 does not vary across the detector/mirror assembly (ie, it is *spatially invariant*), P may be expressed as

$$P_q(x,x') = P_q(x-x') \tag{5}$$

and equation 2 becomes

$$I_q(x) = \sum O_q(x')P_q(x - x') \tag{6}$$

which is by definition a discrete convolution! And this further implies that in the presence of noise equation 2 may be efficiently computed in an iterative manner by simple division in the Fourier domain, with each iteration yielding a sharper restoration of the observed image I.

When coupled with the assumption that the rate of photon impingement upon the optical system is governed by Poisson statistics, equation 6 as derived is the foundation for most sequential Richardson-Lucy restoration solutions.

3 Parallelism

Ok, so where in this picture do we find a need for parallelism to increase computation speed? Well, it turns out that in practice no PSF is wholly spatially invariant, so in many instances this simplifying assumption can lead to increased error and suboptimal restoration.

Remedying this by introducing a spatially-variant PSF, however, can push the computational load out of the pragmatic range of a conventional (ie, single-processor) sequential implementation, the primary reason being that Fourier convolution may no longer be utilized to simplify the iterative computation of equation 2.

Boden et al [2] detail several approaches to incorporating a variable PSF into the restoration model. At the outset this author will explore the embarrassingly parallel technique of subdividing the detector into a mesh, with each segment assumed to have a constant PSF, thereby reaching a compromise between the naive wholly-invariant and the computationally prohibitive fully-variant methods.

Because we assume a constant PSF per segment, Fourier convolution may be employed on a per-segment basis, with each segment restoration being computed independently (with the overlap proviso that some pixels will appear in multiple segments). Further speedup may be obtained by parallelizing the Fourier transform itself, and possibly during a preconditioning phase, where the input data are converted from real values (as they are stored on disk) to complex and where the PSF is shifted so that its center is repositioned to the upper left corner.

4 Implementation

After studying the previous sequential HST RL implementations—one in IRAF (a widely used astronomical analysis system) SPP, another in the commercial IDL language—a prototype sequential implementation will be coded in straight C on a Solaris workstation at the Harvard-Smithsonian Center for Astrophysics, after which the implementation will incorporate the MPI message passing library and migrate to the massively parallel SGI Origin 2000 array at Boston University.

MPICH will then be installed on a workstation cluster at the CFA, and the application will be tuned for production usage during Chandra X-Ray Observatory data processing. Porting to MPI on the SGI array should simplify the translation, as it should be easier to debug a single MPP than a cluster of workstations, each of which will harbor competing computational workloads and filesystems.

Both the parallel and sequential implementations will utilize the CFITSIO package for image I/O (via standard FITS files) and a public-domain FFT package (FFTW [4]) that has delivered excellent benchmark performance.

5 Current Progress and Observations

The effort to convert IRAF and IDL was instructive, and more time-consuming than originally planned for a quarter-semesters worth of effort. IRAF SPP is actually FORTRAN in disguise, where pre-processor macros are used to effect a C-like syntax, while IDL is so extremely compact and powerfully expressive that it hides a considerable amount of detail/complexity that cannot be avoided in C.

Unraveling powerful IDL expressions [like M = MxN where M,N are 512x512 complex matrices, or conjugate(M)] into lengthier C code is a MUST if we hope to gain any notable speedup through parallelization. The IDL and C codes have been included in an appendix for comparison.

Similarly, by inspection it can be realized that some techniques used for maximizing sequential speed (like bulk memory manipulation with memset, memcpy, etc) are counter-productive in a shared-memory programming model. For example, in the sequential case to quickly initialize an array we might use memset(array,0,count), but doing the same in an MPP would pile the initialization responsibility all on one process (because the memset call is essentially atomic). So, to gain MPP performance we would unravel the initialization into a for loop that could then be partitioned more evenly amongst the processors.

Early tests suggest my FFTW-based sequential implementation is 10-15% faster than its IDL predecessor. This alone has created some excitement at the CFA, and since neither IDL nor IRAF supports concurrency, bodes well for the future parallel implementation.

6 Future Directions

Continue the port to the BU SGI Origin200 array. To further confirm the viability of the parallelization, the sequential version will be trivially parallelized by inserting PowerC parallelization pragmas at key points in the code. After this the true MPI port and testing will resume, followed sometime in late summer 1999 by migrating back to the CFA NOW.

Additionally, the core algorithm will be refined by incorporating cutting-edge numerical techniques developed in collaboration by Harvard and BU statisticians, to account for the significantly lower photon counts seen when working in the X-ray spectrum.

References

[1] R. White, "Improvements to the Richardson-Lucy Method" Restoration (Newsletter of Hubble Space Telescope Image Restoration Project), Summer 1993, Space Telescope Science Institute

[2] A. Boden, D. Redding, R. Hanish, J. Mo, "Massively Parallel Spatially-Variant Maximum Likelihood Restoration of Hubble Space Telescope Imagery", Journal of the Optical Society of America, Volume 13 No. 7, July 1996

[3] W.Gropp, E.Lusk, A. Skjellum, "MPICH: A Portable Implementation of MPI", http://www-unix.mcs.anl.gov/mpi/mpich

[4] M. Friego, S. Johnson, "Fastest Fourier Transform in the West", http://theory.lcs.mit.edu/ fftw

[5] E. Kreyszig, "Advanced Engineering Mathematics", 7th Ed., 1993, John Wiley and Sons

A Appendix: Comparing IDL and C Codes

A.1 Excerpted IDL Code

```
; SIMPLE Richardson-Lucy Procedure (no Cosmic Ray or Bad Pixel Removal)

pro rl,d,ah,p,niter,n,screen

epsilon = 1e-6
ah=d
p = fft(p,1)
conjpsf = conj(p)

for i = 0,niter do begin
    ah = fft(ah,1)
    dh = p*ah
    dh = fft(dh,-1)
    dh = d/(dh+epsilon)
    dh = fft(dh,1)
    dh = conjpsf*dh
    dh = fft(dh,-1)
    ah = fft(ah,-1)*dh
endfor

p = fft(p,-1)
end
```

A.2 Excerpted C Code

```
{
/* Resize the PSF to match the image, for convenient arithmetic */
if (psfsize < imsize)
{
   register int i;
   register int oldncols = psfdims[0];
   register int oldrowsize = oldncols*sizeof(complex);
   register int newncols = imdims[0];
   complex *tmp = psf;

   /* Resize PSF and zero out new pixels */
   psf = calloc(imsize,sizeof(complex));
   for(i=0; i<psfdims[0]; i++)
      memmove(psf+i*newncols,tmp+i*oldncols,oldrowsize);
   free(tmp);
}
else if (psfsize > imsize)
ERROR("psf cannot be larger than image to be restored!");

/* Shift center of psf to upper left edge */
shiftl(psf,imdims,psfdims[0]/2,psfdims[1]/2);
psfdims[0] = imdims[0]; psfdims[1] = imdims[1];

/* Create FFT plans for transforming the data to/from Fourier space */
fromfspace = fftwnd_create_plan(2,imdims,FFTW_FORWARD,FFTW_IN_PLACE);
tofspace = fftwnd_create_plan(2,imdims,FFTW_BACKWARD,FFTW_IN_PLACE);
if (!fromfspace || !tofspace)
   ERROR("could not compute either fromfspace or tofspace FFT plan");

/* Transform the PSF into Fourier space */
fftwnd_one(tofspace,psf,NULL);

/* Generate the complex conjugate of the transformed PSF */
psfconj = (complex*)calloc(imsize,sizeof(complex));
for (i=0; i < imsize; i++)
{
    psfconj[i].re = psf[i].re;
    psfconj[i].im = -1.0 * psf[i].im;
}

/* Finally, before iterating the heart of the RL */
/* algorithm, save a copy of the original image  */
orig = (complex*)malloc(imsize*sizeof(complex));
memmove(orig,image,imsize*sizeof(complex));

/* And allocate some scratch space */
scratch = (complex*)malloc(imsize*sizeof(complex));

printf("Beginning RL algorithm, will perform %d iterations\n",iters);

/* Iterate the Richardson-Lucy algorithm */
for (iter=0; iter<=iters; iter++)
{
   printf("lucy  %d\n",iter);

   /* Transform the current image iteration into Fourier space */
   fftwnd_one(tofspace,image,NULL);

   /* Multiply by the PSF  (Note that the FFTW library requires   */
   /* that we doing our own scaling when coming back from fspace) */
   multvec(psf,image,scratch,imsize,&scale);

   /* Transform result back to real space ... */
   fftwnd_one(fromfspace,scratch,NULL);

   /* ... and divide original image by the result */
```

```
   for (i=0; i<imsize; i++)
   {
       scratch[i].re = orig[i].re/(scratch[i].re+EPSILON);
       scratch[i].im = orig[i].im/(scratch[i].re+EPSILON);
   }

   /* Go back to Fourier space and multiply by conjugate(PSF) */
   fftwnd_one(tofspace,scratch,NULL);

   multvec(scratch,psfconj,scratch,imsize,&scale);

   /* Go back to real space ... */
   fftwnd_one(fromfspace,scratch,NULL);
   fftwnd_one(fromfspace,image,NULL);

   /* ... and perform the final multiplication) */
   multvec(image,scratch,image,imsize,&scale);

   printf("After %dth iteration ..., ah(0:19) is \n",iter);
   dumpdata(imdims,image,-20);

}
}

/* multvec: Multiply two vectors of length abs(n).  The multiplication is */
/*     done in the complex field when n>0, and real field when n<0.  */
/*     The result will be scaled by a real factor when scale != NULL */
/*     Having slightly longer code (because code for the scaled mults*/
/*     is essentially duplicated) results in fewer operations in the */
/*     common unscaled case, and fewer memory refs in the scaled case*/
void multvec(complex *a, complex *b, complex *result, int n, double *scale)
{

   if (--n>=0)
{
   /* use a temp var, to permit result array to be same as a or b */
   register double tmp;

   /* perform complex vector multiplication */
   if (!scale)
      do
      {
        tmp  = a[n].re*b[n].re - a[n].im*b[n].im;
        result[n].im = a[n].re*b[n].im + a[n].im*b[n].re;
        result[n].re = tmp;
      } while (--n > -1);
   else
   {
        register double scal = *scale;
        do
        {
           tmp  = (a[n].re*b[n].re - a[n].im*b[n].im)*scal;
           result[n].im = (a[n].re*b[n].im + a[n].im*b[n].re)*scal;
           result[n].re = tmp;
        } while (--n > -1);
   }
}
else
{
   /* perform real vector multiplication */
   n = n*-1 - 1;
   if (!scale)
      do { result[n].re = a[n].re*b[n].re; } while (--n > -1);
   else
   {
      register double scal = *scale;
      do { result[n].re = a[n].re*b[n].re*scal; } while (--n > -1);
   }
```

```
}
}


/* 2D matrix shift, kinda brute force, left along each dimension. */
/* Note: Assumes matrix memory is contiguous & in row-major order */
/*       Could benefit from more generalization ...   */
void
shiftl(complex *arr,int *dims,int ncols,int nrows)
{

        complex *tmp;
        register int big;  /* Small is really < big only when */
        register int small; /* ncols|nrows < dims[rows|cols]/2 */

        /* Wrap by modulus */
        ncols %= dims[0];
        nrows %= dims[1];

        /* Shift the rows (quick, but could be memory hungry for large arr) */
        if (nrows > 0)
        {
                small = nrows*dims[0]*sizeof(complex);
                big   = (dims[1]-nrows)*dims[0]*sizeof(complex);
                tmp = (complex*)malloc(small);

                /* Store the smaller chunk (which would be overwritten) */
                memcpy(tmp,arr,small);
                /* Move the larger chunk to where the smaller chunk was */
                memmove(arr,((char*)arr)+small,big);
                /* And put the smaller chunk in its place */
                memmove(((char*)arr)+big,tmp,small);
                free(tmp);
        }

        /* Shift the cols (slower due to iteration, but uses less memory) */
        if (ncols > 0)
        {
                register int row;
                small = ncols*sizeof(complex);
                big   = (dims[0]-ncols)*sizeof(complex);
                tmp   = (complex*)malloc(small);

                for (row=0; row<dims[1]; arr += dims[0],row++)
                {
                    memcpy(tmp,arr,small);
                    memmove(arr,arr+ncols,big);
                    memmove(((char*)arr)+big,tmp,small);
                }
                free(tmp);
        }
}
```

B Appendix: Control and Data Flow in Richardson-Lucy

Data/Control Flow in Richardson-Lucy