Efficacy of Double Buffered I/O

W. D. Cotton April 27, 2018

***Abstract*—Much of the data processing for radio interferometry involve large amounts of data with relatively small amounts of processing each of multiple passes through the data. It is important to overlap the I/O with the processing to the greatest extent possible to minimize the total run time. The classical technique for this is "multiple buffering"; operating on one buffer while I/O is being done to others. The Linux file system attempts this by caching parts of the file system in unused memory enabling read–ahead and write–behind. This works well on local file systems where the OS knows all the traffic to a given file but the performance is less clear for remote file systems such as NFS or lustre. This memo describes a comparison of single buffered use of the linux file system with double buffered reads. In none of the tests performed did double buffering make a significant improvement. Some machine/disk combinations showed a 30% run time performance from linux overlapping I/O and computation yet other cases showed no improvement.**

***Index Terms*—interferometry, double buffering**

I. Introduction

OVERLAPPING I/O with computing is desirable for good performance in processing radio interferometry and related data. Large data sets and image cubes must be read and written in multiple passes. Multiple buffering, operating on one data buffer while doing I/O to another is a classic solution to this problem and is employed in AIPS. The linux file system is expected to help with this on local files when adequate memory is available and it can anticipate the I/O access pattern. Obit has depended on the Unix file system for good performance. The situation is less clear when the entire problem does not fit in memory or a remote file system is used. This memo discusses a comparison of single and double buffered I/O in a number of cases using the Obit package [1] [1].

II. Double/Single Buffer Test

The test case is reading an image cube approximately 8k × 8k × 400 pixels and doing a fixed amount of computing on each plane using a fixed number of threads. The cube is stored as scaled shorts and is ≈54 GByte. I/O is a plane at a time in which each pixel is scaled to a float. The computational load consists of evaluating the RMS of each plane by a histogram analysis some number of times using a given number of threads. Single buffering consists of simple calls to Obit Image function GetPlane to an open file; double buffering calls GetPlane in a thread created for each plane to read the next image plane. Implementation is in a python routine using the Obit software interface and is shown in Figure 1

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

[1]http://www.cv.nrao.edu/∼bcotton/Obit.html

III. Test Cases

Tests were performed on a number of machines and file systems in otherwise empty machines. Some of the tests used the lustre file system for which the load from other computers was unknown but the tests were performed outside of peak hours. These tests were conducted with a number of computing repeats including 0 to determine to pure I/O time as well as a "no I/O" test to determine the time needed for the computation. Tests were performed using a FITS file and cfitsio.

- panther
  This is a laptop with 2 × 2.6 GHz cores, AVX and 8 GByte of RAM. An SSD as well as a simple hard drive are included.
- zuul02
  This is a workstation with 16 × 2.4 GHz cores and 64 GByte of RAM. There are no local high performance disks but has a good connection to lustre.
- gollum
  This is workstation with 8 × 2.27 GHz cores and 24 GByte of RAM. A local software RAID disk is available.
- zuul05
  This is a workstation with 16 × 2.0 GHz cores and 64 GByte of RAM. A local, fast hardware RAID disk, SSD and a fast lustre connection are available.
- smeagle
  This a workstation with 16 × 3.1 GHz cores with AVX and 256 GByte of RAM A local, software RAID disk, SSD and a slow lustre connection are available.

The various tests run are summarized in Table I. In none of the tests in this table did double buffering improve the performance (column "ratio") and in many cases slightly degraded the performance. However, the double buffering implementation may be sub-optimal. In the following only the single buffered tests will be considered.

A further question is how much did the unix file system manage to overlap IO with the computation. To investigate this, the tests were rerun with only the first plane of the image actually being read but the same computations done as for the full 401 planes with repeats per plane. These timings are shown in Table II. If there were no overlapping of the IO and computation, the I/O only and CPU only timings would equal the total time ("single" in Table I). The fraction of the I/O time "hidden" behind the computation is given by:

$$\text{fract} \;=\; \frac{(i+c-t)}{i}$$

where i is the I/O only time, c is the CPU only time and t is the total for both I/O and computation. These results are given in Table III. Only the gollum and zuul02 tests showed an improvement and that of 30%.

TABLE I
TOTAL TIMING TESTS

machine	disk	nThread	nRepeat	single sec	CPUs	double sec	CPUd	ratio
panther	SSD	2	0	167	1.00	167	1.00	1.00
panther	HD	2	0	548	0.30	541	0.30	1.01
panther	SSD	2	1	559	1.46	561	1.46	1.00
panther	SSD	2	2	959	1.54	950	1.54	1.01
panther	HD	2	2	1339	1.09	1338	1.09	1.00
panther	SSD	2	5	2164	1.59	2150	1.59	1.01
panther	HD	2	5	2557	1.35	2554	1.35	1.00
panther	SSD	2	10	4258	1.60	4167	1.62	1.02
panther	HD	2	10	4667	1.44	4620	1.44	1.01
zuul02	lustre	8	0	450	0.79	464	0.77	0.97
zuul02	lustre	8	5	1451	3.54	1471	3.53	0.99
gollum	RAID	8	0	393	0.78	406	0.76	0.97
gollum	RAID	8	5	1150	2.79	1190	2.71	0.97
zuul05	RAID	8	5	1752	3.37	1813	3.30	0.97
zuul05	SSD	8	5	1831	3.30	1871	3.28	0.98
zuul05	RAID	8	2	751	3.22	807	3.10	1.06
zuul05	RAID	8	2	745	3.23	804	3.12	1.06
zuul05	SSD	8	2	793	3.10	821	3.08	0.97
zuul05	lustre	8	2	815	3.07	868	2.92	0.94
zuul05	RAID	8	0	87	0.88	102	0.94	0.86
zuul05	SSD	8	0	117	0.88	123	0.88	0.95
zuul05	lustre	8	0	167	0.82	194	0.70	0.86
smeagle	RAID	8	5	1724	3.17	1829	3.03	0.94
smeagle	SSD	8	5	1711	3.19	1669	3.20	1.03
smeagle	lustre	8	5	1735	3.16	1949	2.84	0.89
smeagle	RAID	8	0	124	1.00	134	1.00	0.92
smeagle	SSD	8	0	121	1.00	129	1.00	0.94
smeagle	lustre	8	0	127	1.00	131	1.00	0.97

Notes: “nThreads” is the number of threads used in the computation, “nRepeat” is the number of times the plane computation was repeated, “single” is the time for the single buffered test, “CPUs” is the single buffer CPU/Real time ratio, “double” is the time for the double buffered test, “CPUd” is the double buffer CPU/Real time ratio, “ratio” is the ratio of single to double real time.

TABLE II
CPU ONLY TIMING TESTS

machine	nThread	nRepeat	real sec	CPU
panther	2	1	404	1.63
panther	2	2	790	1.65
panther	2	5	1981	1.65
zuul02	8	5	1158	4.26
gollum	8	5	887	3.38
zuul05	8	5	1659	3.50
zuul05	8	2	664	3.22
smeagle	8	5	1590	3.34

Notes: “nThreads” is the number of threads used in the computation, “nRepeat” is the number of times the plane computation was repeated, “real” is the run time, “CPU ” is the CPU/Real time ratio,

TABLE III
SINGLE BUFFERED IO ONLY TIMING TESTS

machine	disk	nRepeat	I/O sec	CPU	Total sec	fract
panther	SSD	1	167	404	559	0.072
panther	SSD	2	167	790	959	-0.012
panther	HD	2	548	790	1338	0.000
panther	SSD	5	167	1981	2146	0.012
panther	HD	5	548	1981	2554	-0.045
zuul02	lustre	5	450	1158	1451	0.349
gollum	RAID	5	393	887	1150	0.331
zuul05	RAID	5	87	1659	1752	-0.070
zuul05	SSD	5	117	1659	1831	-0.470
zuul05	RAID	2	87	664	751	0.000
zuul05	SSD	2	117	664	793	-0.103
zuul05	lustre	2	167	664	815	0.102
smeagle	RAID	5	124	1590	1724	-0.081
smeagle	SSD	5	121	1590	1711	0.000
smeagle	lustre	5	127	1590	1735	-0.142

Notes: “I/O” is the run time for single buffered I/O only, “CPU ” is the CPU only run time, “Total” is the run time single buffered with nRepeat computations, “fract” is the fraction of the I/O time reduced by overlapping computation and I/O.

IV. DISCUSSION

A number of tests were performed involving both a large amount of I/O and computation using single and double buffering. In none of the tests performed did double buffering make a significant reduction in the run time.

The standard Unix read-ahead was disappointing; this reduced the I/O time on zuul02 and gollum by about 1/3 whereas there was no benefit seen in the other tests. These tests were dominated by the compute time allowing ample opportunity to overlap I/O and computation. This result is not understood.

Other observations include:

- The RAID disk on zuul5 outperformed the SSD.
- The pathetic CPU speed on zuul05 is seriously undermatched to the blazing I/O speed.
- Past speed issues for the lustre file system seem to be resolved, at least for zuul05 and for these tests.
- The I/O performance of smeagle exceeded expectations, especially on the lustre file system. However, the CPU

speed is far less than expected; the “CPU only” test took nearly twice the time of the test on gollum with significantly slower, older cores and with half the memory bus width.

REFERENCES

[1] W. D. Cotton, “Obit: A Development Environment for Astronomical Algorithms,” *PASP*, vol. 120, pp. 439–448, 2008.

Fig. 1. Double Buffered Test Routine

```
def testDouble(inImage, doDouble, nr, err):
    """
    Test double buffering

    reads cube and determines RMS per plane
    * inImage   input cube
    * doDouble  True for double buffering
    * nr        Number of repeats of RMS
    * err        Python Obit Error/message stack
    """
    inImage.Open(Image.READONLY,err)
    t0 = os.times()[4]    # Initial time
    # Create buffers
    nplane = inImage.Desc.Dict['inaxes'][2]  # Number of planes
    doPlane = [1,1,1,1,1]
    buf1 = FArray.FArray("Buff1",inImage.Desc.Dict['inaxes'][0:2])
    if doDouble:
        buf2 = FArray.FArray("Buff2",inImage.Desc.Dict['inaxes'][0:2])
        buffers = [buf1,buf2]
    else:
         buffers = [buf1]
    ibuff = 0
    th = None
    # initial read
    inImage.GetPlane(buffers[ibuff],doPlane,err)
    # Loop
    for iplane in range(0,nplane):
        # If double start next read in a thread
        if doDouble and iplane<nplane-1:
            doPlane[0] = iplane+2; nextBuff = 1-ibuff
            th = threading.Thread(target=inImage.GetPlane,\
                          args=(buffers[nextBuff],doPlane,err))
            th.start()
        # Work
        RMS = 0.0
        for j in range(0,nr):
            RMS += buffers[ibuff].RMS
        if not iplane%20:
            print 'plane',iplane, 'RMS',RMS/max(1,nr),\
               't=%7.1f sec'%(os.times()[4]-t0)

        # If double, join
        if doDouble and th:
            th.join()
            ibuff = nextBuff; del th; th = None
        # otherwise read next until last
        elif iplane<nplane-1:
            doPlane[0] = iplane+2
            inImage.GetPlane(buffers[ibuff],doPlane,err)
    inImage.Close(err)
```