lustre®

Applications IO Performance on Jaguar

An investigation by the Lustre Center of Excellence

Lustre Center of Excellence – Oak Ridge National Laboratory

Table of Contents

Table of Contents

Introduction

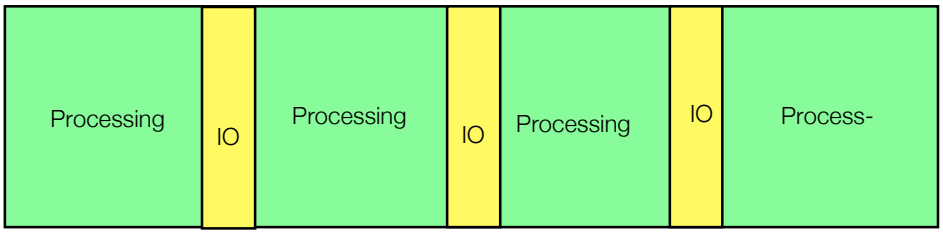
Users on the Jaguar and Jaguar PF systems at Oak Ridge National Laboratory (ORNL) are not always achieving satisfactory applications performance. The time to completion for applications is often too long and subject to wide variability. A single application processing a similar sized data set will vary in completion time by multiple factors on different runs. This is a source of dissatisfaction and frustration for the users.

The Lustre Center of Excellence (LCE) has started an investigation to identify the sources of the performance problems and variability. Mike Booth of the LCE is leading the project. This paper will be a living document and will document the planned tasks for the investigation, the status of the tasks and the results of the investigation.

Applications IO Model

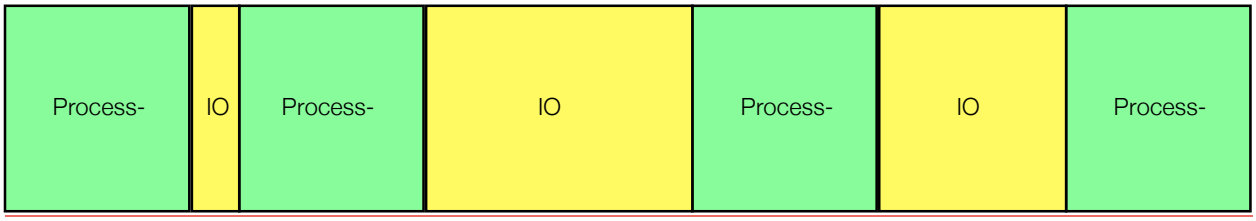
The initial focus of the investigation is the writing of restart files. The writing of these files is a majority of the IO performed by many of the applications. Also, the current IO model used for these restart files means that poor or variable IO performance when writing them will have a direct impact on the completion time of the applications.

At high level, many large scale parallel applications use a synchronous approach to writing out restart files. All of the processes (tens of thousands in a large run) will pause at a barrier, all will write their portion of the restart file and, when all the IO is complete, they will resume processing. This is illustrated in the figure below:



The variability of IO performance on Jaguar XT5 is often orders of magnitude, dependent upon variables not in the control of the user. If the IO to write the restart files is slower than expected or more variable than the total elapsed time for the application can expand significantly as illustrated below:

IO model for restart files (Assumes fast, consistent IO performance)



Application time expands with slower, variable IO performance

Summary of activities

The current and planned activities for the IO investigation fall into two categories.

1. Investigations into why the "System" is not performing
2. Investigations into how to achieve the best performance out of the "System" as it exists.

   The term "System" is used here to include all products involved in IO, from the Cray I/O system call, through the Lustre Client, to the OSTs, all the way to the disk drives.

Details for each investigation activity and results are kept in this section and will be updated on an ongoing basis.

The activities are summarized in the table below.

#	Activity	Status	Originator
1	GTC with ADIOS runs on Jaguar	Complete	Scott Klasky
2	Study Lustre Client caching	On hold	Mike Booth
3	Interaction of Lustre I/O with MPI collectives	On hold	Mike Booth
4	Laminar IO (prototype for more predictable IO)	On hold	Scott Klasky
5	Create a new Transport Method for ADIOS from LaminarIO	Active	Scott Klasky

#	Activity	Status	Originator
6	Benchmark HPCMP Tutorial (Henry Newman)	Complete	Dan Ferber

Follow up projects

#	Project	Status	Originator
1	Cache behavior/performance Benchmark	Proposed	Mike Booth
2	MPI/IO SeaStar Balance Performance Benchmark	Proposed	Mike Booth
3	Asynchronous Cache Page IO to benefit synchronous IO Applications	Proposed	Mike Booth
4	Coordinate IO performance investigation with Shane Canon at NERSC	Proposed	
5	Laminar IO in combination with direct IO	Proposed	

Activity details

1. GTC with ADIOS runs on Jaguar

STATUS: Complete

Learn how to run compile and run GTC. Link in ADIOS and make performance runs with modifications made by WangDi of the LCE to the ADIOS code.

Results were delivered to Scott Klasky.

2. Study Lustre Client caching

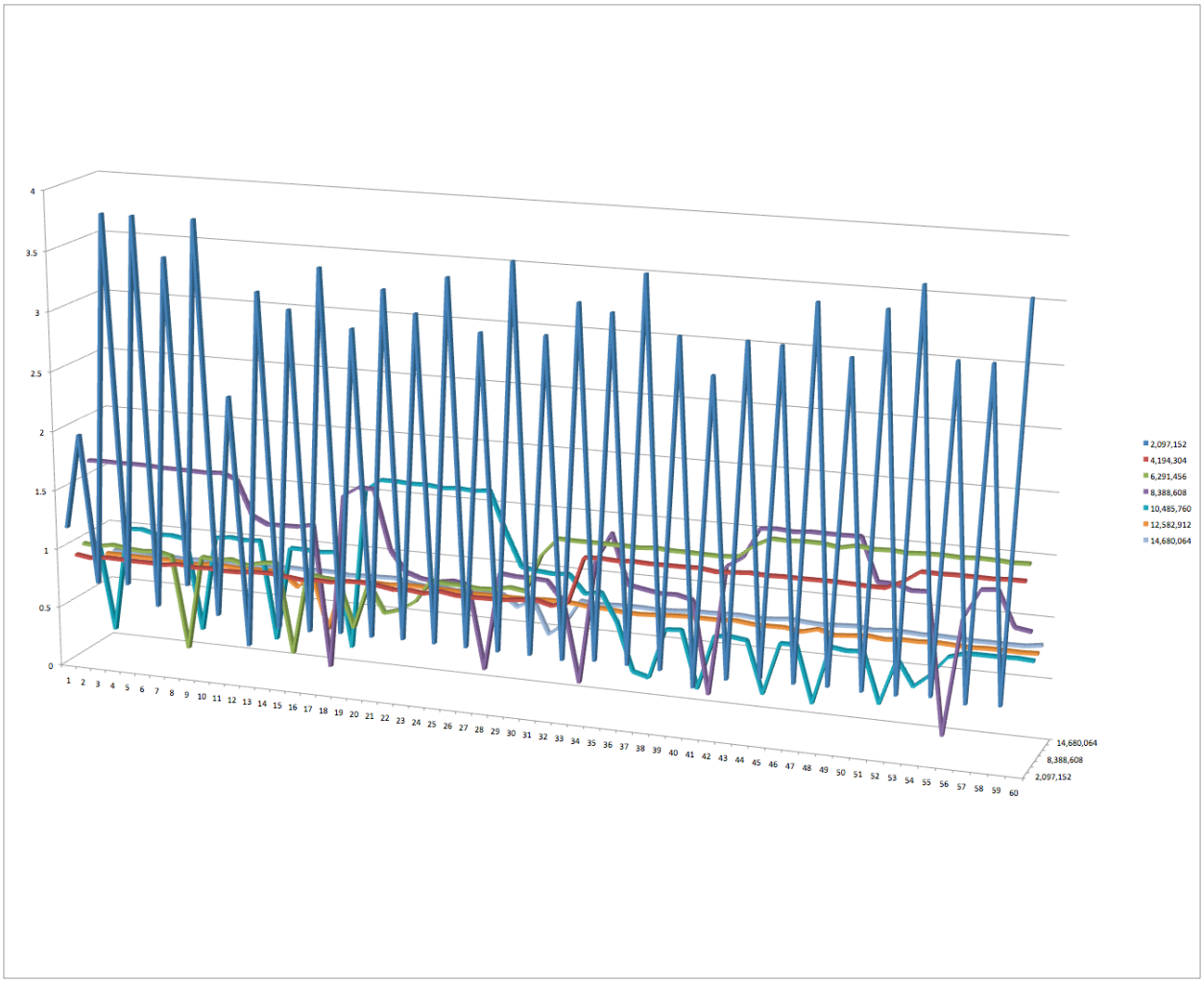
STATUS: On hold to pursue other activities

The observed cache management behavior of the Lustre client appears to be inconsistent with the design.

For example: It is expected that I/O cache pages should bleed out to disk in a first in/first out and page aging order. In studying erratic write performance on the system, performance tests focused more and more on understanding if the cache behavior was following design. If the I/O cached pages were asynchronously written from the system cache, the application would perceive I/O speeds equal to memory copy speeds when measuring the time across the write call when the system cache is clean. This was not what was being observed by the applications.

Why is this a significant issue? All I/O to the Lustre filesystem is through this Lustre client to the system cache management. If the behavior of this part of the overall system is erratic, conclusions built upon measurements made across the Lustre client are suspect.

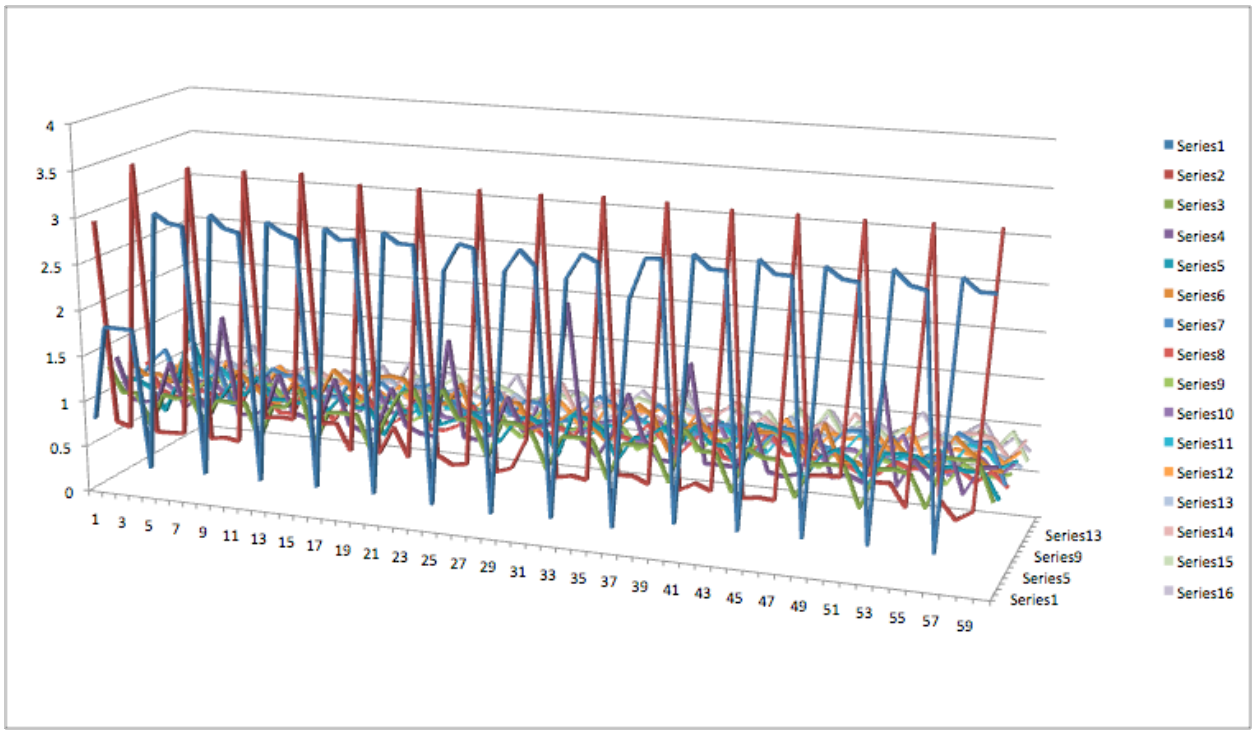
The following graph plots perceived I/O performance across write statements. Each line in the graph represents a different write size.



As the size of the record increases the perceived write speed stabilizes. There is no delay in the program between each timed write.

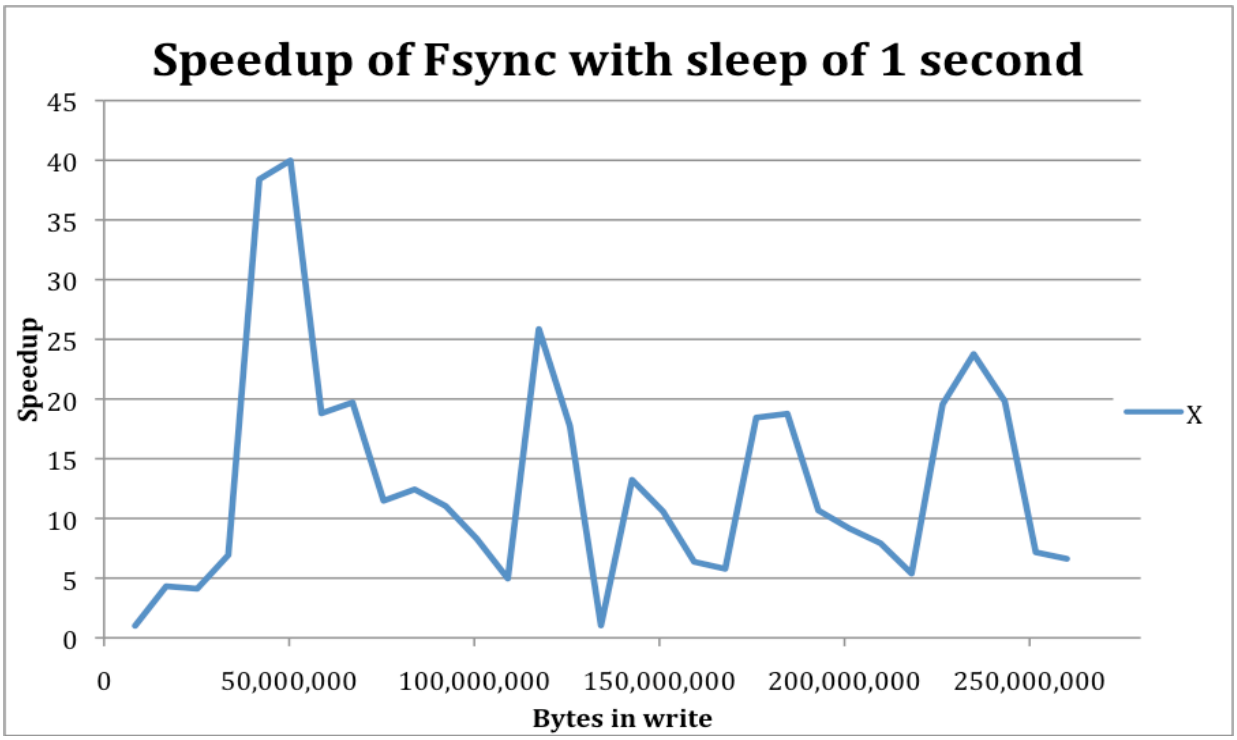
The same test, but between each write, a call to sleep(1 second) was made. The time to complete the average write on the larges write is 1/10 of a second. So, the time required to clear the buffer under the slowest conditions is greatly exceeded by the sleep of one second.

The plot with that run is below:



It can be seen is that the perceived performance is not the one expected by asynchronous buffers being emptied in parallel with other activities during the 1 second of sleep time.

A similar test was run, but measuring the time to complete a call to fsync. Fsync will complete when the system cache for the file has been flushed to disk.



This graph demonstrates the amount of data written to disk during the call to sleep of 1 second, as a function of dirty system cache. It appears that the amount of dirty cache to write to disk during the second of sleep is erratic, but some is written each time.

Andrew Uselton of NERSC has also measured behavior inconsistencies. Andrew's measurements are more conclusive demonstrating erratic system cache behavior. In Andrew's test, he serially wrote data to a file greater than the memory size on the writing client node. At this point in the run, it would be expected that for a first in/first out page management, the data in the system cache would be the tail end of the file. But when reading the file back in, reads to the beginning of the file are completed at memory copy speed, thus it was concluded that those pages were still in cache.

We met with Shane Canon at the Lustre User Group meeting to discuss how to share information and how to coordinate our activities with NERSC's investigations.

Development was started on a caching simulator. The cache simulator attempts to reproduce the first in first out cache page schema to see if response from Jaguar is consistent with design of Lustre and Linux cache schemes. Once we are able reproduce the behavior, the desire is to set page timeouts and see if a higher performance can be achieved for applications that write large restart files utilizing the asynchronous feature of a page cache.

STATUS: On hold

No progress, continues to be on hold due to focus on Laminar IO data method

3. Interaction of Lustre I/O with MPI collectives

STATUS: On hold

An operating assumption by many Lustre users and developers is that I/O cannot take place during time critical MPI collectives. The general belief is that it is faster overall to complete all IO before beginning computations that involve tightly coupled MPI collectives than it is to allow the IO to complete asynchronously with the computations. The understanding is that IO creates so much interference with MPI communications that the traffic jam creates a net loss of efficiency and an overall increase in the wall clock time to complete the job.

Mike developed a preadsheet to study the advantages of Asynchronous over Barrier Synchronous Interaction of Lustre I/O with MPI collectives. The spreadsheet demonstrates that effective use of cache for asynchronous io could result in dramatic savings of CPU hours and reduce the peak io performance required to satisfy the needs of applications writing restart files.

Oleg Drokin and Mike Booth did some test investigations that eventually lead to the following test.

1200 PEs on Jaguar

1. MPI Barrier by each PE
    - Barrier completes in a fraction of a second
2. Each PE writes 250Mbytes with a simple write() system call
    - Write returns very quickly, in less than 0.5 seconds. This performance is the result of memory copy of data to system cache.
3. MPI Barrier by each PE
    - This barrier completes in about 5 seconds
4. MPI Barrier by each PE
    - This barrier completes in about 2 seconds.

Conclusions from the above test:

1. The observation that MPI is dramatically impacted by IO occurring in the background is correct
2. The size of the impact implies an inappropriately high priority is given to IO over MPI.
    a. MPI barriers are almost completely stalled by the presence of IO
    b. As there is no priority scheme in the Cray SeaStar queue management
        i. The perceived Priority of IO must be due to either the size and/or quantity of requests of IO compared the size or quantity of the MPI collective calls. In effect IO has locked MPI out of the SeaStar queue.

- SeaStar needs to be studied as it appears that IO is able to overwhelm and lock out MPI traffic. At John Carrier's request, Kitrick Sheets is working with us to understand what is happening on the network.
- Programs to establish benchmarks of performance criteria for MPI interaction with IO need to be developed.

STATUS: On hold

No progress, continues to be on hold due to focus on Laminar IO data method

4. Laminar IO Prototype

STATUS: Hold, almost complete

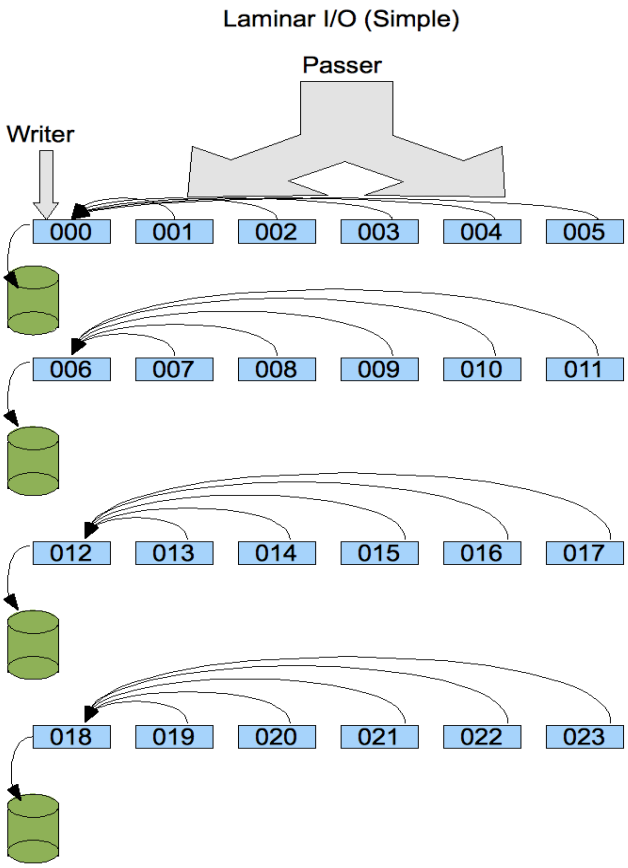
Prototype a new IO method with more consistent IO performance.

Problem: JaguarXT5 has smaller “islands” of performance, which appears to be due to the newer Infiniband network, and SATA drives. The same settings of stripe size and count will often yield wild variations in performance at > 10k PE.

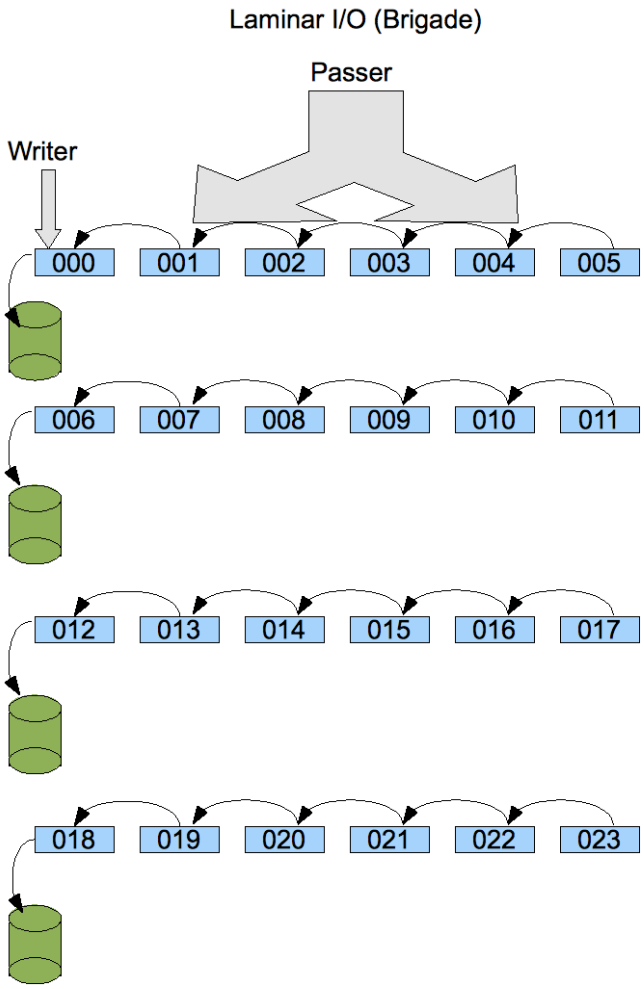
Per Scott Klasky’s observations, best performance is achieved when writing to the same number of files as there are OST’s on the Jaguar XT5 system (671 OSTs).

Develop a prototype to achieve reasonably predictable and consistent I/O performance during normal batch hours. The prototype aggregates IO to a limited number of nodes (roughly equal to the number of OSTs in the system). The unique attribute to this aggregation is that no additional memory is required for the aggregation. Current MPIO aggregation methods require that the aggregation node have enough memory to collect all the buffers from the other PEs.

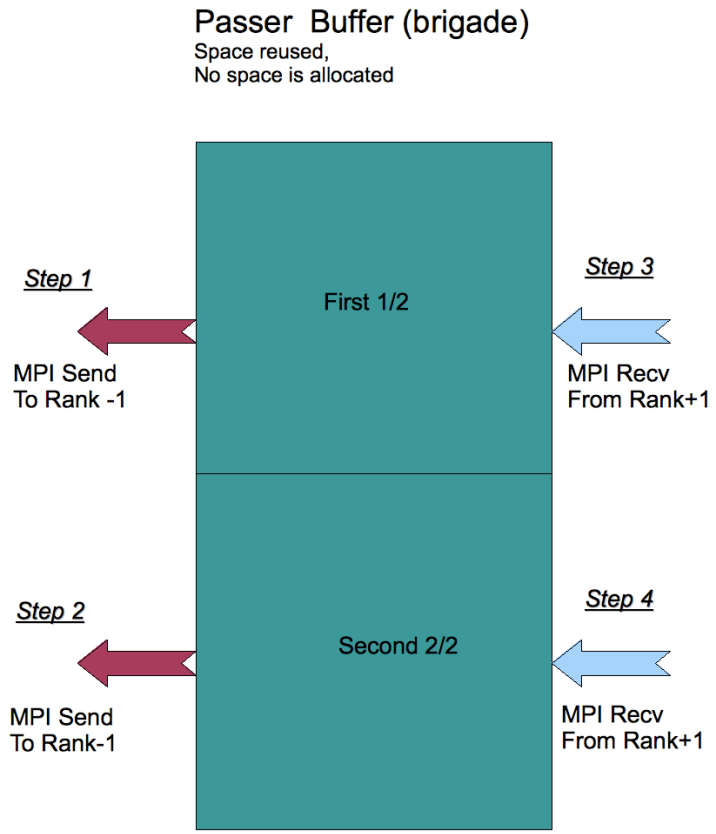
Two transport methods were used to move data from PEs to the aggregated writing node. The first was called simple:



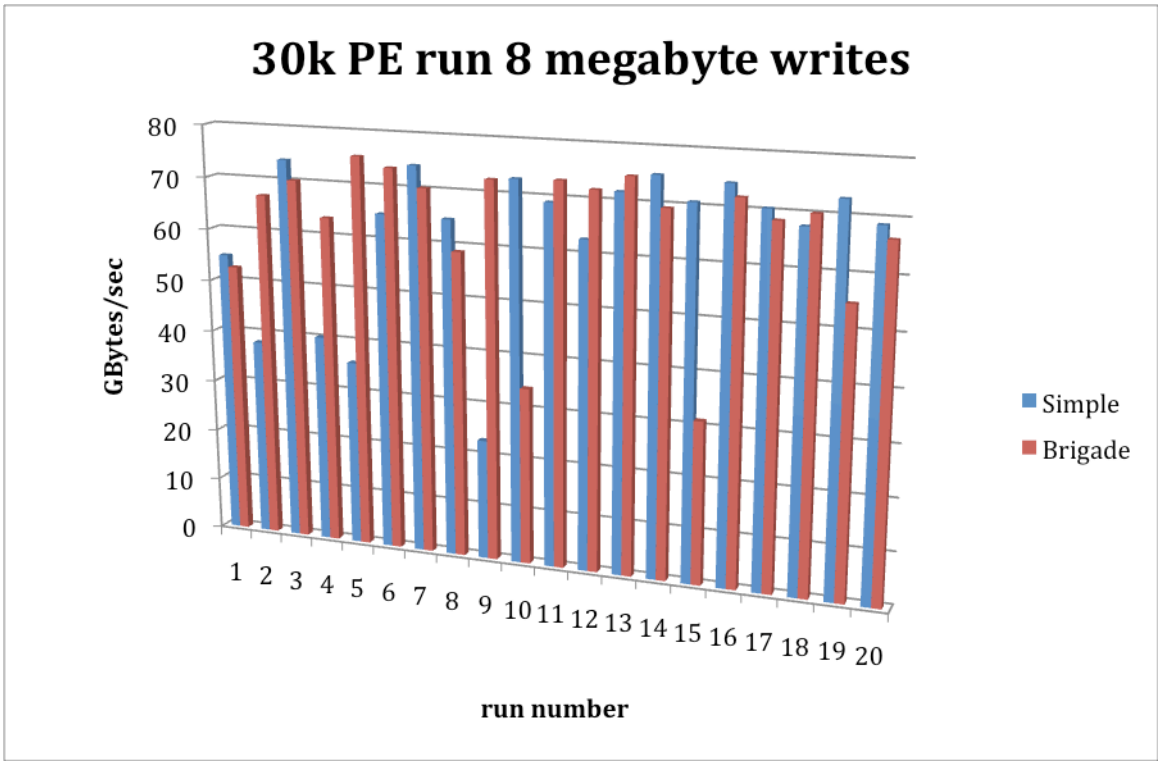
The second was called a brigade method, as it resembled a bucket brigade:


Laminar I/O (Brigade)
Passer
Writer
000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023

The buffer optimization to maintain asynchronous transfers and reuse of the memory space:



The best and most consistent results were achieved writing to most (450 OSTs) OSTs with a stripe count of 1. This performance was achieved with a relatively small amount of IO per PE (8 megabytes). These runs were made during the middle of the day, with many other jobs running on JaguarPF (XT5).



The most consistent results were achieved with the Brigade method as it has the smaller fall in performance. These results include file creation, close and fsync.

At the same time Jay Lofstead also wrote a method similar to these, except the writer rotated among the PEs to maintain consistent number of PEs writing to disk. This method also achieves good results and is implemented in ADIOS.

Next request is to see if Laminar IO can benefit from Direct IO methods.

STATUS: Hold

No progress, continues to be on hold due to focus on Laminar IO data method

5. Transport method for ADIOS from Laminar IO prototype

STATUS: Active

Use the ADIOS Developer Manual section from the ADIOS Manual to create a transport method with LaminarIO

Significant study on how to develop the transport model, first it was decided to use adios_mpio_stagger.c. This was tightly aligned with MPI methods and now pursuing adios_posix.c as the template to follow in implementation of Laminar driver.

Problems to be solved for Laminar as a method in ADIOS:

Prototype assumed each PE was writing the same size record. The prototype reused the space occupied by the write record as a transfer buffer. To have a limit on the size of the transfer buffer we must:

(1) Allocate a double buffer

(2) break a write record larger than one of the buffers into mutliple segments

(3) insert control flow into the data stream

(4) remove the control flow before write to filesystem

(5) control flow for seeks prior to writes in each PEs record

Plan:

The plan is to replace the posix writes in the adios_posix.c creating adios_posi_laminar.c with a routine that performs a brigade aggregation

If this is successful, a read method will be devised in a similar layout.

Definition of terms:

Writer = the 'rank' process that moves data to the Filesystem.

Passer = the processes that pass record data down stream to the writer.

Laminar 'write group' = one writer, n passers to that one writer.

Stream = stream of data from the writer created by the brigade passing of data to the writer then to the filesystem.

Control header = data inserted into the 'stream to demarc 'records', segments while passing stream data between processors

Record = a single rank's data contribution to the Stream

Segment = portion of a record that can transmitted in a single message passing call. This will be ½ the size of the malloc'ed space.

Order of file system operations

For each write group,

Seek to the writer's requested seek position

Write the writer's record to disk

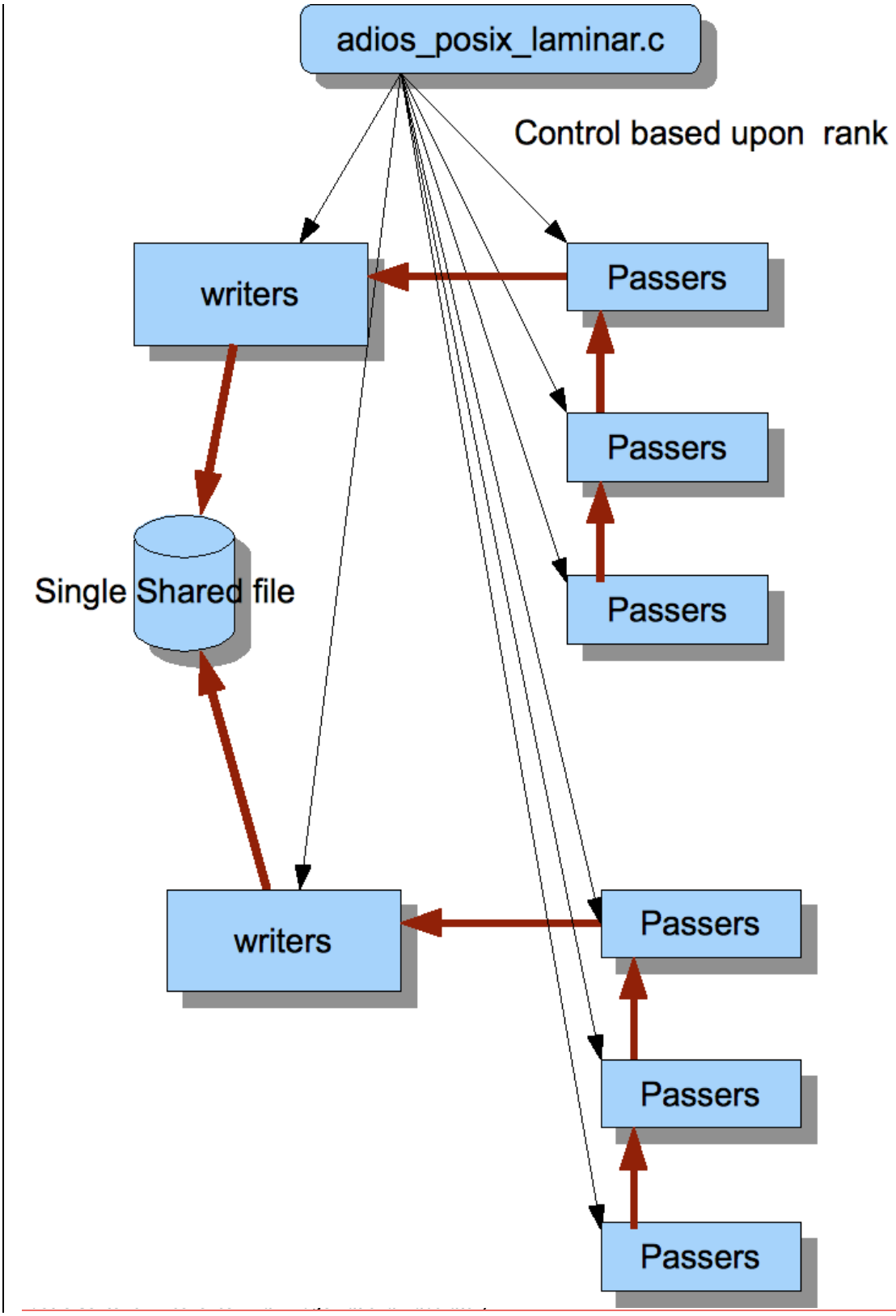
For each passer in the write group in order of

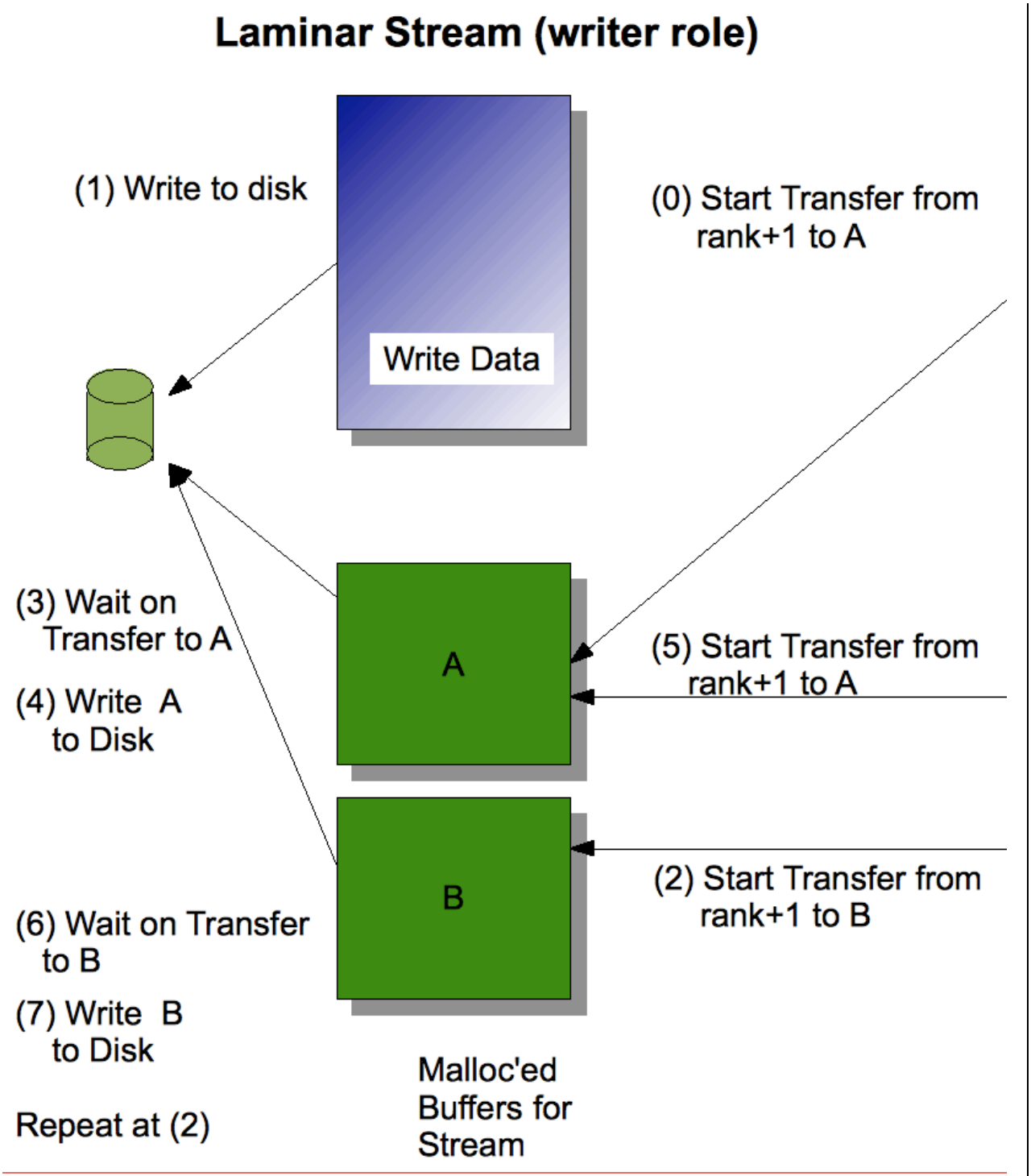
Smallest rank to largest rank.

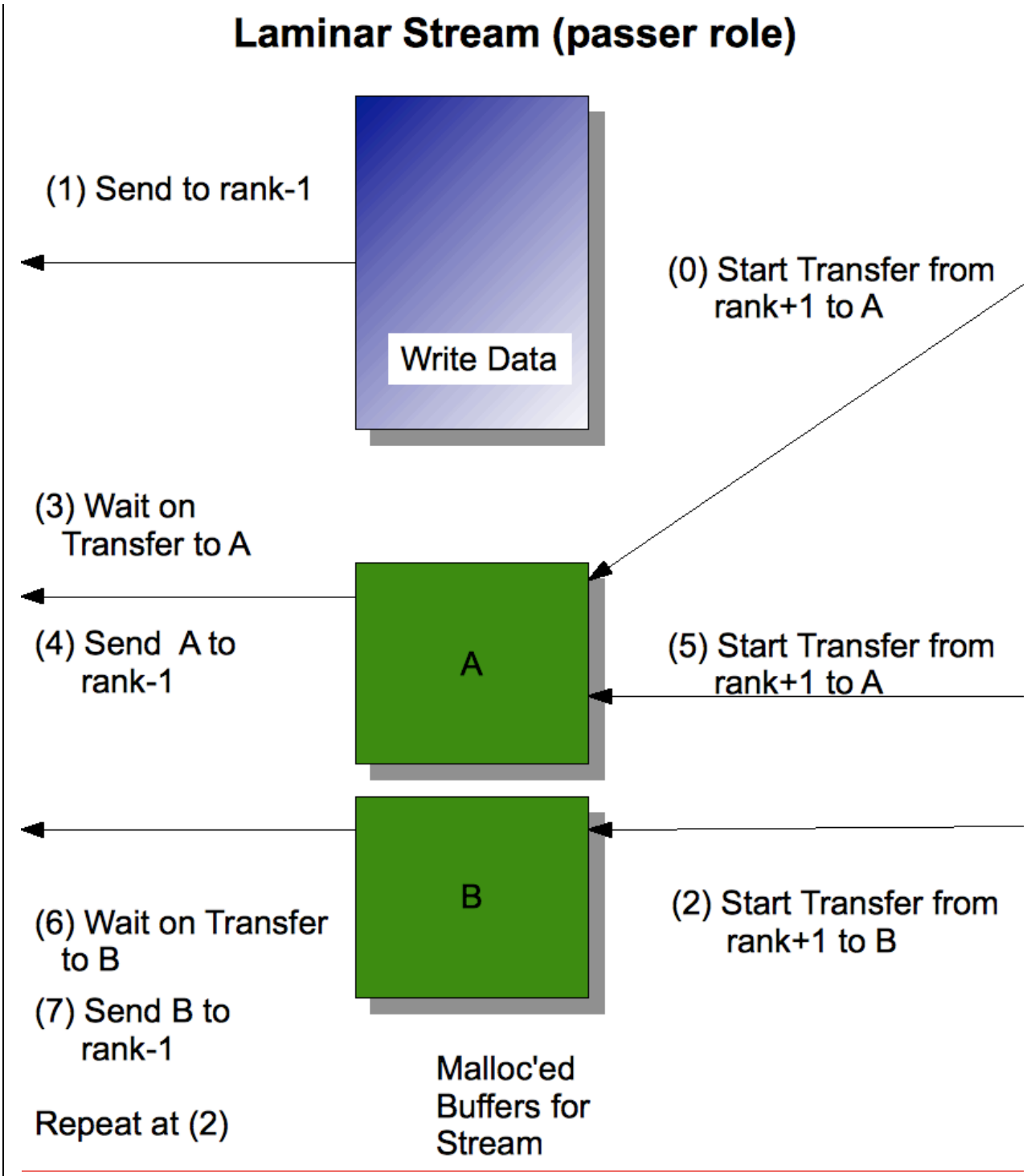
Seek to the passer rank's requested

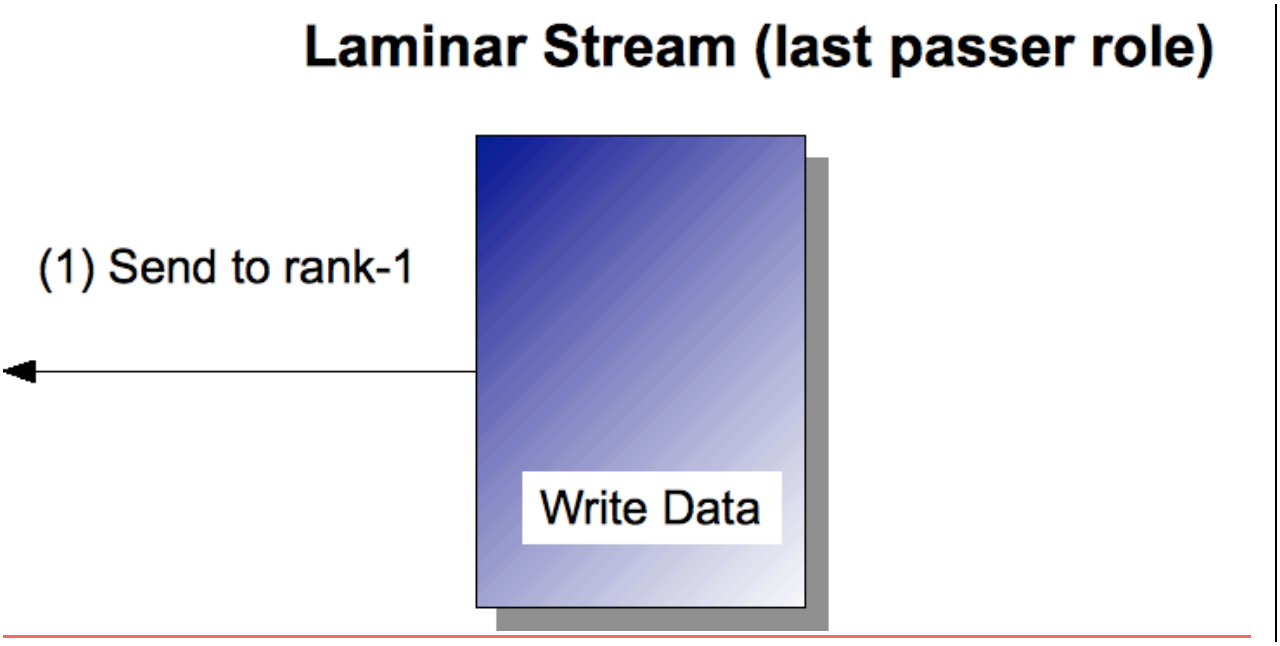
Seek position

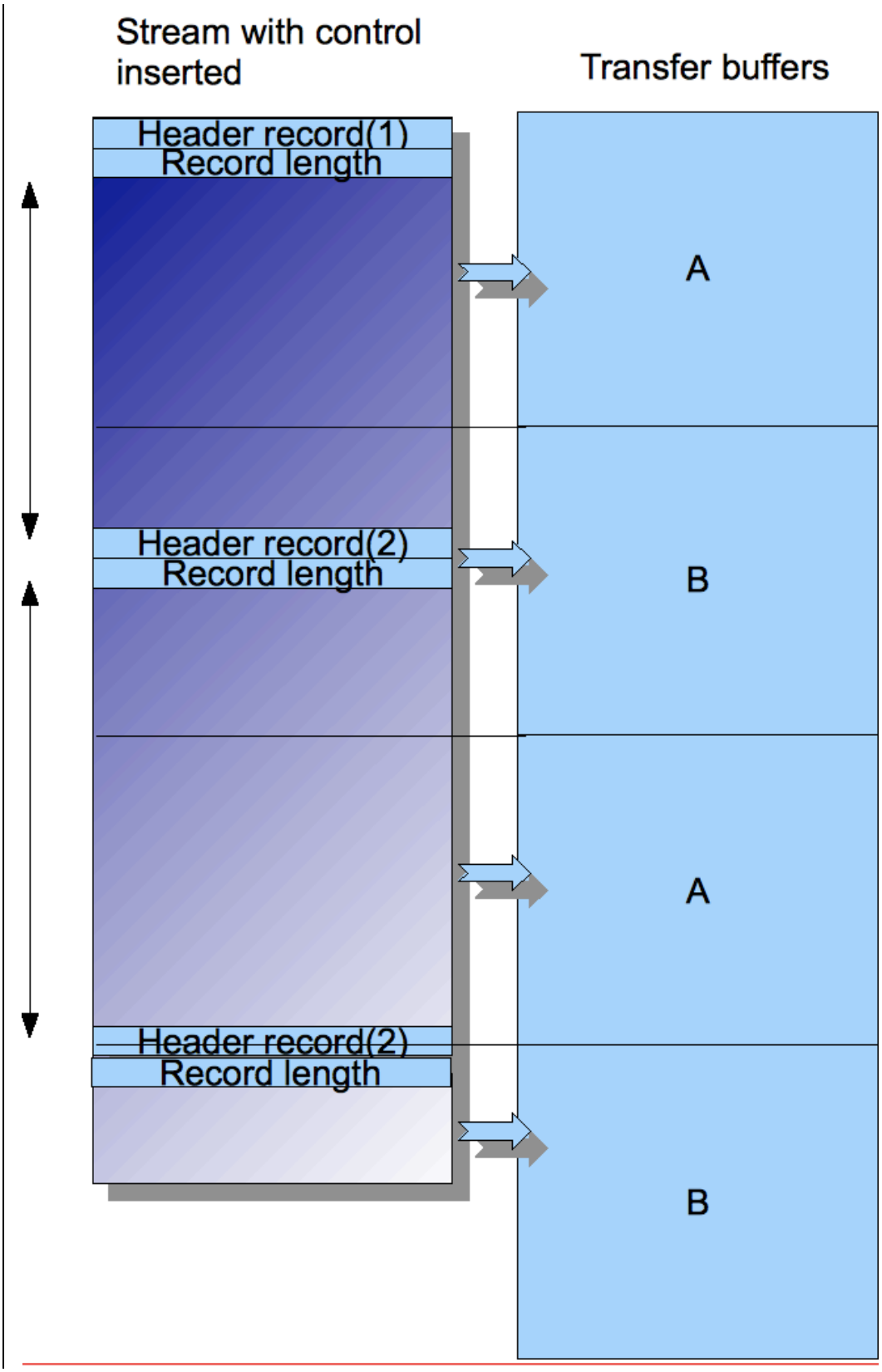
Write passer's record


adios_posix_laminar.c
Control based upon rank
writers
Passers
Passers
Passers
Single Shared file
writers
Passers
Passers
Passers


Laminar Stream (writer role)
(1) Write to disk
(0) Start Transfer from rank+1 to A
Write Data
(3) Wait on Transfer to A
(4) Write A to Disk
A
(5) Start Transfer from rank+1 to A
B
(2) Start Transfer from rank+1 to B
(6) Wait on Transfer to B
(7) Write B to Disk
Repeat at (2)
Malloc'ed Buffers for Stream


Laminar Stream (passer role)
(1) Send to rank-1
Write Data
(0) Start Transfer from rank+1 to A
(3) Wait on Transfer to A
(4) Send A to rank-1
A
(5) Start Transfer from rank+1 to A
B
(6) Wait on Transfer to B
(7) Send B to rank-1
Repeat at (2)
(2) Start Transfer from rank+1 to B
Malloc'ed Buffers for Stream


Laminar Stream (last passer role)
(1) Send to rank-1
Write Data


Stream with control inserted
Transfer buffers
Header record(1)
Record length
Header record(2)
Record length
Header record(2)
Record length
A
B
A
B

6. Benchmark for HPCMP tutuorial

STATUS: Complete

Email from Henry Newman requesting the work:

Mike,

Base on some information I got from a few users on HDF5, I am asking everyone to run the following test so I can show the user community that aligning data is a good thing.

Many threads open a single file
One thread writes 256 bytes
*Now all threads write the equivalent of 80% of memory say 1.5 GB with a starting offset of 256 bytes+(thread #*1.5 GB).*
Provide MB/sec and CPU time
Then do the same test but write a 4 KB header.

Is this possible for you to do or should I ask someone else. If you can do it please provide any tuning options you suggest for both cases.

Thanks in advance.

hsn

Henry Newman
Instrumental Inc/ CTO
2748 East 82nd Street
Bloomington, MN 55425
W 952-345-2822
F 952-345-2837

Henry indicated that the other people were using IOR as a method. After a few days of playing with IOR, I decided it would be easier to write a c program from scratch. With this, a method to assure that at least as many ost's were used as PEs, up to the amount of osts on the machine.

C code developed:

```
#include <stdio.h>
#include "mpi.h"
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>

#define _LARGEFILE64_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>

#define TEST_SIZE (1024*1024*1024)
#define STRIPE_COUNT 1024*1024
#define FALSE 0
#define TRUE  1
#define FPP     FALSE
#define OST_MAX 600
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
#define FIRST_WRITE  256
//#define HEADER      4*1024
// set this HEADER to zero when running without header
//
//   Prototypes
//
long lseek64              (int fd,
                           long offset,
                           int whence);
int  llapi_file_open      (const char *name,
                           unsigned long stripe_size,
                           int stripe_offset,
                    int stripe_count,
                           int stripe_pattern);
//
int main(int argc, char **argv)
{
void     *buf1, *buf2;
int      fd;
double           start, end, time_diff;
int      rank;
int      pes ;
int      filenum;
char     filename[256];
unsigned long offset;
int      iter;

MPI_Comm     comm=MPI_COMM_WORLD;
MPI_Init         (&argc, &argv);
```

```
MPI_Comm_rank       (MPI_COMM_WORLD, &rank);
MPI_Comm_size       (comm, &pes);

        MPI_Barrier (comm);

        buf1=malloc(TEST_SIZE);
        buf2=malloc(TEST_SIZE);

        if (!buf1 || !buf2)
                return -1;
        if ( FPP == TRUE )
                filenum=rank;
        else
                filenum=pes;

  for (iter = 0 ; iter < 4 ; iter++)
  {
        if (rank == 0) printf("Iteration %d\n", iter);
        MPI_Barrier (comm);
        // Rank 0 will create the file first, envolving as many OSTs as PEs, up to a max of the
number of OSTs
        // the other PEs will wait until Rank 0 has defined the file and then they will open it
        sprintf(filename, "output.%d", filenum);
        if (rank == 0)
           { if (iter == 0) {
                fd=llapi_file_open(filename, (unsigned long)STRIPE_COUNT, 0,
MIN(OST_MAX,pes), 1);
                MPI_Barrier(comm); }
                // on all but the first iteration do not create the file,, just open it
                else            {
                fd = open(filename, O_RDWR); }
            }
        else {
                MPI_Barrier(comm);
                fd = open(filename, O_RDWR);
           }

        start = MPI_Wtime();  //  start write test
         offset = lseek64(fd,(long)rank*(long)(TEST_SIZE+FIRST_WRITE),SEEK_SET);
         if(FIRST_WRITE > 0) offset = write(fd, buf1,FIRST_WRITE);
         offset = write(fd, buf1, TEST_SIZE);
```

```
            fsync(fd);
            MPI_Barrier(comm);
           end = MPI_Wtime();    // end write test

      time_diff = end-start;
      if(rank==0) printf("%d pes write speed: %lf MB/sec, %lf seconds\n",
                              pes, (long)pes*(double)TEST_SIZE/time_diff/1024/1024,time_diff );
           MPI_Barrier(comm);

           start = MPI_Wtime();  // start read test
            offset = lseek64(fd,(long)rank*(long)(TEST_SIZE+FIRST_WRITE),SEEK_SET);
            offset = read(fd, buf2, TEST_SIZE);
            fsync(fd);   // fsync here for consistancy,, but as we have only been reading it should
have no effect
            MPI_Barrier(comm);
           end = MPI_Wtime();   // end read test

      time_diff = end-start;
      if(rank==0) printf("%d pes read speed: %lf MB/sec, %lf seconds\n",
                              pes, (long)pes*(double)TEST_SIZE/time_diff/1024/1024 ,time_diff);
           close(fd);

  }
           free(buf1);
           free(buf2);
MPI_Barrier (comm);

           return 0;
}

#define uint64_t unsigned long
#define uint32_t unsigned int
#define uint16_t unsigned short
#define lastost 671
#include <sys/ioctl.h>
#include <fcntl.h>
#define LL_IOC_LOV_SETSTRIPE           _IOW ('f', 154, long)
struct lov_user_ost_data {     /* per-stripe data structure */
      uint64_t l_object_id;       /* OST object ID */
      uint64_t l_object_gr;       /* OST object group (creating MDS number) */
```

```
        uint32_t l_ost_gen;         /* generation of this OST index */
        uint32_t l_ost_idx;         /* OST index in LOV */
} __attribute__((packed));
struct lov_user_md {          /* LOV EA user data (host-endian) */
        uint32_t lmm_magic;         /* magic number = LOV_USER_MAGIC_V1 */
        uint32_t lmm_pattern;       /* LOV_PATTERN_RAID0, LOV_PATTERN_RAID1 */
        uint64_t lmm_object_id;     /* LOV object ID */
        uint64_t lmm_object_gr;     /* LOV object group */
        uint32_t lmm_stripe_size;   /* size of stripe in bytes */
        uint16_t lmm_stripe_count;  /* num stripes in use for this object */
        uint16_t lmm_stripe_offset;  /* starting stripe offset in lmm_objects */
        struct lov_user_ost_data  lmm_objects[0]; /* per-stripe data */
} __attribute__((packed));

#define LOV_USER_MAGIC 0x0BD10BD0
#define O_LOV_DELAY_CREATE 0100000000

int llapi_file_open(const char *name,
      unsigned long stripe_size, int stripe_offset,
                int stripe_count, int stripe_pattern)
{
    struct lov_user_md lum = { 0 };

    int fd, rc = 0;
    int isdir = 0;
    int page_size;

    fd = open(name, O_CREAT | O_LOV_DELAY_CREATE, 0666);

    if (fd <= 0) {
            fprintf(stderr, "error open the file %s \n", name);
            return -1;
    }
/*  consider that stripe_offset is the OST number */
   stripe_offset=stripe_offset%lastost;
/*  Initialize IOCTL striping pattern structure */
    lum.lmm_magic = LOV_USER_MAGIC;
    lum.lmm_pattern = stripe_pattern;
    lum.lmm_stripe_size = stripe_size;
    lum.lmm_stripe_count = stripe_count;
    lum.lmm_stripe_offset = stripe_offset;
```

```
    if (ioctl(fd, LL_IOC_LOV_SETSTRIPE, &lum)) {
            fprintf(stderr, "error setstripe  %s \n", name);
            return -1;
    }

    return fd;

}
```

Galen reach an agreement at the Lustre Scalability Summit with Henry Newman on how to share results. Consolidated results for the runs on both Jaguar XT5 and XT4.

Jaguar XT5										
Read		Offset		NO Offset		Write	Offset		No Offset	
PES	teratior	MB/Sec	Sec	MB/Sec	Sec	PES	MB/Sec	Sec	MB/Sec	Sec
4	0	296	13.8	204	20.1	4	412	9.9	415	9.9
4	1	60	68.8	220	18.6	4	1,948	2.1	1,671	2.5
4	2	808	5.1	562	7.3	4	1,872	2.2	2,000	2.0
4	3	79	52.2	89	46.3	4	56	72.8	402	10.2
32	0	2,972	11.0	1,102	29.7	32	1,226	26.7	1,659	19.8
32	1	7,179	4.6	1,048	31.3	32	9,851	3.3	11,093	3.0
32	2	1,813	18.1	1,730	18.9	32	8,971	3.7	6,317	5.2
32	3	15,902	2.1	13,905	2.4	32	880	37.2	1,051	31.2
128	0	2,921	44.9	4,418	29.7	128	1,262	103.9	1,470	89.2
128	1	28,885	4.5	31,418	4.2	128	10,093	13.0	15,410	8.5
128	2	2,456	53.4	2,805	46.7	128	11,389	11.5	3,883	33.8
128	3	18,847	7.0	31,109	4.2	128	2,513	52.1	5,049	26.0
256	0	4,744	55.3	10,605	24.7	256	2,079	126.1	3,981	65.8
256	1	14,218	18.4	52,464	5.0	256	19,764	13.3	29,572	8.9
256	2	2,126	123.3	3,341	78.5	256	23,122	11.3	17,502	15.0
256	3	42,100	6.2	53,031	4.9	256	3,936	66.6	5,317	49.3
Jaguar XT4										
4	0	1,378	3.0	393	10.4	4	698	5.9	672	6.1
4	1	1,879	2.2	98	41.9	4	1,628	2.5	1,882	2.2
4	2	774	5.3	742	5.5	4	1,886	2.2	1,858	2.2
4	3	185	22.2	696	5.9	4	169	24.3	113	36.3
32	0	814	40.3	720	45.5	32	3,077	10.6	2,997	10.9
32	1	15,253	2.1	14,550	2.3	32	11,786	2.8	9,839	3.3
32	2	3,415	9.6	3,436	9.5	32	12,429	2.6	13,538	2.4
32	3	11,940	2.7	10,964	3.0	32	683	48.0	665	49.3
128	0	1,623	80.8	2,482	52.8	128	6,672	19.6	6,641	19.7
128	1	4,827	27.2	38,004	3.4	128	25,994	5.0	26,993	4.9
128	2	8,577	15.3	8,558	15.3	128	26,147	5.0	29,210	4.5
128	3	3,488	37.6	34,240	3.8	128	2,192	59.8	2,439	53.7
256	0	4,726	55.5	3,827	68.5	256	12,413	21.1	11,537	22.7
256	1	59,519	4.4	6,809	38.5	256	48,509	5.4	48,978	5.4
256	2	15,732	16.7	15,698	16.7	256	53,391	4.9	52,971	4.9
256	3	61,151	4.3	50,639	5.2	256	4,561	57.5	4,807	54.5