SGP MEMO NO. 19

MORE THOUGHTS ON DISK I/O EFFICIENCY

Written by Jim Torson
Version 1
July 22, 1982

This is an attempt to be more quantitative about the disk I/O efficiency question. Given that we have a certain fixed amount of memory that can be used for buffers, is it better to fill the entire buffer in each I/O call or to fill half the buffer and do double buffering? There is no single answer since the relative performance varies depending upon the ratio of CPU time to I/O wait time.

The diagram shows a plot of normalized total job time for double buffering as a function of the ratio of CPU time to I/O time. The normalized total job time for double buffering is defined as the job time for double buffering divided by the job time for single buffering. The CPU time to I/O time ratio is defined in terms of times for the single buffering case.

For CPU-I/O ratios of 2.0 and greater, double buffering is faster. At 2.0, double buffering reduces the job time by one third. At larger ratios, there is a smaller improvement. In this region, the gain in speed results from the "disappearance" of the I/O time since the I/O is done concurrently with the computing. The larger the amount of computing, the smaller the fractional improvement that we see by eliminating the I/O wait time.

Things are more interesting for CPU-I/O ratios that are smaller than 2.0. In this case, the performance of double buffering varies depending upon the I/O time to get a half-buffer for double buffering versus the time to get a full buffer for single buffering. The lower curve assumes that the I/O time for the half-buffer is half as long as the I/O time for a full buffer. The maximum gain for double buffering occurs at a CPU-I/O ratio of 1.0. In this case, double buffering makes the job twice as fast. As the CPU time becomes neglibible compared with the I/O time, the double buffering performance becomes the same as the single buffering performance.

In reality, we cannot read in a half buffer in half as much time. Some time ago Barry did some experiments which gave the following results:

Buffer size (disk blocks)	I/O time (ms)
1	17
2	20
4	19
8	22

Thus, the real situation is closer to the top curve, which assumes that the I/O time is the same for a half buffer as for a full buffer. When the CPU-I/O ratio is 1.0, the performance of single buffering and double buffering is the same. For negligible CPU time, double buffering causes the job to take twice as long.

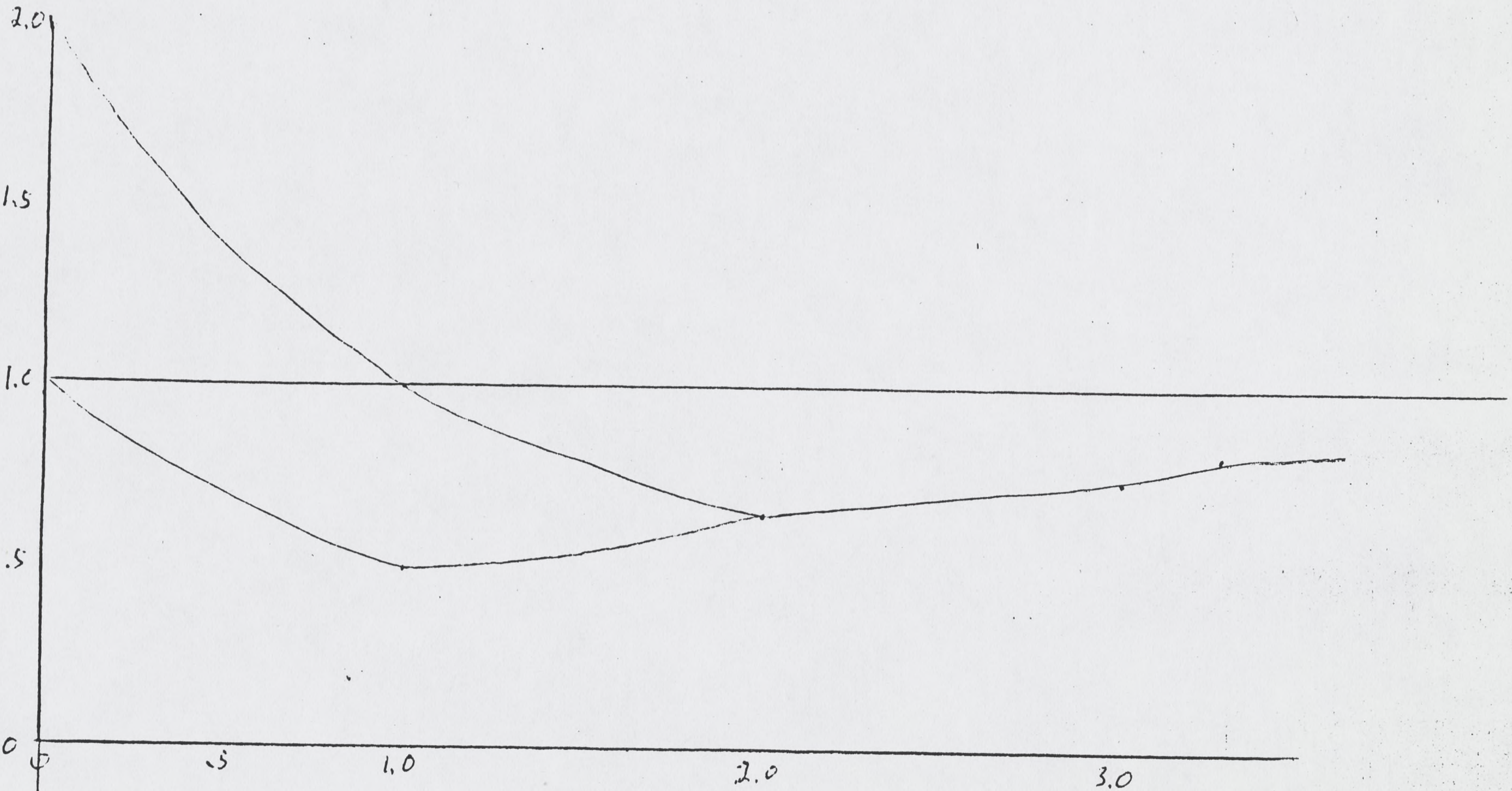
What will the CPU-I/O ratio be for our tasks? Perhaps the calling program should tell the I/O routines whether single buffering or double buffering is to be done. How accurately can the person coding a task guess the CPU-I/O ratio? Of course, if it is just an option specified in an initialization call, he could easily change it and see which way runs faster for a given task.

The above of course assumes that the data files are allocated as contiguous files and that most I/O requests ask for an integral number of disk blocks beginning on a block boundary. In this situation, the I/O time does not vary significantly for different size transfers. On another system (e.g., the VAX) these might not be valid assumptions.

The above also assumed that the task in question was the only thing running in the computer. What happens if you have multiple tasks accessing the disk? For heavy disk useage, each I/O request will take longer since the heads have to be moved. For our disks, the average head seek time is 30 milliseconds, with a maximum of 55 milliseconds. Thus, the I/O time might be doubled. This will mean the CPU-I/O ratio for a given task will be lower. Thus, more tasks would be faster if single buffering were used.

If single buffering is used, there will be half as many I/O calls in a system. For a system with multiple concurrent tasks, there will thus be fewer disk seeks. Nobody can access data when the heads are moving, so this would allow a higher overall disk I/O throughput.

Comparison of Double Buffering and Single Buffering



Carl

THOUGHTS ON DISK I/O EFFICIENCY

Written by Jim Torson
Version 1
July 16, 1982

Three ways to improve the I/O efficiency (speed):

1. Read an integral number of blocks beginning on a block boundary. Data is transferred directly into the buffer rather than being broken up into two or three separate disk transfers. Eliminates copying data from the one block intermediate buffer used by Al's routines.

2. Read in larger buffer with each input. You wait fewer times for the disk to turn. This works because transferring several blocks in an I/O operation is frequently nearly as quick as transferring a single block. If the time spent waiting for the I/O was a significant part of the total task time, you can get substantial speed improvements. There may be problems with this technique when reading a file "backwards," i.e., reading line by line top to bottom.

3. Double buffer. You do some computation concurrently with waiting for the I/O. If the buffer is the same size as with single buffering, you wait for the disk the same number of times. You can at best only double the speed of a task - only when the compute time is about the same as the time spent waiting for the I/O. In most cases, improvement is smaller. This of course requires two buffers. Thus the maximum line size that can be processed may be half as much.

Ways to implement reading larger buffers:

L-1. Read into a large "hidden" buffer. User asks for lines or partial lines of data. Data copied into user's buffer as needed. (Some calls just copy data from system buffer to user buffer.)

Advantages:

a. Call is simple - user program can easily process map lines.

b. I/O routine is fairly easy to write.

Disadvantages:

a. Uses too much memory space. (May not be a problem if buffer is put in a core-resident overlay.)

b. Extra CPU execution to copy data to user's buffer.

L-2. User specifies a starting location and a pixel count. Data is read directly into the user's buffer (if an integral number of blocks starting on a block boundary is requested). (Eric's proposal.)

Advantages:

a. I/O routine is very easy to write.

b. No CPU execution to copy data from system buffer to user buffer (if integral number of blocks starting on a block boundary).

c. Caller can control the size of the buffer.

Disadvantages:

a. Extra book-keeping needed by the caller if he wants to process in terms of map lines and if a buffer holds more than one map line. Also extra book-keeping if buffer is not an integral number of lines. Also extra book-keeping if reading lines top to bottom.

L-3. User specifies a starting location. Data is read directly into user's buffer. Each call "gets" one map line (or a partial line). A pointer to the line is returned to the caller. Some calls just return an updated pointer.

Advantages:

a. No CPU execution to copy data (if integral number of blocks starting on block boundary).

b. Caller can control the size of the buffer.

c. Easy for caller to process forwards in terms of map lines. In this case the book-keeping is handled by the I/O routines.

d. If the I/O routines know that map lines are being read top to bottom, then the I/O routines could handle the book-keeping for reading backwards.

Disadvantages:

a. Calling program is a little bit more complex than if a map line is returned at the beginning of the user's buffer.

Ways to implement double buffering:

D-1. Similar to L-1. Read into 2 "hidden" buffers. User asks for lines or partial lines. Data is copied into user's buffer as needed.

Advantages:

a. Call is simple - user can easily process map lines.
b. I/O routine is fairly easy to write.

Disadvantages:

a. Uses too much memory. (May not be a problem if buffers are put in a core-resident overlay.)
b. Extra CPU exucution needed for copying data.

D-2. Similar to L-2. User specifies a starting location and a pixel count. Data is read directly into user's buffer (if integral number of blocks starting on block boundary). The I/O is queued. User must call a wait routine before using the data. (The same call could be used for non-queued I/O, in which case it functions the same as L-2.)

Advantages:

a. I/O routine is very easy to write.
b. No CPU execution to copy data (if integral number of blocks starting on block boundary).
c. Caller can control the size of the buffer.

Disadvantages:

a. Extra complexity in calling program to keep track of the two buffers.
Extra book-keeping needed by the caller if he wants to process in terms of map lines and a buffer holds more than one map line. Also extra book-keeping if buffer is not an integral number of lines.
Also extra book-keeping if reading lines top to bottom.

D-3. Same as D-2 except a read of a single line (or partial line) is queued on each call. (This is the proposed modified IMPS I/O.) (This is similar to old IMPS I/O except only transfer of a complete line was permitted.)
(The same call could be used for non-queued I/O, in which case it would appear to the user to function the same as L-1.)

Advantages:

a. I/O routine is very easy to write.
b. No CPU execution for copy of data.

Disadvantages:

a. If a map line (or the desired piece of a line) is much smaller than the buffer size, then part of the buffer space is not used.
b. Extra complexity in calling program to keep track of the two buffers.

D-4. User specifies a single buffer. Each call gets half a buffer worth of data by waiting for the I/O that was queued on the previous call and then queueing the next I/O which goes directly into the other half of the user's buffer. A pointer to the data is returned to the user. (This is similar to the way FITS works.)

Advantages:

a. I/O routine is fairly easy to write.
b. No CPU exucution to copy data.
c. Caller can control size of buffer.
d. Calling program book-keeping is only a little more complex than L-1 or D-1.

Disadvantages:

a. User program book-keeping is a bit more complicated if you want to allow reading map lines top to bottom.

D-5. Similar to L-3. User specifies a starting location. Data is read directly into user's buffer. Each call "gets" one map line (or partial line) by waiting for a previously queued I/O and then it queues the read of the next line. A pointer to the data is returned to the user. (AIPS I/O is similar to this except that lines (or partial lines within a map subsection) are read sequentially backwards or forwards.)

Advantages:

a. I/O routine is fairly easy to write.
b. No CPU exucution to copy data.
c. Caller can control size of buffer.
d. Calling program book-keeping is only a little more complex than L-1 or D-1.

Disadvantages:

a. If a map line (or the desired piece of a line) is much smaller than the buffer size, then part of the buffer space is not used.

Note that we could also consider schemes that are combinations of some of the basic types of schemes described above. For example, we could combine D-5 and L-3. An I/O operation will actually read in the maximum number of lines that will fit in half the buffer that is supplied by the user. Some calls would then just need to return an updated pointer which is within the same half-buffer. Other calls would have to wait for completion of the read into the other half-buffer.

In addition to considering the basic types of I/O schemes described above, we can consider various ways for the caller to specify the desired pieces of the map file to be read in. The following are some of the possible options:

Ways for caller to specify location in disk file for a read operation:

1. No specification. First read gets a block of data starting at the beginning of the file. Next read gets the next sequential block, etc.

2. Pixel number or byte address.

3. Column number, line number and channel number.

Places where starting location for read could be specified:

1. Nowhere. Applies to sequential reading of blocks of data.

2. In a single initialization call which specifies the map subsection that is to be processed. (This is the way AIPS I/O works.)

3. In a location-setting call that is separate from the read. (This is the way Eric's proposed I/O routines work.)

4. In the read call. (This is the way the old IMPS I/O works.)

Ways for caller to specify the amount of data to be read in:

1. No specification. Each call could get a single entire map line. (This is the way the old IMPS I/O works.) Or, each call could fill the entire buffer with pixels. (This is the way FITS works.)

2. Pixel count or byte count. (Eric's I/O routines.)

Types of input processing that we want to handle:

1. Line at a time in forward direction.
2. Partial line at a time in forward direction.
3. Sequential without regard to lines, columns, channels.
4. Partial line at a time in backwards direction.
5. Partial line at a time at "random" locations.

Now, which type of I/O scheme would be "best" for our use?

L-1. The need for an extra buffer makes this scheme undesirable. However, we might be able to eliminate this problem by putting the buffer in a core-resident overlay. (Also, the DISPLAY computer is not yet running with a large amount of physical memory. Things look promising though.) This would still leave the problem of extra CPU execution to copy the data from the system buffer to the user's buffer. If 16-bit data is being read from the disk and converted to floating point, then this isn't a problem since we have to access the data for conversion anyway. How much of our disk data will be in 32-bit floating point format? And, how much of a problem is it to copy data from the system buffer to the user buffer? Perhaps this scheme should be considered further.

L-2. For sequential processing of an entire map, this scheme would allow both a simple application program and efficient operation. For accessing a map line by line, the calling program could be kept simple by just having each call ask for the number of pixels in a line. However, in cases where a buffer could hold several map lines, the efficiency would suffer greatly since each I/O call would fill only a small part of the available buffer. If we are processing a small continuum map, this would probably not be a serious problem since there is a small amount of data anyway. However, in the spectral line case, the total amount of data to be processed could be large even though the line size is small enough that several lines would fit into the buffer. In order to improve the I/O efficiency, the application programs would have to ask for multiple lines in each I/O call and then do the book-keeping themselves. This scheme is thus not desirable for processing line by line.

L-3. For processing line by line, this scheme would require a slightly more complex calling program than would scheme L-2. The structures of the programs would be the same, only the data accessing would be different. An L-2 program would access the I'th element in the buffer by saying BUFF(I) and an L-3 program would say BUFF(I+OFFSET), where OFFSET is the pointer to the beginning of the current line being processed. An advantage of this scheme would be that reading in multiple map lines with each call to the I/O system would be automatically handled without any additional complexity for book-keeping in the calling program. We could view this scheme as similar to L-2 except that the I/O routines take care of the book-keeping needed for reading in multiple lines. A program that is not concerned with line numbers, column numbers and channel numbers could still be written to process in terms of lines. This would result in extra subroutine

calls since the read routine would be called once for each map line rather than once for each buffer worth of map lines. A problem would be that the size of the map that could be processed would be limited by the line size that would fit in the buffer. This scheme seems to have worthwhile advantages and will be considered further below.

Before evaluating the various type of double buffering schemes, let's consider double buffering in general. Given that you have a certain fixed amount of memory for buffers, is it faster to fill the entire space in one read or to do double buffering with buffers half as big? If the computing that the task has to do is negligible compared with the I/O and if the file is contiguous on the disk, then single buffering is better since you would wait for the disk to turn fewer times. If the file is scattered all over the disk then it probably doesn't make much difference.
If the compute time is about the same as the I/O time, then single buffering will give you N cycles of read-compute, where each cycle is half read and half compute. If the read and the compute each take one unit of time, then the whole job will take 2N units of time. Double buffering would give 2N cycles of concurrent reading and computing. The compute part in each cycle will take half as long as in the single buffering case. If each read (for half as much data) takes half as long, then we will have 2N cycles which are each one half unit of time long, i.e., half as long as the single buffering cycles. The whole job would then take N units of time and would be twice as fast as single buffering. However, an access to fill the half-size buffer actually takes nearly as long as an access to fill the full buffer. Thus, total job time for double buffering would be nearly as long as for single buffering. If the compute time is large compared with the I/O time, then double buffering still does not help much. It thus appears that for sequential processing of contiguous files, double buffering does not provide a significant gain in speed over doing single buffering with a buffer that is twice as big.

The FITS program can have three concurrent things going on at one time: a disk read, computation (conversion of previous disk buffer to tape format), and a tape write (previously converted data). Does this really speed things up over just using a larger disk buffer? (The size of the tape buffer is fixed by the FITS definition.)

There may be a case where double buffering gains you nearly a doubleing in speed. Suppose you are processing a subsection of a large map and two partial map lines will fit in your buffer space. If your buffer isn't big enough to hold enough

data to contain more than one of the partial lines of interest then for single buffering you would still have to make an I/O request to read in each partial line individually. If the compute time is about the same as the I/O time then double buffering could double the speed of the program.

If there are other tasks running in the system, double buffering may even make things slower since you will have more separate I/O requests and thus more time spent waiting for head seeks.

D-1. This double buffering scheme has the same problems that L-1 has. If you have the memory, L-1 would be a better scheme to use.

D-2 and D-3. These schemes are undesirable because they cause too much complexity in the calling program for keeping track of the two buffers.

D-4. This scheme is undesirable since it requires too much complexity in the calling program for the case of reading line by line.

D-5. As discussed above, in most cases, scheme L-3 is better. However, this might be useful for reading partial lines if a partial line would fit in a half buffer but the whole buffer cannot hold a contiguous piece of the file which contains more than one of the partial lines of interest.

Before describing a proposed set of routines for our use, let's list the main concepts to be used in the design:

1. Many applications are most conveintly done by processing in terms of map lines. The routines will be based on the L-3 scheme. When the caller asks for a line of map data, the available buffer will be filled with map lines so that subsequent requests from the user can just return an updated pointer without actually accessing the disk. In some cases, the D-5 scheme will be used. The caller will not need to know which scheme is being used by the I/O routines.

2. There will be a separate call to access the map data sequentially without regard to map lines. This will allow easy coding of things like a copy program which will not have any limit on the size of the map that can be processed.

3. A map line buffer similar in concept to the old IMPS scheme will be used. The buffer will consist of space for some book-keeping data in addition to the actual data buffer. One of the book-keeping items will specify the size of the data buffer in this particular map line buffer. All tasks will be coded so that they can process at least a partial line that contains 1024 floating point numbers. However, some tasks may use larger map line buffers for effiency. Some tasks such as map copy or FITS will be capable of sequentially processing any size map. Other book-keeping items in the map line buffer will keep track of what part of the map file is currently contained in the data buffer.

4. When a map is opened for read, the caller can specify the type of line by line access to be used: forwards, backwards, or random. For forwards or backwards, the read routine will fill the buffer with lines either beginning or ending with the requested line. For random access, the routine will only read in the requested data. (This will be useful for reading corresponding lines along the frequency axis of a spectral line cube.) If the caller does not specify the mode, or specifies it incorrectly, things will still work properly but there may be a loss of efficiency.

5. We will avoid having Al's I/O routines break up the disk disk transfer into two or three pieces whenever possible. This will be done by asking for an integral number of blocks that begin on a disk block boundary.

In addition to the access mode specification in the map open routine, the following are some proposed routines:

SUBROUTINE SRDMIO(MID, ML, NUM)

This routine does a sequential read of the map file. ML is a map line buffer. The buffer is completely filled with pixel data. The pixels are converted to floating point format if necessary. NUM returns the number of pixels that have been read in. This is the number of floating point pixel that will fit in the buffer except that there might be a partial buffer at the end. This returns zero when you have reached the end of the file.

SUBROUTINE RDMIO(MID, COLUMN, LINE, CHANEL, NUM, ML, OFFSET)

This routine "reads" a line or partial line of the map. COLUMN, LINE and CHANEL specify the starting location and NUM specifies the number of pixels desired. If COLUMN is one and NUM is MAPNX then it gets an entire line. ML is the map line buffer that returns the desired data (in floating point format). OFFSET is the index into the map line data array which points to the element just prior to the first element being returned. For example, if we used the files ML1.DCL and ML1DAT.DCL to define the map line buffer, the following would read in line 5 of a map and calculate the sum of the pixel values in the line:

```
        CALL RDMIO(MID1, 1, 5, 1, MAPNX, ML1, OFFSET)
        TEMP = 0.0
        DO 100, I = 1, MAPNX
100             TEMP = TEMP + ML1R(I+OFFSET)
```

Another way to code the loop might be:

```
        DO 100, I = 1+OFFSET, MAPNX+OFFSET
100             TEMP = TEMP + ML1R(I)
```

The following is a sketch of the algorithm that might be used by RDMIO. For simplicity, the following assumes we are only using the L-3 type single buffering. It is left as an exercise for the student to put in the D-5 type double buffering where appropriate.

```
1. Check for proper access type:
If access type is not initialized
    then set access type (in MID) to "line-by-line"
    else if access type is "sequential"
        then error;


2.  See if we already have desired data:
If specified data is already in the buffer
    then return a pointer to it;


3.  See if buffer is big enough:
If buffer will not hold NUM floating point numbers
    then error;


4.  Calculate file location of requested data:
START = file address of first requested pixel;
NBYTES = number of bytes in requested number of pixels;


5.  Align on disk block boundary:
If NBYTES at START is not an integral number of disk blocks
    then begin
        Save START and NBYTES;
        Adjust START and NBYTES to describe integral number of blocks;
        If NBYTES of data (converted to floating point) wouldn't fit
                in the buffer then restore old START and NBYTES;
        end;


6.  Anticipate next request:
If direction is not "random"
    then begin
        If direction is "forward"
            then Increment NBYTES to include as many map lines as will
                        fit in buffer (in floating point format)
            else Decrement START to include as many map lines as will
                        fit in buffer (in floating point format);
        end;


7.  Read and convert the data:
Ask A1 for NBYTES of data beginning at START in the file;
Convert the data to floating point format (if necessary);
```