Storage Scale Performance Update

Nov. 17 2024, UG SC24 Atlanta (v1.01)

John Lewars (Storage Scale Development Performance Architect)

With Slides and help from **Olaf Weiser** (Storage Scale development)

Additional Slides from :
Yan Xu (Storage Scale Performance Analyst)
Mara Miranda (Storage Scale Performance Analyst)

Disclaimer

Storage Scale

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

IBM reserves the right to change product specifications and offerings at any time without notice. This publication could include technical inaccuracies or typographical errors. References herein to IBM products and services do not imply that IBM intends to make them available in all countries.

Agenda

→io500 Results on SSS 6000 submitted for SC 24

→A Note On Streaming Bandwidth Results on the SSS 6000

→ Improvements to ior-hard-read (fine-grain-read-sharing)

(coming in Storage Scale 5.2.2)

→ Improvements to mmap write (also coming in Storage Scale 5.2.2)

→Various additional performance improvements (if we have time)

Details on IBM’s Recent ESS io500 Submissions

Hardware Details Regarding IBM’s SSS6000 Cluster io500 submission for SC24

- **1x ESS 6000 Building Block, 2 servers/canisters per BB running ESS 6.2.1**
  - Tucson submission uses an 8MB file system block size
  - Ehningen submission uses a 16MB file system block size, NPS-4 (Numa Nodes Per Socket) BIOS Tuning, and PCI Tuning: INT_LOG_MAX_PAYLOAD_SIZE = AUTOMATIC(0) (instead of 4KiB)
  - Eight CX-7 links per server/canister (VPI adapters)
    - Tucson uses HDR200 links
    - Ehningen uses 200 Gpbs RoCE links
  - Both SSS 6000s have:
    - Dual AMD EPYC 9454 48-Core Emb Processors
    - 48x Samsung 1733a NVMe Drives, shared by both canisters in the building block (twin-tailed)

```
#INT_LOG_MAX_PAYLOAD:

for i in `ls /sys/class/infiniband/`; do
   echo y | mlxconfig -d $i -e s
INT_LOG_MAX_PAYLOAD_SIZE=0
done
```

#NPS BIOS Setting is under **"AMD CBS"** (Core and Bus Settings)

- **10x Lenovo AMD clients:**
  - One NDR400-Infiniband connection used per client (Tucson) and two 200 Gbps connections per client (Ehningen)
  - Tucson: 768 GB memory (pagepool=48GB) + Dual EPYC 7302P 16-Core sockets per client (SMT off and only the first socket used by io500 and Storage Scale)
  - Ehningen: 192 GB memory (pagepool=32GB) + Single EPYC 9274F 24-Core socket per client (SMT on)

Details on IBM’s Recent ESS io500 Submissions

Configuration Details Regarding IBM’s SSS6000 Cluster io500 submission for SC24:

- IBM Tucson: Manager node role assigned to all 10 clients and both SSS 6000 storage servers

- IBM Ehningen: Manager node role assigned to all 16 clients and four servers (only 10 clients and the two SSS 6000 servers described above are directly running io500)
- In the Ehningen run there are two additional clients in the cluster that are not assigned as managers.

MPI-Level Changes only for Ehningen Run:

- **mpi-args:** `--bind-to numa:1`

（Binding to the same socket as the single CX-7 adapter used for the io500 run)

Benchmark Configuration

- An external pfind script was used, which is included in our io500 submission and is available from IBM (contact jlewars@us.ibm.com)

Details on IBM’s Recent ESS io500 Submissions (cont.)

RDMA enabled on all clients and servers (verbsRdmaSend + verbsRdma)

Servers’ Relevant mmchconfig Tuning:

- dataShipServerBufferPct 50
- preferDesignatedMnode yes
- maxMBpS 50000

Clients’ Relevant Non-default mmchconfig Tuning (many options set by the gssClientConfig.sh script, which should be run first):

- maxStatCache 131072 #over-rides gssClientConfig.sh
- nBucketGroups 1024
- preferDesignatedMnode yes
- fsyncIsGlobal no
- dataShipClientBuffersPerServer 50
- dataShipClientBufferPct 50
- autoCompactDir 0
- fgdlTokenBatchAcquire 1

Only for the find benchmark (set on-demand by a helper script, which is available from jlewars@us.ibm.com on request):

- inodePrefetchThreshold 2000000

Details on IBM’s Recent ESS io500 Submissions (cont.)

Client Tuning Specific to the Tucson and Ehningen Submissions

Ehningen-specific mmchconfig Tuning:

- /usr/lpp/mmfs/samples/gss/gssClientConfig.sh -P 32768 # starting point performance tuning w/ 32G pagepool
- verbsRdmaCm enable # required for RoCE
- maxMBpS 50000 # (but maxMBpS tuned via helper script to 2048 for just the ior-hard-read benchmark)

Tucson-specific mmchconfig Tuning:

- /usr/lpp/mmfs/samples/gss/gssClientConfig.sh -P 49152 # starting point performance tuning w/ 48G pagepool
- maxMBpS 24000
- numactlOptioni 0   <= Binding mmfsd to same socket (0) as the CX-7 network adapter used for io500
- numactlOptionN 0   <= Binding mmfsd to same socket (0) as the CX-7 network adapter used for io500
- verbsPorts mlx5_0/1 <= RDMA enabled for only the CX-7 network adapter in first NUMA domain

IO500 submissions from IBM for SC24 (both with pre-GA IBM Storage Scale 5.2.2.0)

Storage Scale

Benchmark Storage Scale 5.1.9.0	IBM Tucson (Dual AMD EPYC 9254 24-Core Processor, 8MB Blocksize, Infiniband) – SSS6000 1 BB – SC24	IBM Ehningen (Single EPYC 9274F 24-Core 10 clients, 16 MB Blocksize, RoCE) – SSS6000 1 BB – SC24	Percent Difference
ior-easy-write	151.1	148.0	-2.1
mdtest-easy-write	191.2	322.9	68.9
ior-hard-write	28.3	44.2	56.2
mdtest-hard-write	24.4	27.6	13.1
find	7516.7	12203.1	62.3
ior-easy-read	284.2	310.4	9.2
mdtest-easy-stat	571.3	961.9	68.4
ior-hard-read	107.5	130.2	21.1
mdtest-hard-stat	272.7	327.2	20.0
mdtest-easy-delete	165.3	133.7	-19.1
mdtest-hard-read	321.6	305.7	-4.9
mdtest-hard-delete	29.4	36.1	22.8
Bandwidth GiB/s	106.9	127.5	19.3
kIOPS	232.5	290.3	24.9
TOTAL Score	157.7	192.4	22.0

- In the Ehningen run, to get better ior-easy bandwidth using only 20 tasks per node (optimal for the performance of the overall suite), three changes were made, compared to the settings used for io500 on the IBM Tucson system:
  - maxMBpS=50000 (mmchconfig)
  - NPS=4 (set in BIOS)
  - INT_LOG_MAX_PAYLOAD_SIZE = AUTOMATIC(0) (PCI tuning)
- Regarding the latest software improvements, the main change improves the fine-grain-read-sharing hint used for the ior-hard-read benchmark

Example Bandwidth Tests On Ehningen SSS 6000 System

Storage Scale

With the settings described on the previous charts on the Ehningen SSS 600 system, we ran ior-easy-write and ior-easy-read 10 times in a row with TRIM disabled, unmounting the file system in between each run:

With this test, ior-easy-read may benefit from data being in the SSS 6000's memory cache (GNR cache):

```
runior.10.32.write.1.out:Write MB/s = 164905
runior.10.32.write.2.out:Write MB/s = 163049
runior.10.32.write.3.out:Write MB/s = 158445
runior.10.32.write.4.out:Write MB/s = 110026
runior.10.32.write.5.out:Write MB/s = 109780
runior.10.32.write.6.out:Write MB/s = 110550
runior.10.32.write.7.out:Write MB/s = 97791
runior.10.32.write.8.out:Write MB/s = 104367
runior.10.32.write.9.out:Write MB/s = 94346
runior.10.32.write.10.out:Write MB/s = 105825
```

```
runior.10.32.read.1.out:Read MB/s = 334813
runior.10.32.read.2.out:Read MB/s = 339096
runior.10.32.read.3.out:Read MB/s = 333998
runior.10.32.read.4.out:Read MB/s = 331219
runior.10.32.read.5.out:Read MB/s = 334249
runior.10.32.read.6.out:Read MB/s = 337686
runior.10.32.read.7.out:Read MB/s = 338150
runior.10.32.read.8.out:Read MB/s = 338448
runior.10.32.read.9.out:Read MB/s = 337775
runior.10.32.read.10.out:Read MB/s = 335021
```

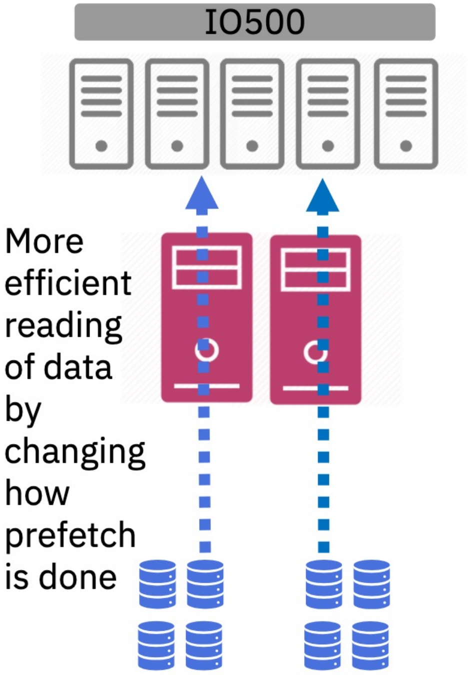
So, the SSS 6000 with the Samsung 1733a drives shipping now is capable of more than the 155 GB/s write performance that has been referenced in other slides but note trim should be enabled to address potential write degradation over time (the mmreclaimspace command can be run to force trim discards to be issued).

The ior-read-hard Benchmark Challenges and Action Plan

- Originally presented in an Storage Scale Expert talk in July 2022

- **What makes ior hard read so hard?**
  - Shared file reads currently causes aggressive prefetching leading to client nodes reading data that will not be consumed.
  - Small read request size limits efficiencies of network transfers if prefetch isn't done correctly.
  - The benchmark ensure that all tasks read data written by another task, which means there are token considerations (can be addressed by MPI hint to release tokens).
  - Like ior hard write, a 47008 byte write request size is used, which prevents Direct IO from being used.

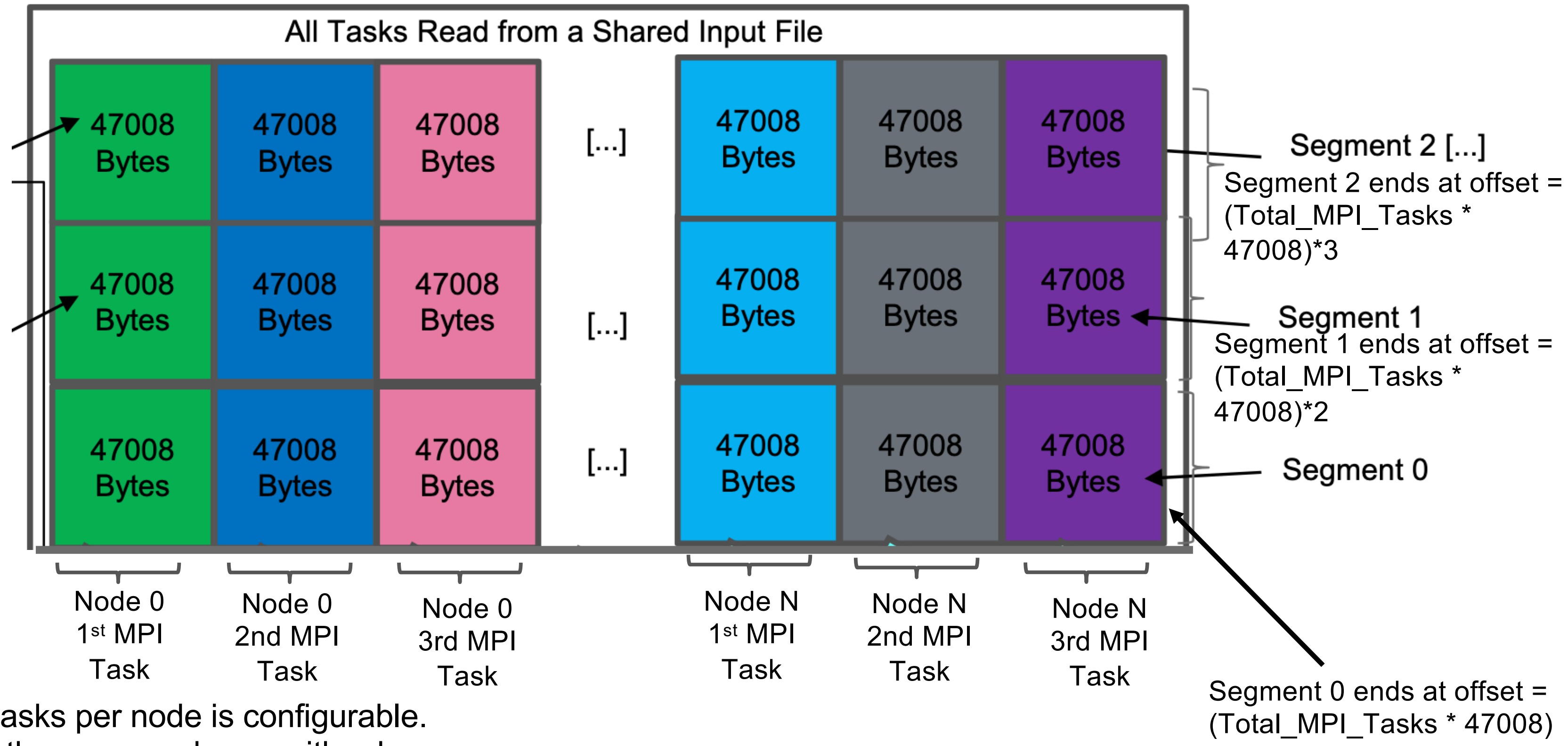
- **Action plan:**
  - Multiple design changes are being made to prefetch with optimizations controlled via fcntl hint. A fineGrainReadSharing hint has been added to IOR via this commit: https://github.com/hpc/ior/issues/390


IO500
More efficient reading of data by changing how prefetch is done

Access Pattern for ior-hard-read

The ior-hard-read benchmark shifts the mapping of task IDs across nodes to avoid nodes from reading cached data they may have previously written.

47008 bytes written by the first MPI task running on node N. With the exception of the first node (which reads data written by Node N) all nodes read the data written by the previous node in the hostlist (so Node 1 reads data written by Node 0, Node 2 reads data written by node 1, etc.)



All Tasks Read from a Shared Input File

Node 0 1st MPI Task	Node 0 2nd MPI Task	Node 0 3rd MPI Task		Node N 1st MPI Task	Node N 2nd MPI Task	Node N 3rd MPI Task	
47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes	Segment 2 [...]
47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes	Segment 1
47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes	Segment 0

Segment 2 ends at offset = (Total_MPI_Tasks * 47008)*3

Segment 1 ends at offset = (Total_MPI_Tasks * 47008)*2

Segment 0 ends at offset = (Total_MPI_Tasks * 47008)

Number of tasks per node is configurable.
All tasks on the same node can either be:

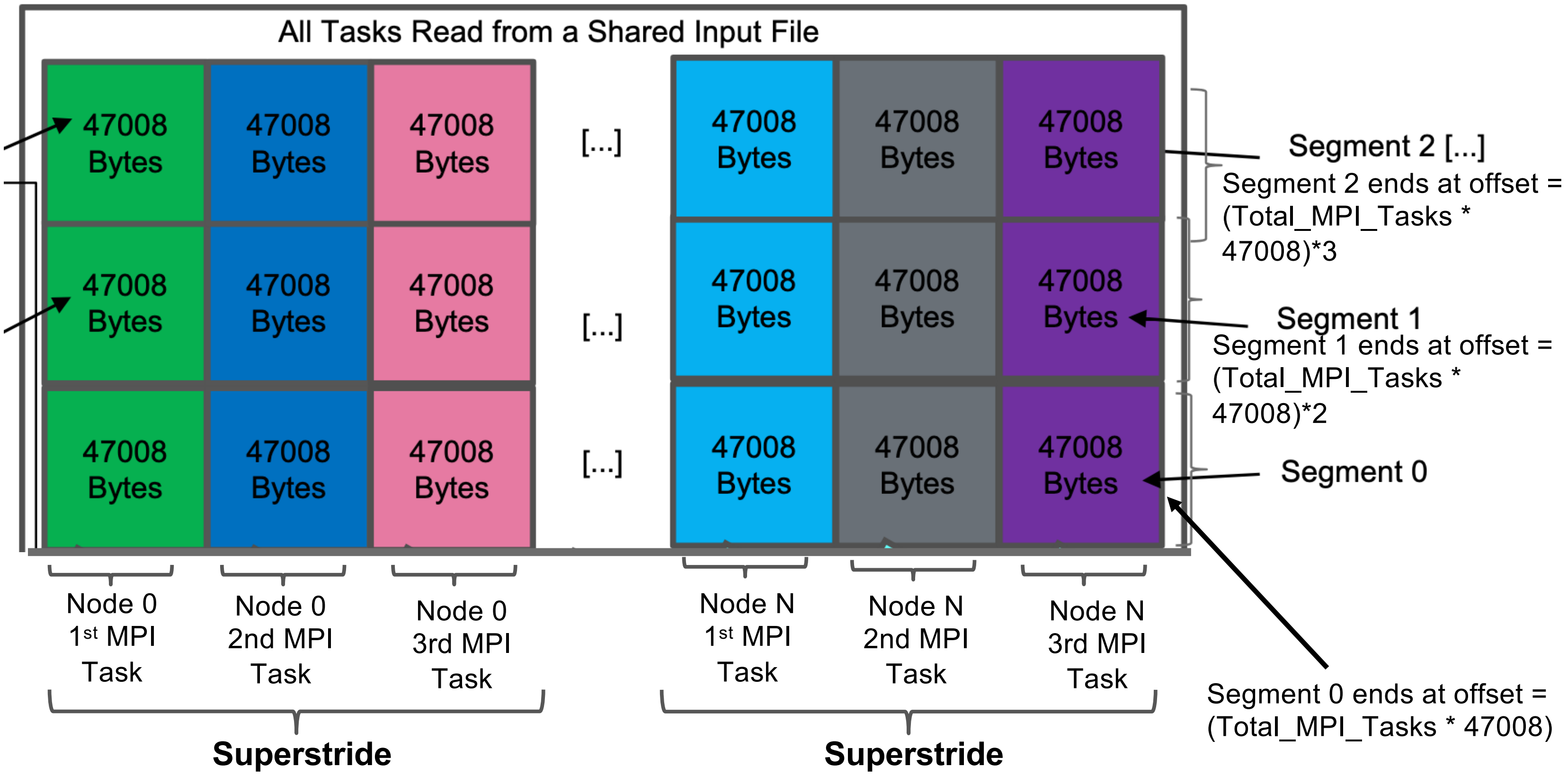
1. Contiguous (in terms of MPI task IDs) as depicted in this slide (which gives the opportunity to coalesce prefetches) **or**
2. Round-robin’ed (e.g., with 4 tasks per node and 8 nodes, the first node has tasks 0, 8, 16, and 24, etc.)

ior-hard-read and Design Changes to Prefetch

- In Storage Scale 5.2.2, IBM intends to make prefetch aware of the interaction between multiple instances/threads on the same node doing strided reads (fine-grain-read-sharing)

The ior-hard-read benchmark shifts the mapping of task IDs across nodes to avoid nodes from reading cached data they may have previously written.

47008 bytes written by the first MPI task running on node N. With the exception of the first node (which reads data written by Node N) all nodes read the data written by the previous node in the hostlist (so Node 1 reads data written by Node 0, Node 2 reads data written by node 1, etc.)



All Tasks Read from a Shared Input File

47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes
47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes
47008 Bytes	47008 Bytes	47008 Bytes	[...]	47008 Bytes	47008 Bytes	47008 Bytes
Node 0 1st MPI Task	Node 0 2nd MPI Task	Node 0 3rd MPI Task		Node N 1st MPI Task	Node N 2nd MPI Task	Node N 3rd MPI Task

Superstride **Superstride**

Segment 2 [...]
Segment 2 ends at offset = (Total_MPI_Tasks * 47008)*3

Segment 1
Segment 1 ends at offset = (Total_MPI_Tasks * 47008)*2

Segment 0

Segment 0 ends at offset = (Total_MPI_Tasks * 47008)

Superstride prefetching coalesce prefetches across instances doing strided reads, improving efficiency.

MPI task/job mapping When Using Open-MPI

Specifying Host Nodes

https://www.open-mpi.org/doc/v4.0/man1/mpirun.1.php

```
mpirun -H aa,aa,bb ./a.out
```

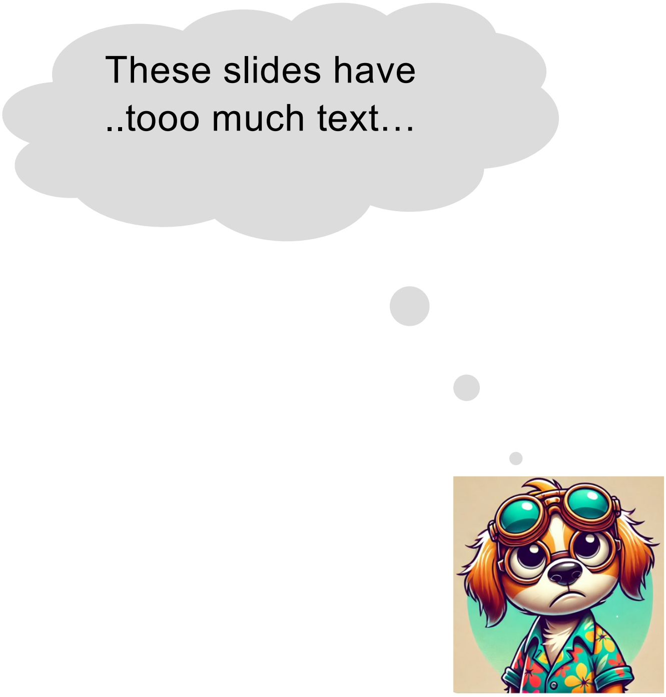
launches two processes on node aa and one on bb.

```
% cat myhostfile
aa slots=2
bb slots=2
cc slots=2
```

Here, we list both the host names (aa, bb, and cc) but also how many slots there are for each.

```
mpirun -hostfile myhostfile ./a.out
```

will launch two processes on each of the three nodes.



Host nodes can be identified on the mpirun command line with the -host option or in a hostfile.
For example,

```
mpirun -H aa,aa,bb ./a.out
```

launches two processes on node aa and one on bb.
Or, consider the hostfile

```
% cat myhostfile
aa slots=2
bb slots=2
cc slots=2
```

Here, we list both the host names (aa, bb, and cc) but also how many slots there are for each.

```
mpirun -hostfile myhostfile ./a.out
```

will launch two processes on each of the three nodes.

```
mpirun -hostfile myhostfile -host aa ./a.out
```

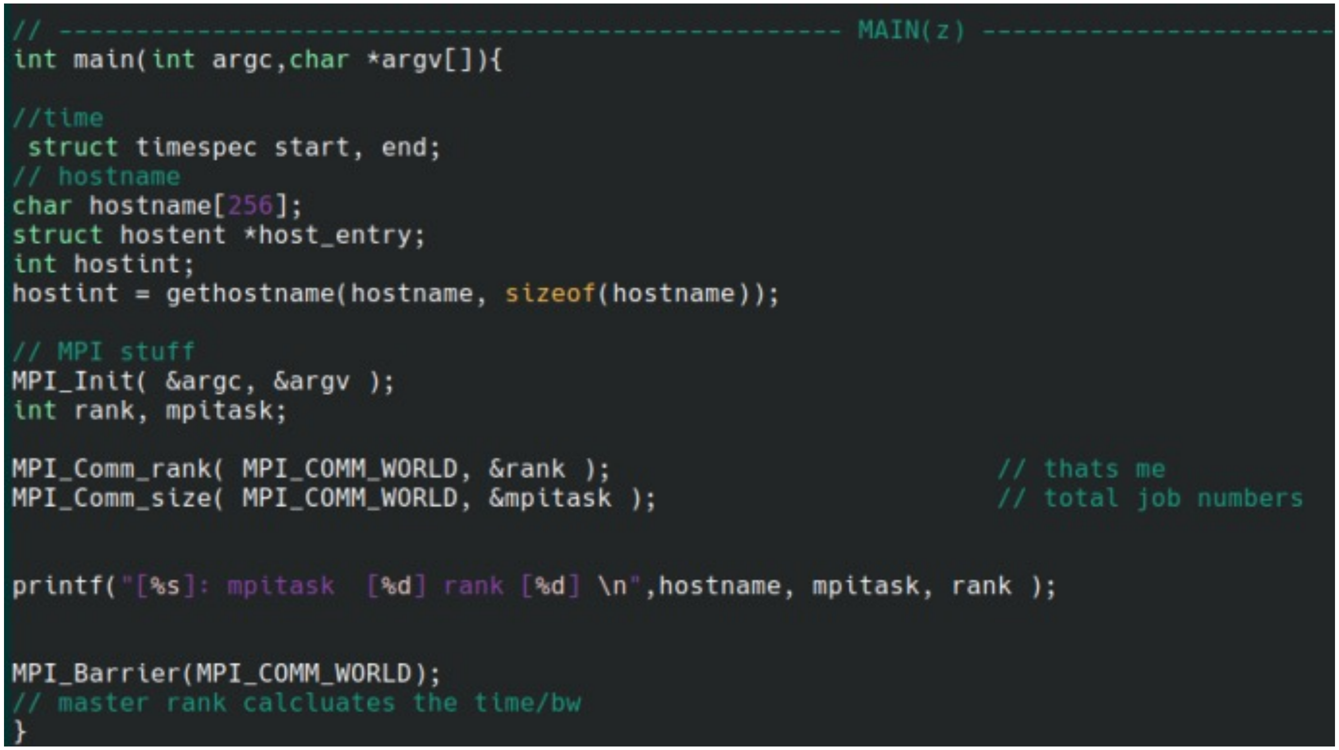
will launch two processes, both on node aa.

```
mpirun -hostfile myhostfile -host dd ./a.out
```

will find no hosts to run on and abort with an error. That is, the specified host dd is not in the specified hostfile.
When running under resource managers (e.g., SLURM, Torque, etc.), Open MPI will obtain both the hostnames and the number of slots directly from the resource manger.

MPI task/job mapping cont.

- We can use a simple program to show how mpi allocate task IDs:



```
// --------------------------------------------------- MAIN(z) ----------------------
int main(int argc,char *argv[]){

//time
 struct timespec start, end;
// hostname
char hostname[256];
struct hostent *host_entry;
int hostint;
hostint = gethostname(hostname, sizeof(hostname));

// MPI stuff
MPI_Init( &argc, &argv );
int rank, mpitask;

MPI_Comm_rank( MPI_COMM_WORLD, &rank );                  // thats me
MPI_Comm_size( MPI_COMM_WORLD, &mpitask );               // total job numbers


printf("[%s]: mpitask  [%d] rank [%d] \n",hostname, mpitask, rank );


MPI_Barrier(MPI_COMM_WORLD);
// master rank calcluates the time/bw
}
```

MPI task/job mapping cont.

mpirun -H aa,bb -np 8 ./a.out
launches 8 processes. Since only two hosts are specified, after the first two processes are mapped,
 one to aa and one to bb,
the remaining processes oversubscribe the specified hosts.

```
[root@fscc-sr665-8 ~]# mpiexec -n 10 -hostfile hostlist /gpfs/bb1nvme/a.out | sort -nk 1
[fscc-sr665-11]: mpitask  [10] rank [0]
[fscc-sr665-11]: mpitask  [10] rank [5]
[fscc-sr665-12]: mpitask  [10] rank [1]
[fscc-sr665-12]: mpitask  [10] rank [6]
[fscc-sr665-13]: mpitask  [10] rank [2]
[fscc-sr665-13]: mpitask  [10] rank [7]
[fscc-sr665-14]: mpitask  [10] rank [3]
[fscc-sr665-14]: mpitask  [10] rank [8]
[fscc-sr665-15]: mpitask  [10] rank [4]
[fscc-sr665-15]: mpitask  [10] rank [9]
```

MPI task/job mapping cont.

mpirun -H aa,bb -np 8 ./a.out
launches 8 processes. Since only two hosts are specified, after the first two processes are mapped,
  one to aa and one to bb,
the remaining processes oversubscribe the specified hosts.

```
[root@fscc-sr665-8 ~]# mpiexec -n 10 -hostfile hostlist /gpfs/bb1nvme/a.out | sort -nk 1
[fscc-sr665-11]: mpitask  [10] rank [0]
[fscc-sr665-11]: mpitask  [10] rank [5]
[fscc-sr665-12]: mpitask  [10] rank [1]
[fscc-sr665-12]: mpitask  [10] rank [6]
[fscc-sr665-13]: mpitask  [10] rank [2]
[fscc-sr665-13]: mpitask  [10] rank [7]
[fscc-sr665-14]: mpitask  [10] rank [3]
[fscc-sr665-14]: mpitask  [10] rank [8]
[fscc-sr665-15]: mpitask  [10] rank [4]
[fscc-sr665-15]: mpitask  [10] rank [9]
```

```
[root@fscc-sr665-8 ~]# mpiexec -n 10 -ppn 2 -hostfile hostlist /gpfs/bb1nvme/a.out | sort -nk 1
[fscc-sr665-11]: mpitask  [10] rank [0]
[fscc-sr665-11]: mpitask  [10] rank [1]
[fscc-sr665-12]: mpitask  [10] rank [2]
[fscc-sr665-12]: mpitask  [10] rank [3]
[fscc-sr665-13]: mpitask  [10] rank [4]
[fscc-sr665-13]: mpitask  [10] rank [5]
[fscc-sr665-14]: mpitask  [10] rank [6]
[fscc-sr665-14]: mpitask  [10] rank [7]
[fscc-sr665-15]: mpitask  [10] rank [8]
[fscc-sr665-15]: mpitask  [10] rank [9]
[root@fscc-sr665-8 ~]#
```

Storage Scale IOR task/job mapping w/mpich –bringing these two pattern together

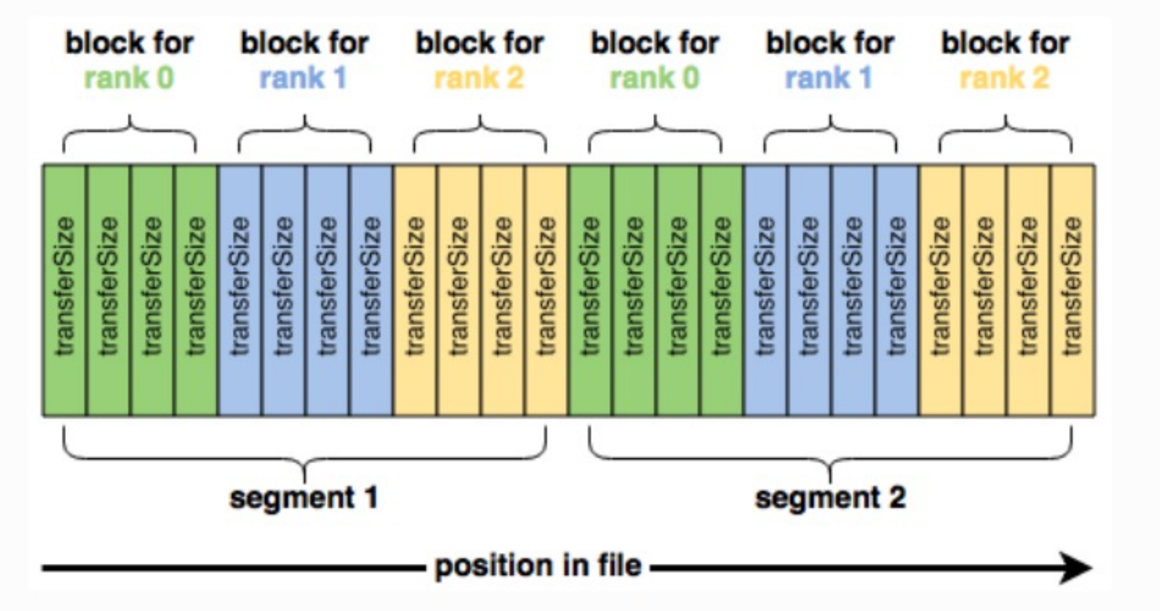
Storage Scale

default mapping

```
[fscc-sr665-11]: mpitask  [10] rank [0]
[fscc-sr665-11]: mpitask  [10] rank [5]
[fscc-sr665-12]: mpitask  [10] rank [1]
[fscc-sr665-12]: mpitask  [10] rank [6]
[fscc-sr665-13]: mpitask  [10] rank [2]
[fscc-sr665-13]: mpitask  [10] rank [7]
[fscc-sr665-14]: mpitask  [10] rank [3]
[fscc-sr665-14]: mpitask  [10] rank [8]
[fscc-sr665-15]: mpitask  [10] rank [4]
[fscc-sr665-15]: mpitask  [10] rank [9]
```

-ppn mapping

```
[fscc-sr665-11]: mpitask  [10] rank [0]
[fscc-sr665-11]: mpitask  [10] rank [1]
[fscc-sr665-12]: mpitask  [10] rank [2]
[fscc-sr665-12]: mpitask  [10] rank [3]
[fscc-sr665-13]: mpitask  [10] rank [4]
[fscc-sr665-13]: mpitask  [10] rank [5]
[fscc-sr665-14]: mpitask  [10] rank [6]
[fscc-sr665-14]: mpitask  [10] rank [7]
[fscc-sr665-15]: mpitask  [10] rank [8]
[fscc-sr665-15]: mpitask  [10] rank [9]
```

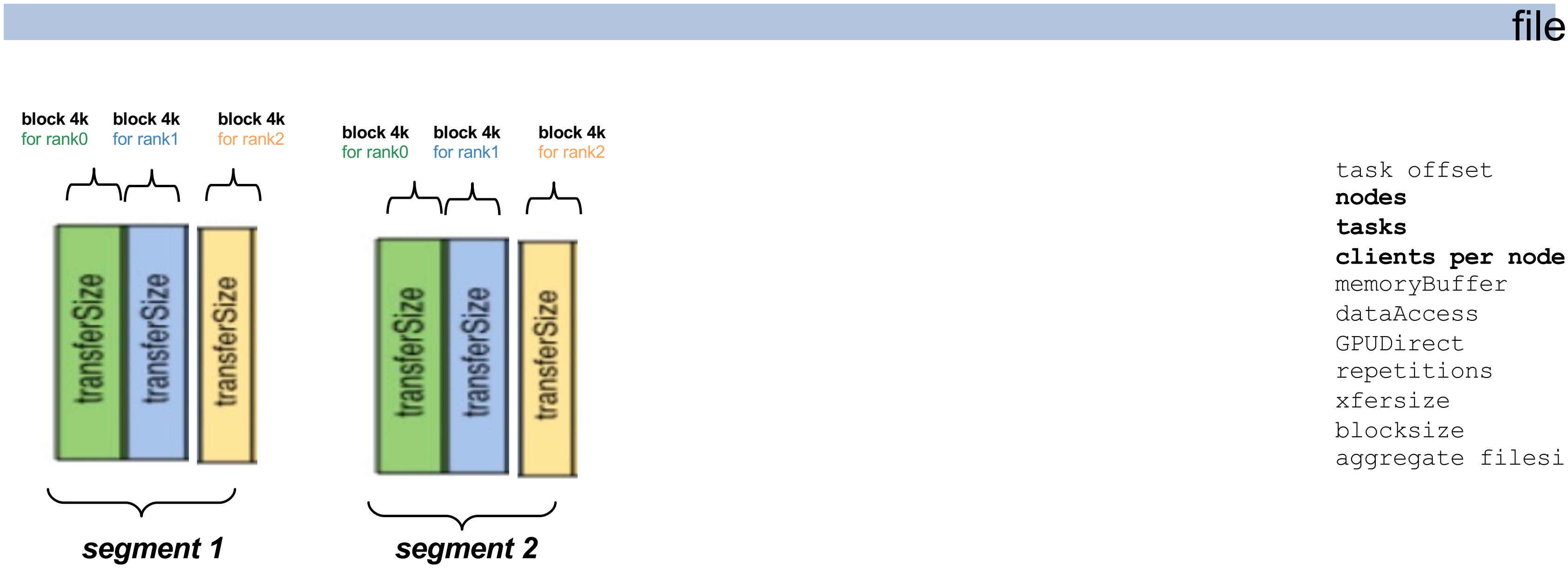


IOR task for FZJulich‘s version of ior-hard-read (-t 4k -b 4k . . .) – IO pattern zoomed

Storage Scale

```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```

one shared file



```
task offset          : 1
nodes                : 16
tasks                : 320
clients per node     : 20
memoryBuffer         : CPU
dataAccess           : CPU
GPUDirect            : 0
repetitions          : 1
xfersize             : 4096 bytes
blocksize            : 4096 bytes
aggregate filesize   : 11.92 TiB
```

SegmentSize= total task * blocksize

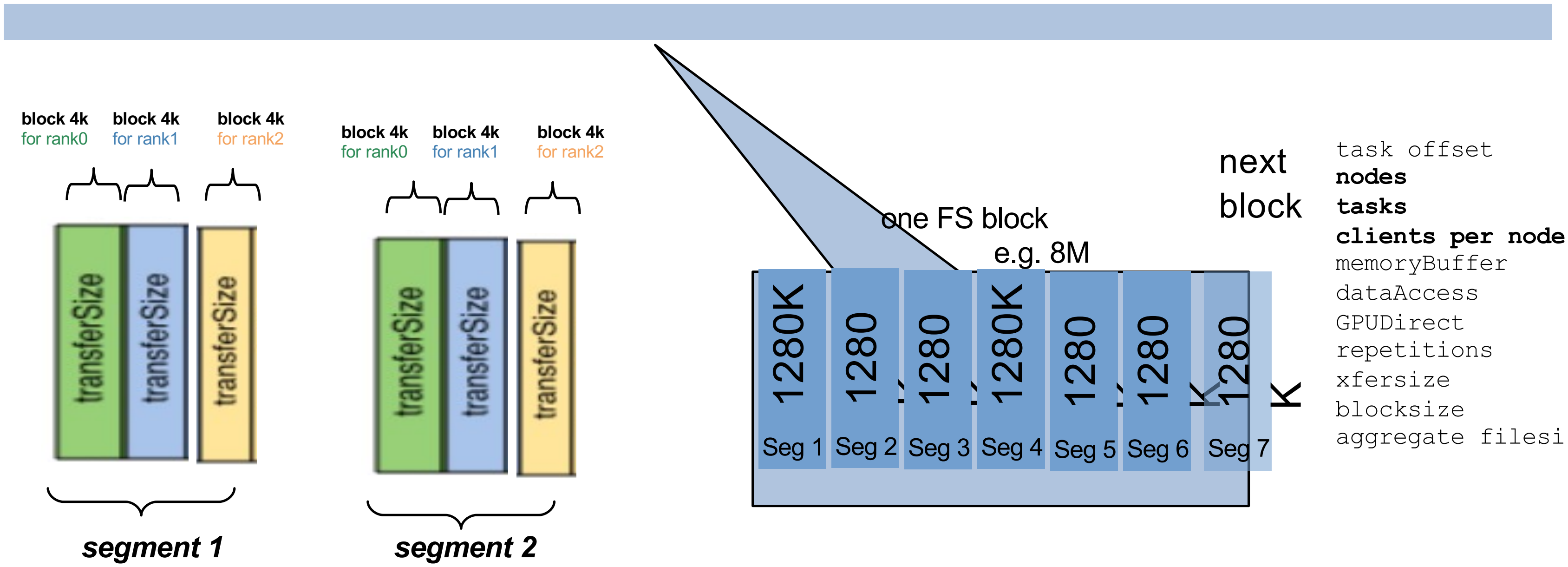
segSize= 320 * 4k = 1280 K

1280K * 10000000 = 11.92 TB

GPFS IOR task for FZJulich -t 4k -b 4k – IO pattern zoomed

```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```

one
shared file



```
task offset           : 1
nodes                 : 16
tasks                 : 320
clients per node      : 20
memoryBuffer          : CPU
dataAccess            : CPU
GPUDirect             : 0
repetitions           : 1
xfersize              : 4096 bytes
blocksize             : 4096 bytes
aggregate filesize    : 11.92 TiB
```

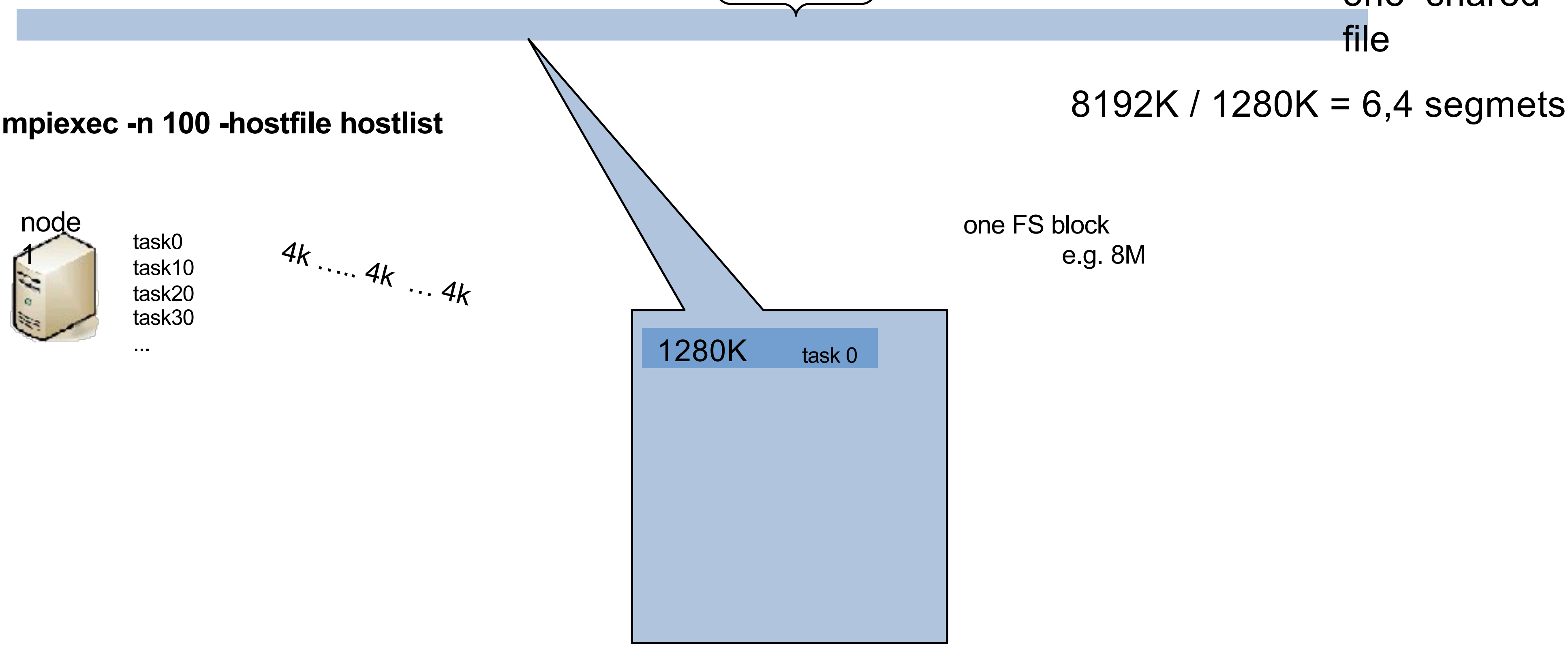
SegmentSize= total task * blocksize

segSize= 320 * 4k = 1280 K

1280K * 10000000 = 11.92 TB

watch the difference – default distribution or -ppn

```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```

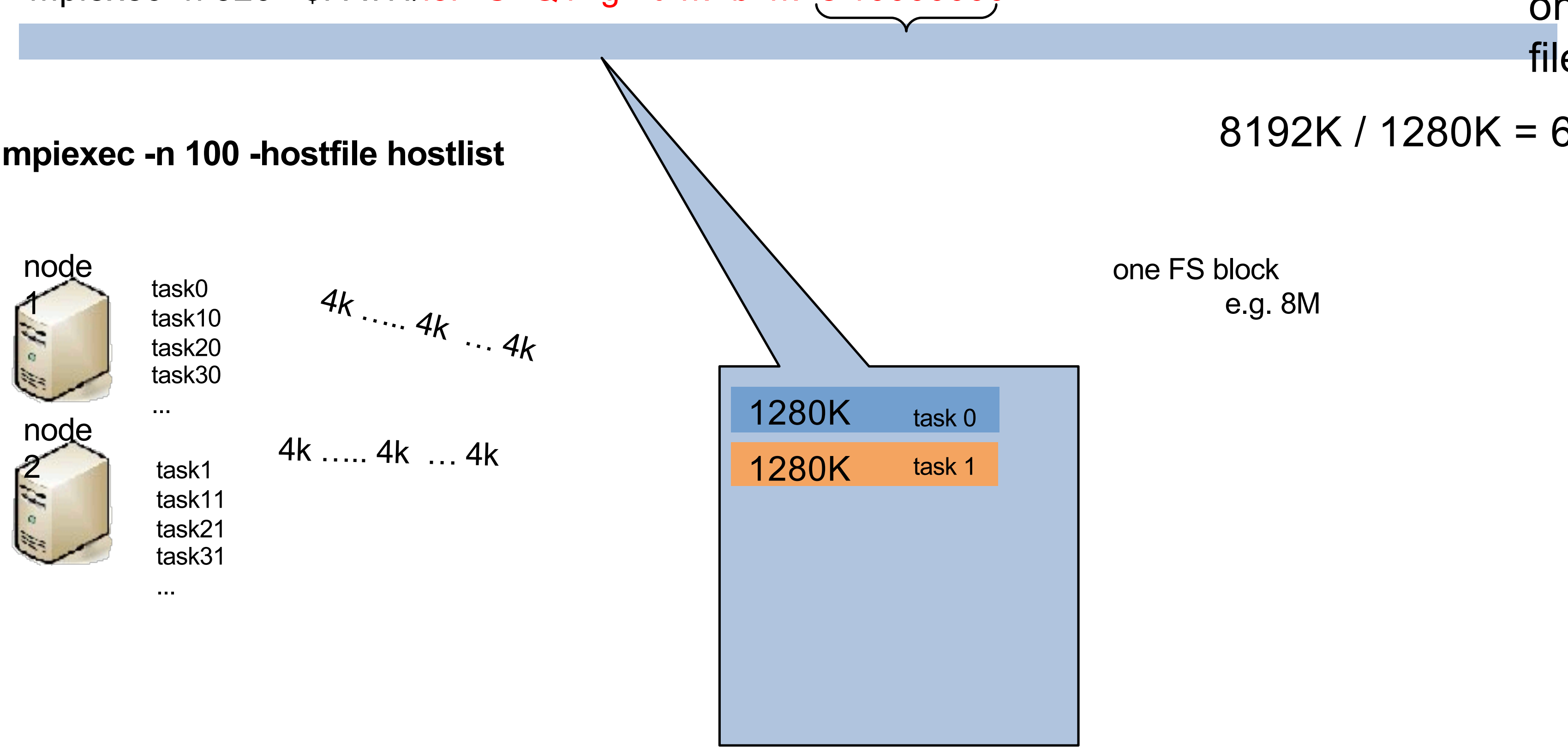


mpiexec -n 100 -hostfile hostlist

8192K / 1280K = 6,4 segmets

watch the difference – default distribution or -ppn

```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```



one shared file

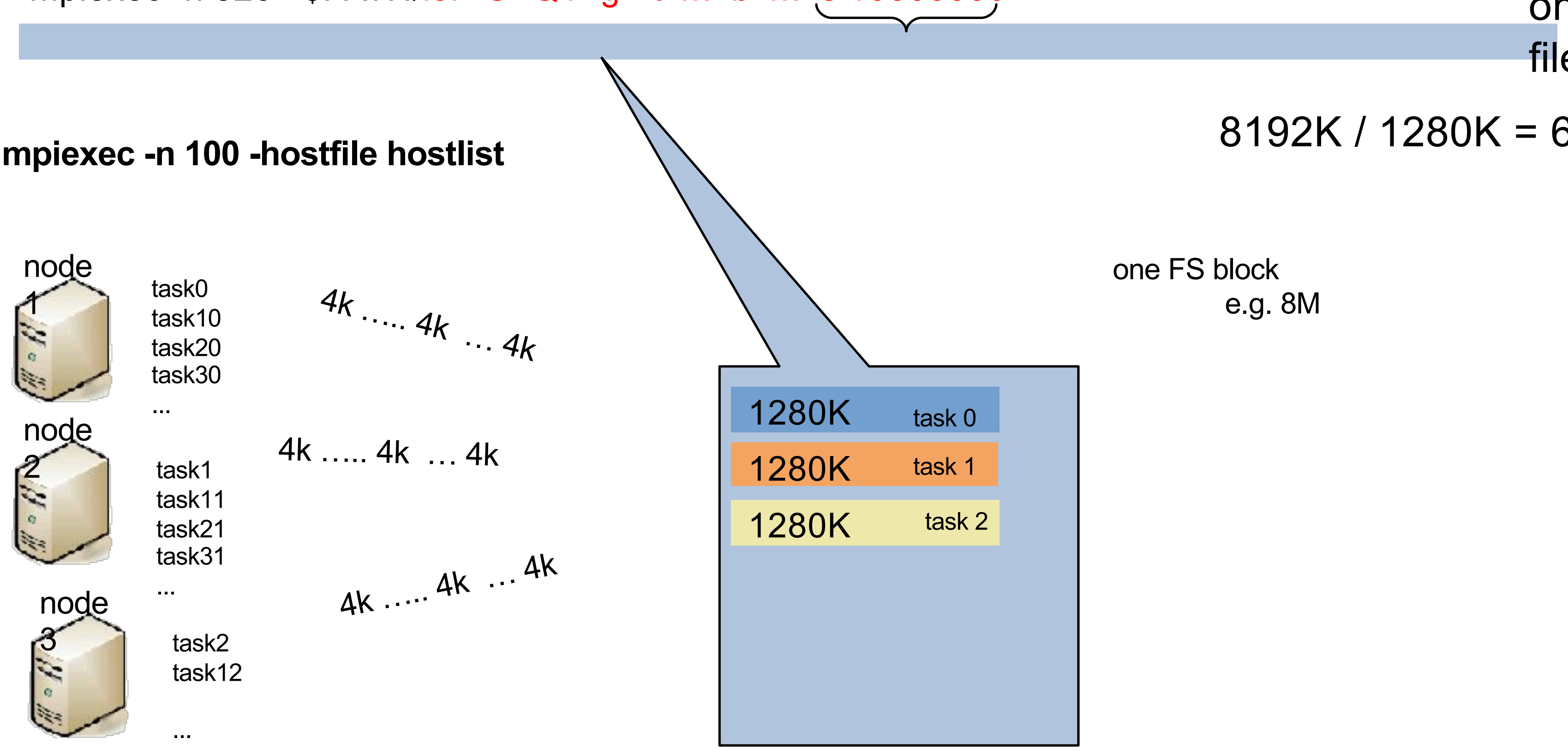
8192K / 1280K = 6,4 segmets

mpiexec -n 100 -hostfile hostlist

one FS block e.g. 8M

watch the difference – default distribution or -ppn

```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```



one shared file

8192K / 1280K = 6,4 segmets

mpiexec -n 100 -hostfile hostlist

one FS block
e.g. 8M

1280K task 0

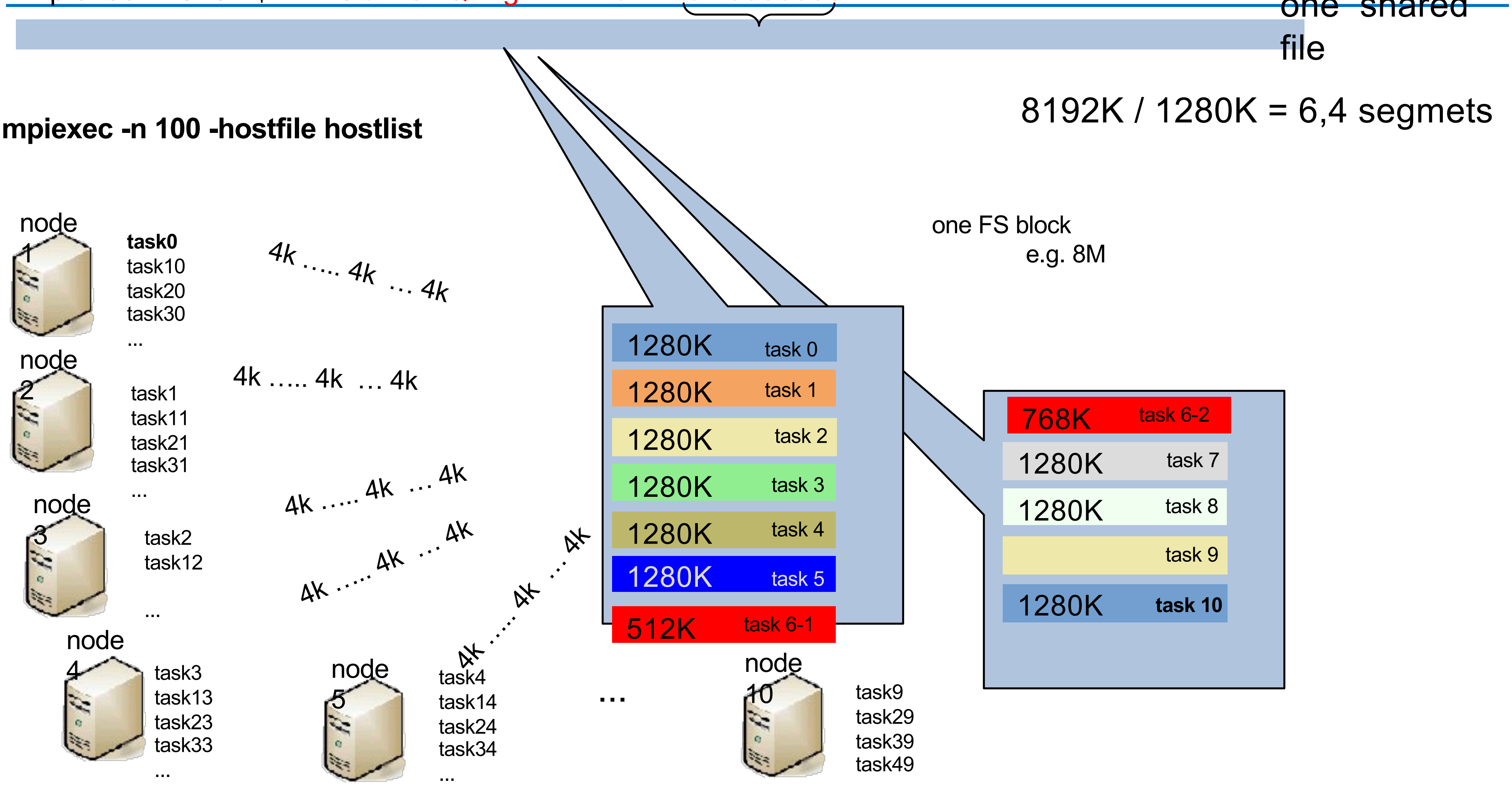
1280K task 1

1280K task 2

watch the difference – default distribution or -ppn

Storage Scale

mpiexec -n 320 $PATH/ior -C -Q1 -g -t 4k -b 4k -s 10000000



one shared file

8192K / 1280K = 6,4 segmets

mpiexec -n 100 -hostfile hostlist

one FS block
e.g. 8M

node 1
task0
task10
task20
task30
...

node 2
task1
task11
task21
task31
...

node 3
task2
task12
...

node 4
task3
task13
task23
task33
...

node 5
task4
task14
task24
task34
...

...

node 10
task9
task29
task39
task49

4k ..... 4k ... 4k

4k ..... 4k ... 4k

4k ..... 4k ... 4k

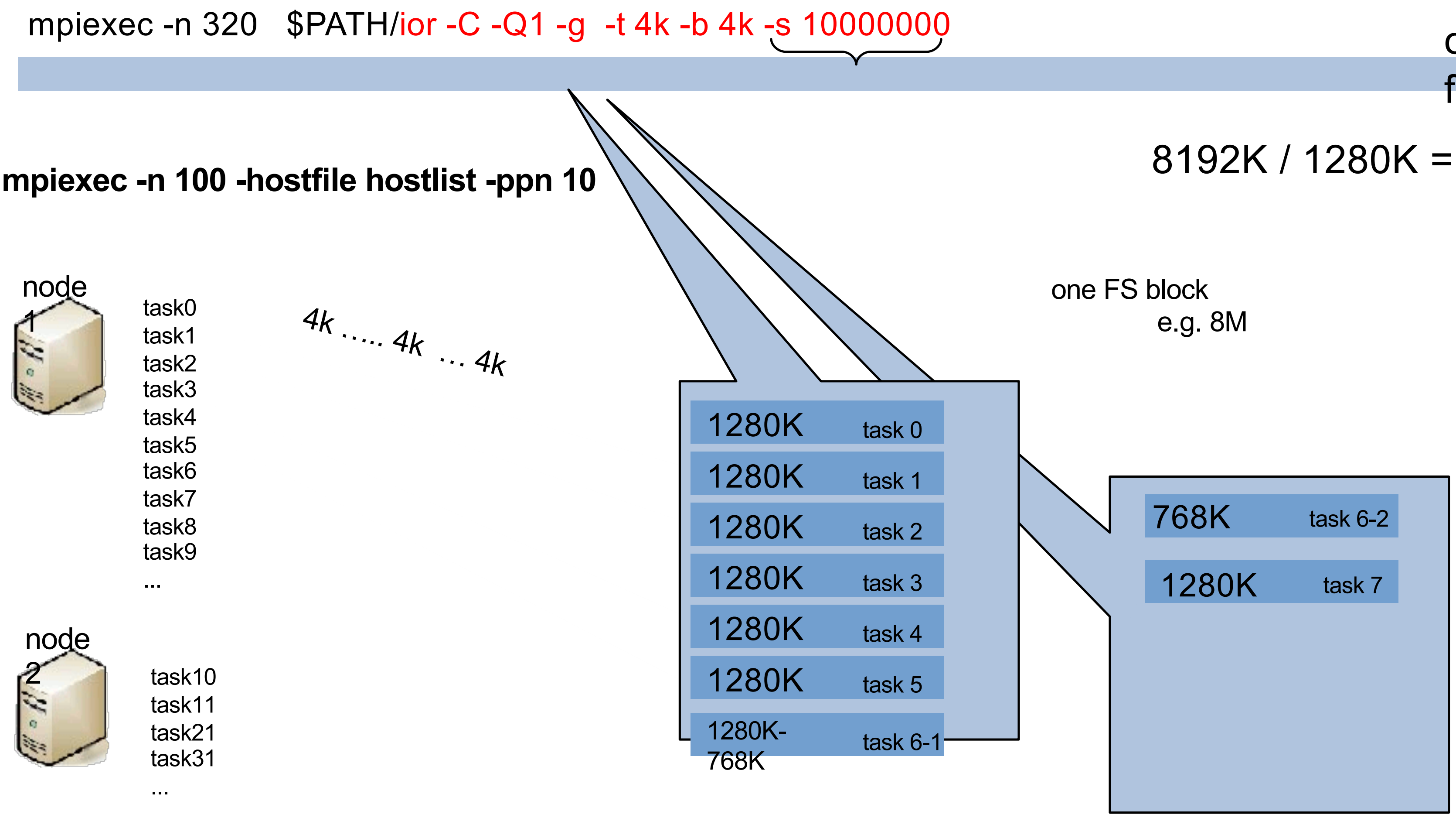
4k ..... 4k ... 4k

4k ..... 4k ... 4k

1280K task 0
1280K task 1
1280K task 2
1280K task 3
1280K task 4
1280K task 5
512K task 6-1

768K task 6-2
1280K task 7
1280K task 8
task 9
1280K **task 10**

watch the difference – default distribution or -ppn



```
mpiexec -n 320   $PATH/ior -C -Q1 -g  -t 4k -b 4k -s 10000000
```

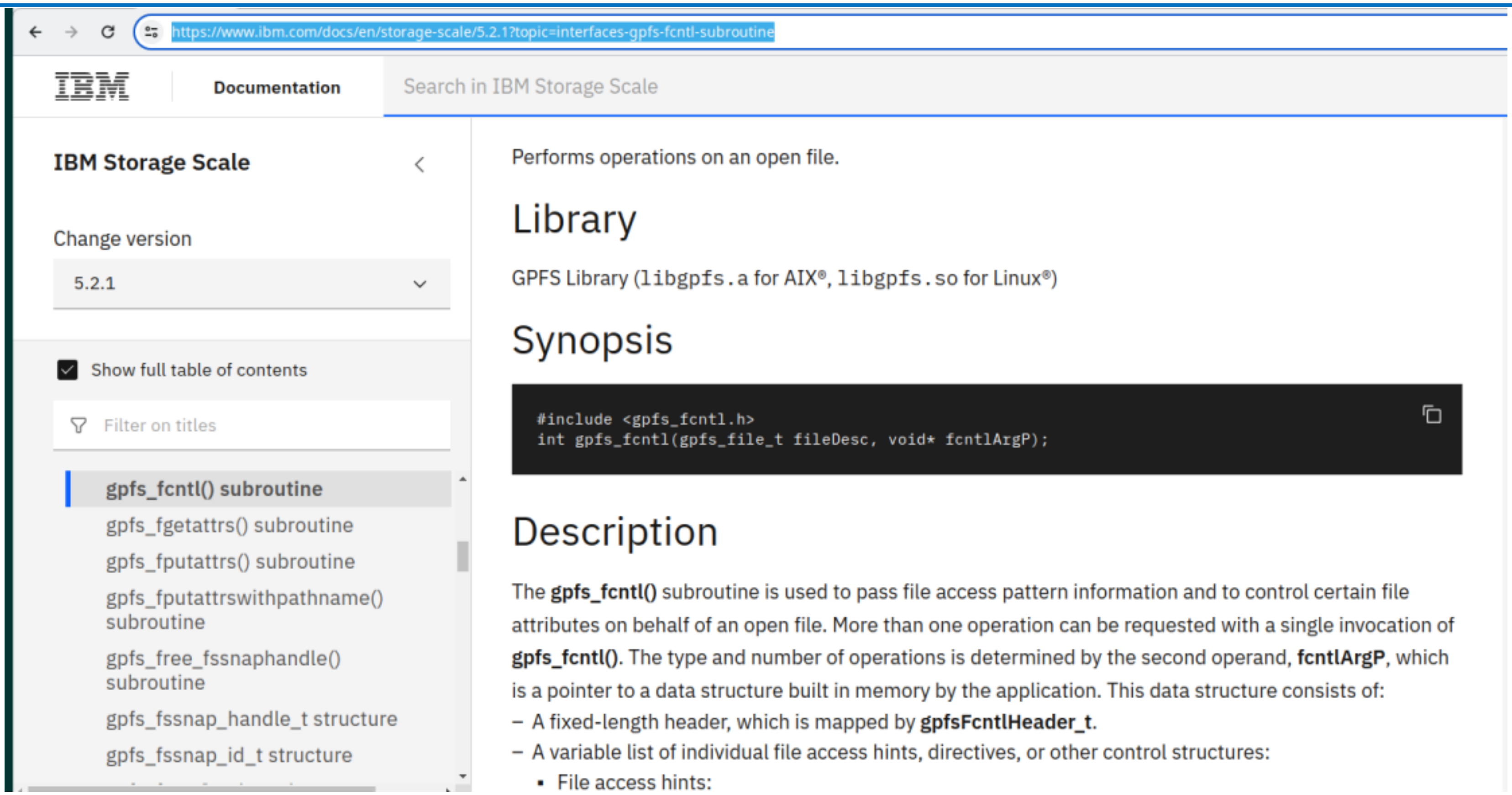
one shared file

8192K / 1280K = 6,4 segmets

```
mpiexec -n 100 -hostfile hostlist -ppn 10
```

Code enhancement for strided workloads - how to use it

Storage Scale


https://www.ibm.com/docs/en/storage-scale/5.2.1?topic=interfaces-gpfs-fcntl-subroutine
IBM
Documentation
Search in IBM Storage Scale
IBM Storage Scale
Filter on titles
gpfs_fcntl() subroutine
gpfs_fgetattrs() subroutine
gpfs_fputattrs() subroutine
gpfs_fputattrswithpathname() subroutine
gpfs_free_fssnaphandle() subroutine
gpfs_fssnap_handle_t structure
gpfs_fssnap_id_t structure

Change version: 5.2.1

☑ Show full table of contents

Performs operations on an open file.

Library

GPFS Library (`libgpfs.a` for AIX®, `libgpfs.so` for Linux®)

Synopsis

```
#include <gpfs_fcntl.h>
int gpfs_fcntl(gpfs_file_t fileDesc, void* fcntlArgP);
```

Description

The **gpfs_fcntl()** subroutine is used to pass file access pattern information and to control certain file attributes on behalf of an open file. More than one operation can be requested with a single invocation of **gpfs_fcntl()**. The type and number of operations is determined by the second operand, **fcntlArgP**, which is a pointer to a data structure built in memory by the application. This data structure consists of:

- A fixed-length header, which is mapped by **gpfsFcntlHeader_t**.
- A variable list of individual file access hints, directives, or other control structures:
  - File access hints:

https://www.ibm.com/docs/en/storage-scale/5.2.1?topic=interfaces-gpfs-fcntl-subroutine

Starting Point for ior-hard-read, Bottlenecks Seen Before Adding fine-grain-read-sharing

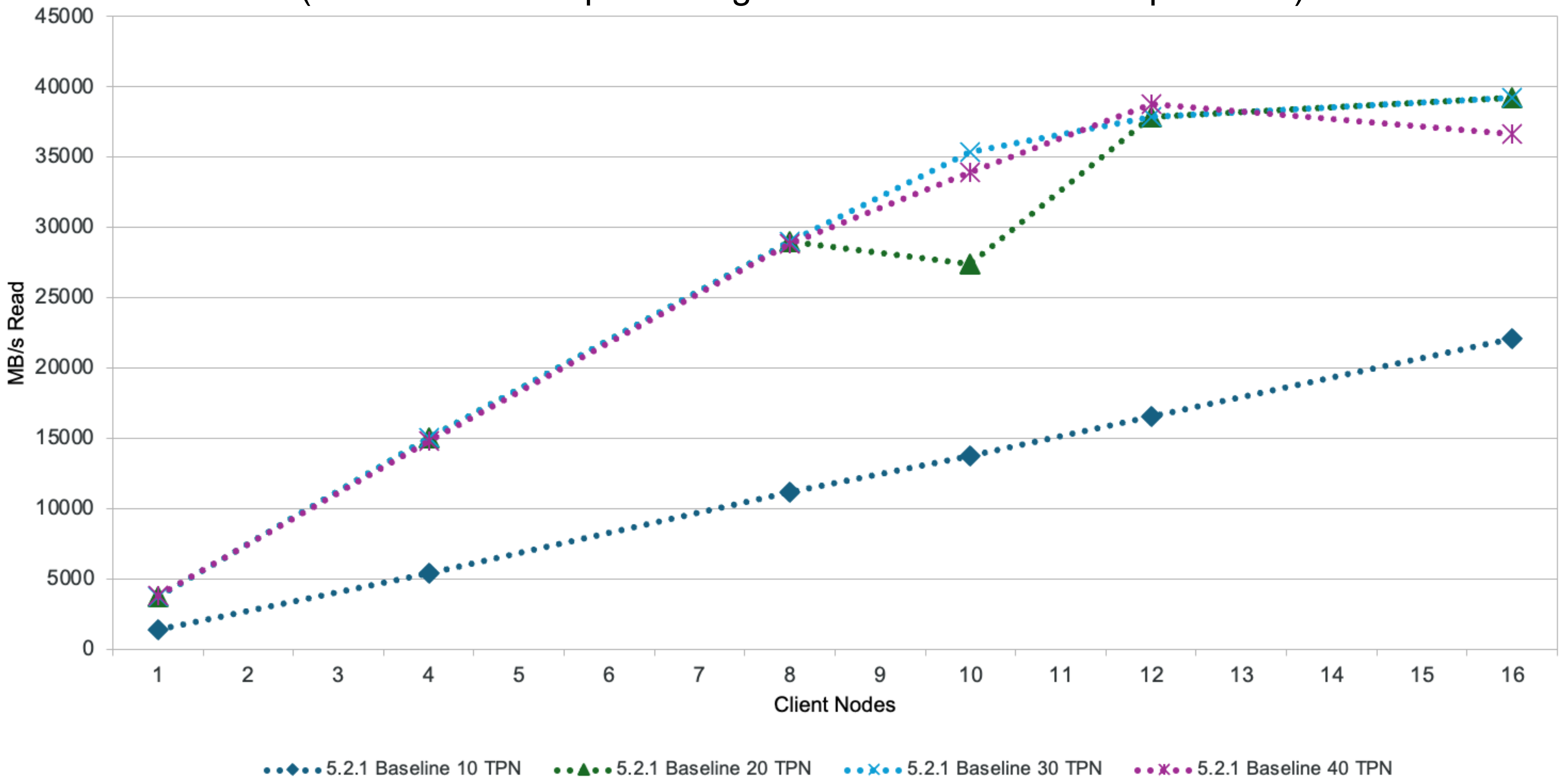
These kinds of waiters are a major bottleneck if fine-grain-sharing is not used for ior-hard-read:

```
[root@fscc-sr655v3-16 ~]# mmfsadm dump waiters
Waiting 0.0181 sec since 2024-09-10_11:00:23, monitored, thread 37228 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0177 sec since 2024-09-10_11:00:23, monitored, thread 40174 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0174 sec since 2024-09-10_11:00:23, monitored, thread 31938 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0136 sec since 2024-09-10_11:00:23, monitored, thread 40170 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0136 sec since 2024-09-10_11:00:23, monitored, thread 40188 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0136 sec since 2024-09-10_11:00:23, monitored, thread 40175 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0134 sec since 2024-09-10_11:00:23, monitored, thread 40173 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40177 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40169 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 33373 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 38095 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40165 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 32333 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40166 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40167 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40178 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0133 sec since 2024-09-10_11:00:23, monitored, thread 40168 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0132 sec since 2024-09-10_11:00:23, monitored, thread 39202 PrefetchWorkerThread: on ThCond 0x18024BA4AA0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0131 sec since 2024-09-10_11:00:23, monitored, thread 40176 PrefetchWorkerThread: on ThCond 0x18024BA30A0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
Waiting 0.0131 sec since 2024-09-10_11:00:23, monitored, thread 40164 PrefetchWorkerThread: on ThCond 0x18024BA30A0 (SyncPairCondvar), reason 'waiting for weak exclusive ThSXLock'
```

Lab Performance of io500 ior-hard-read on 5.2.1

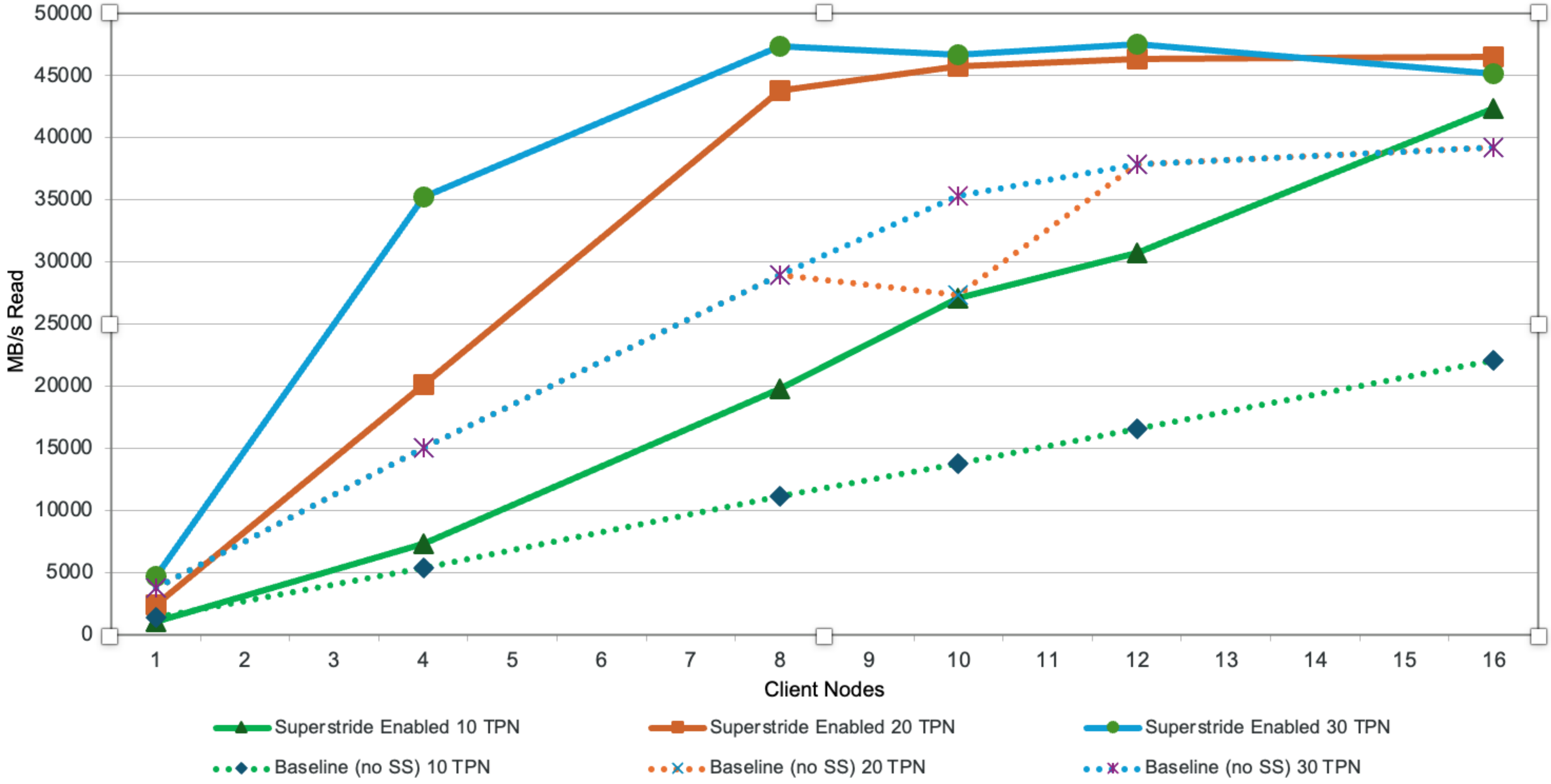
Each Result is the Maximum of 3 Iterations/Runs on an ESS3500 with 24 NVMe Drives

(Performance Stops Scaling at between 20-30 tasks per node)



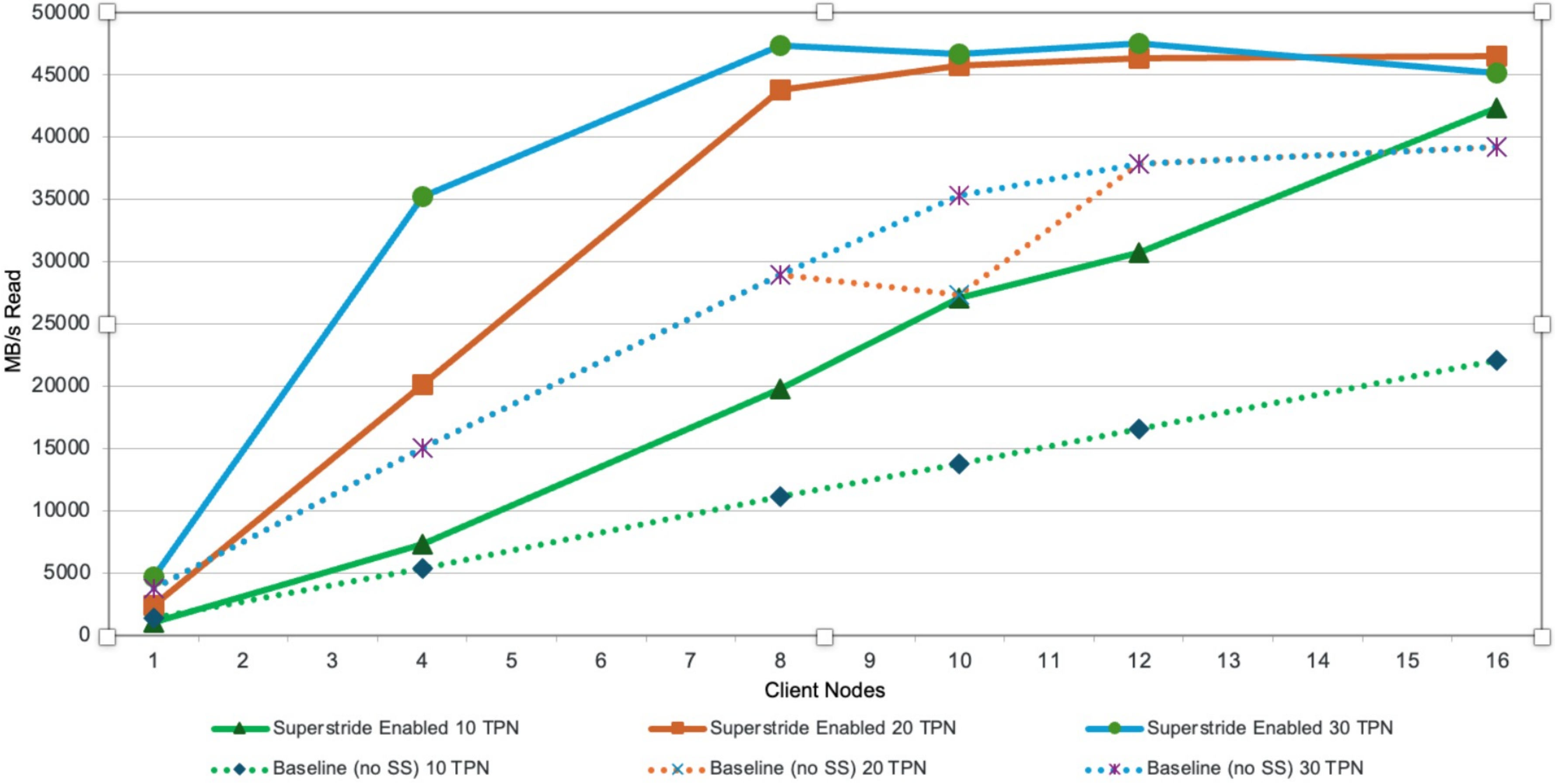
Current Lab Performance of io500 ior-hard-read on pre-ga 5.2.2

Each Result is the Maximum of 3 Iterations/Runs on an ESS3500 with 24 NVMe Drives

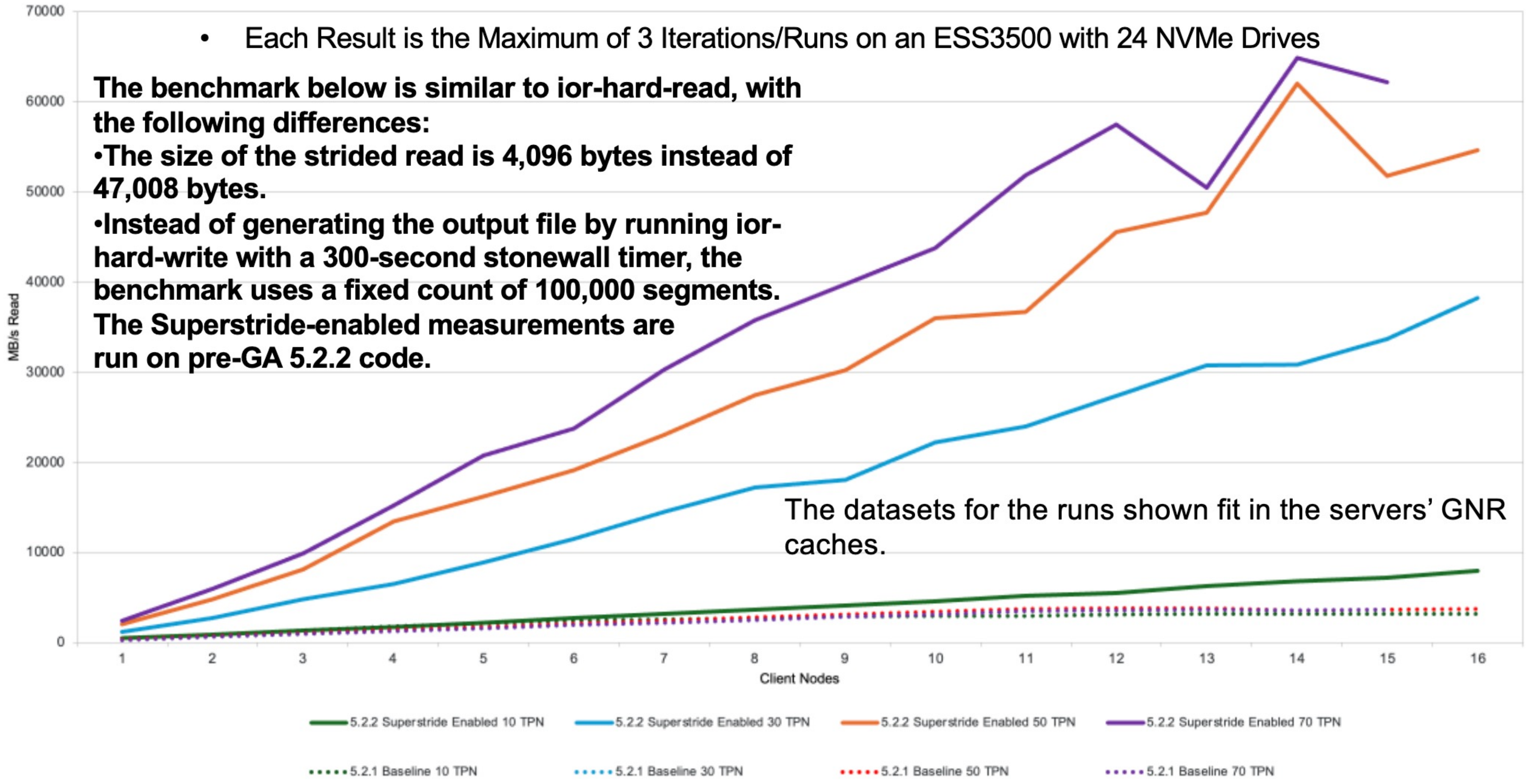


Current Lab Performance of io500 ior-hard-read

- Each Result is the Maximum of 3 Iterations/Runs on an ESS3500 with 24 NVMe Drives
- The new superstride function in the pre-GA Scale 5.2.2 code improves ior-hard-read scaling



Current Lab Performance of Customer Variant of ior-hard-read



- Each Result is the Maximum of 3 Iterations/Runs on an ESS3500 with 24 NVMe Drives

The benchmark below is similar to ior-hard-read, with the following differences:
•The size of the strided read is 4,096 bytes instead of 47,008 bytes.
•Instead of generating the output file by running ior-hard-write with a 300-second stonewall timer, the benchmark uses a fixed count of 100,000 segments.
The Superstride-enabled measurements are run on pre-GA 5.2.2 code.

The datasets for the runs shown fit in the servers' GNR caches.

Mmap: The Basics

- The mmap() facility allows a process to create a memory-mapped region for a file, enabling direct memory access to the file's contents.

- Instead of using traditional read() and write() system calls, a mapped file can be accessed and modified as if it were part of the program’s memory.

- This feature provides a convenient way to manipulate file data using pointers, offering an alternative flexible I/O method to using POSIX calls.

  - Side-note: According to POSIX (http://www.unix.org/single_unix_specification), the behavior of a file system is undefined if normal read or write system calls are issued against a file that is at the same time being accessed through mmap: The application must ensure correct synchronization when using mmap() in conjunction with any other file access method, such as read() and write(), standard input/output, and shmat().

- Changes made to the mapped memory are automatically reflected in the file (for writeable mappings), while reading from the mapped memory fetches data directly from the file.

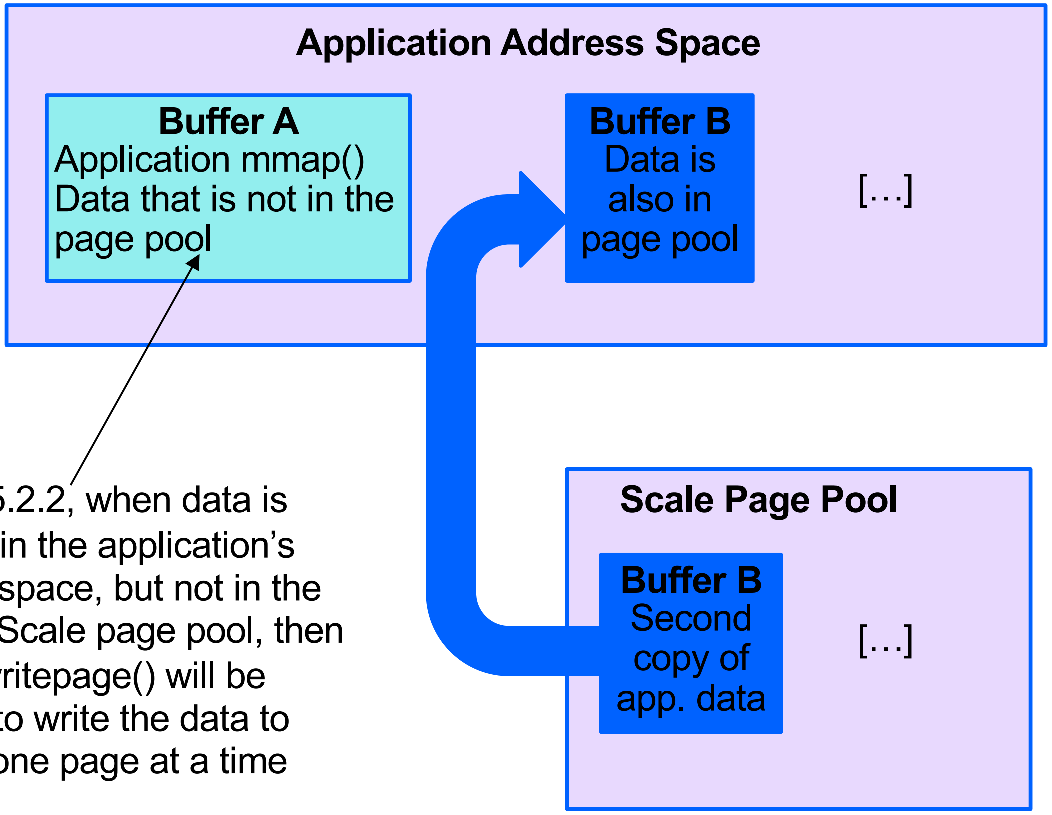
Improvement coming for mmap Writes in 5.2.2

- When a user modifies memory that has been mmaped the data isn’t necessarily written to storage immediately. Writes to storage can occur for multiple reasons, including:
  - Periodic writebacks of data
  - Memory pressure
  - An explicit request from the user via msync()
  - An munmap() call to end the mapping (this is implicit in the exit flow for a process on Linux)
  - Scale will flush dirty mmap pages in response to operations such as handling token revokes and initiating the snapshot quiesce flow

- Prior to 5.2.2, the flushing of dirty mmap data is done via Direct I/O one page at a time if the data to be written is not in the page pool

- Coming in 5.2.2, a new internal gpfs_i_writepages() function is slated to be added to provide better write performance for the case in which mmaped data to be written is not available in the page pool -- Storage Scale will keep a list of virtually contiguous mapped pages and attempt to write up to a full file system block of data in a single I/O operation when the pages are not in the page pool
  - We’ve seen mmap write performance improvements in the range of 2x to 10x for such cases

Improvement coming for mmap Writes in 5.2.2 cont.



- Prior to 5.2.2, when data is mapped in the application's address space, but not in the Storage Scale page pool, then gpfs_i_writepage() will be invoked to write the data to storage one page at a time

- Prior to 5.2.2, when data is in both in the page pool and the application's address space, optimal performance will be obtained writing this data to storage because we can write larger chunks (typically all the virtually contiguous data up to a full block), but the down-side is that this path requires maintaining two copies of the data

- In 5.2.2, a new gpfs_i_writepage**s**() function is added so that multiple pages can be flushed to storage in a single operation, when the data is not in the pagepool (e.g. Buffer A in the shown diagram)

- Writing data in larger chunks makes more efficient use of storage and network bandwidth

Example mmap Sequential Write Performance Test with iozone

```
iozone -r 4096k -s 320g -i 0 -C -B -+u -+m client.list.iozone -+n -w -t 16
        Iozone: Performance Test of File I/O
                Version $Revision: 3.506 $
[…]

        Record Size 4096 kB
        File size set to 335544320 kB
        Using mmap files
        CPU utilization Resolution = 0.000 seconds.
        CPU utilization Excel chart enabled
        Network distribution mode enabled.
        No retest option selected
        Setting no_unlink
        Command line used: iozone -r 4096k -s 320g -i 0 -C -B -+u -+m
client.list.iozone -+n -w -t 16
        Output is in kBytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 kBytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
        Throughput test with 16 processes
        Each process writes a 335544320 kByte file in 4096 kByte records
```

Example mmap Sequential Write Performance Test with iozone with Scale 5.2.1

```
        Test running:
        Children see throughput for 16 initial writers  = 3434808.53 kB/sec
        Min throughput per process                      =  172065.73 kB/sec
        Max throughput per process                      =  242590.41 kB/sec
        Avg throughput per process                      =  214675.53 kB/sec
        Min xfer                                        = 237998080.00 kB
        CPU Utilization: Wall time 1846.320    CPU time 8531.498    CPU
utilization 462.08 %

        Child[0] xfer count = 296591360.00 kB, Throughput =  214420.53 kB/sec,
wall=1536.026, cpu=532.873, %= 34.69
        Child[1] xfer count = 324034560.00 kB, Throughput =  234266.62 kB/sec,
wall=1420.051, cpu=517.188, %= 36.42
Child[15] xfer count = 253612032.00 kB, Throughput =  183180.95 kB/sec,
wall=1802.350, cpu=576.398, %= 31.98

iozone test complete.
```

```
cat client.list.iozone
fscc-sr655v3-1 /gpfs/bb8nvme /usr/local/bin/iozone
[…]
fscc-sr655v3-16 /gpfs/bb8nvme /usr/local/bin/iozone
```

Example mmap Sequential Write Performance Test with iozone with pre-GA Scale 5.2.2

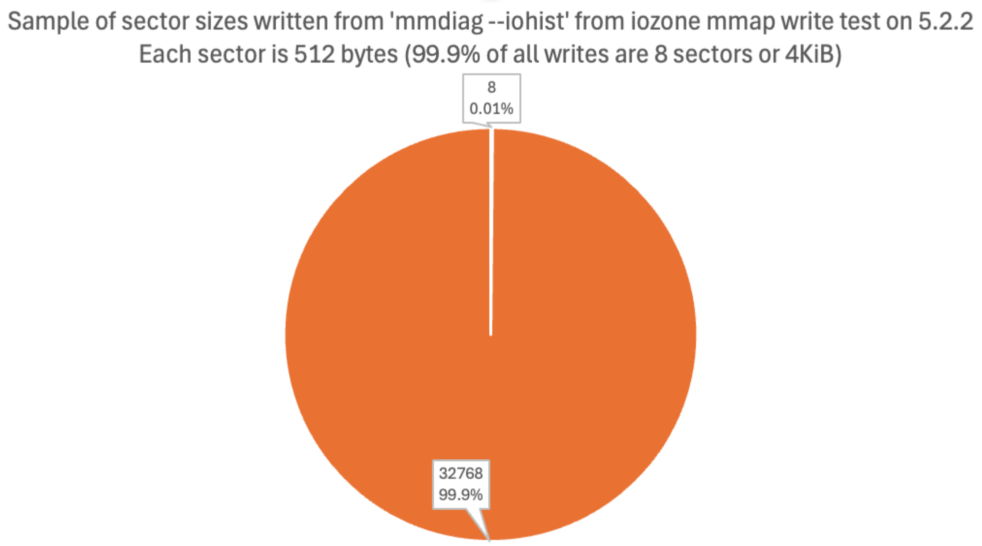
Test running:

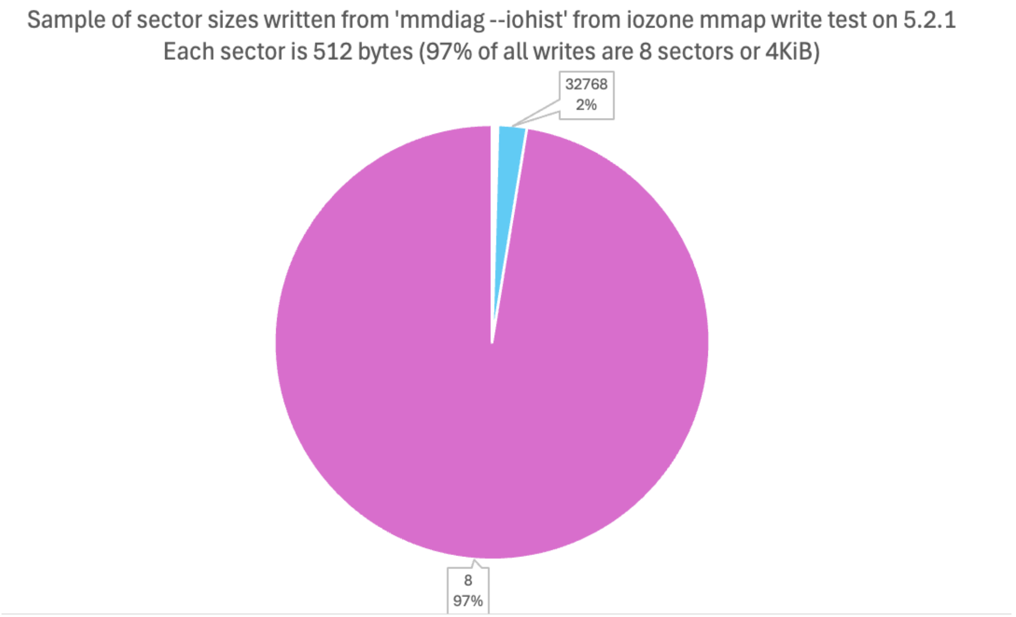
```
        Children see throughput for 16 initial writers  = 8742036.52 kB/sec
        Min throughput per process                      =  188138.77 kB/sec
        Max throughput per process                      =  749017.94 kB/sec
        Avg throughput per process                      =  546377.28 kB/sec
 CPU Utilization: Wall time 1671.646    CPU time 6887.430    CPU utilization
412.01 %

        Child[0] xfer count = 335212544.00 kB, Throughput =  748271.31 kB/sec,
wall=448.362, cpu=362.127, %= 80.77
        Child[1] xfer count = 332029952.00 kB, Throughput =  741161.50 kB/sec,
wall=452.867, cpu=365.989, %= 80.82
[…]
Child[15] xfer count = 85319680.00 kB, Throughput =  190288.11 kB/sec,
wall=1635.845, cpu=573.107, %= 35.03
```

```
For the same example, the Scale pre-GA 5.2.2 code get 8.7 GB/s write throughput
                      the Scale GA     5.2.1 code got 3.4 GB/s write throughput
```

Comparing Application Write I/O Sizes for mmap Write pre-GA Scale 5.2.2 vs 5.2.1

- Sampling ‘mmdiag –iostat for the IO requests submitted by the clients:
  - almost all the write I/Os are full block I/Os (32768 sectors = 16M) in the case of the pre-GA 5.2.2 run
  - almost all the write I/Os are 4KiB in the Scale 5.2.1 run





NFS performance improvements – Configuration

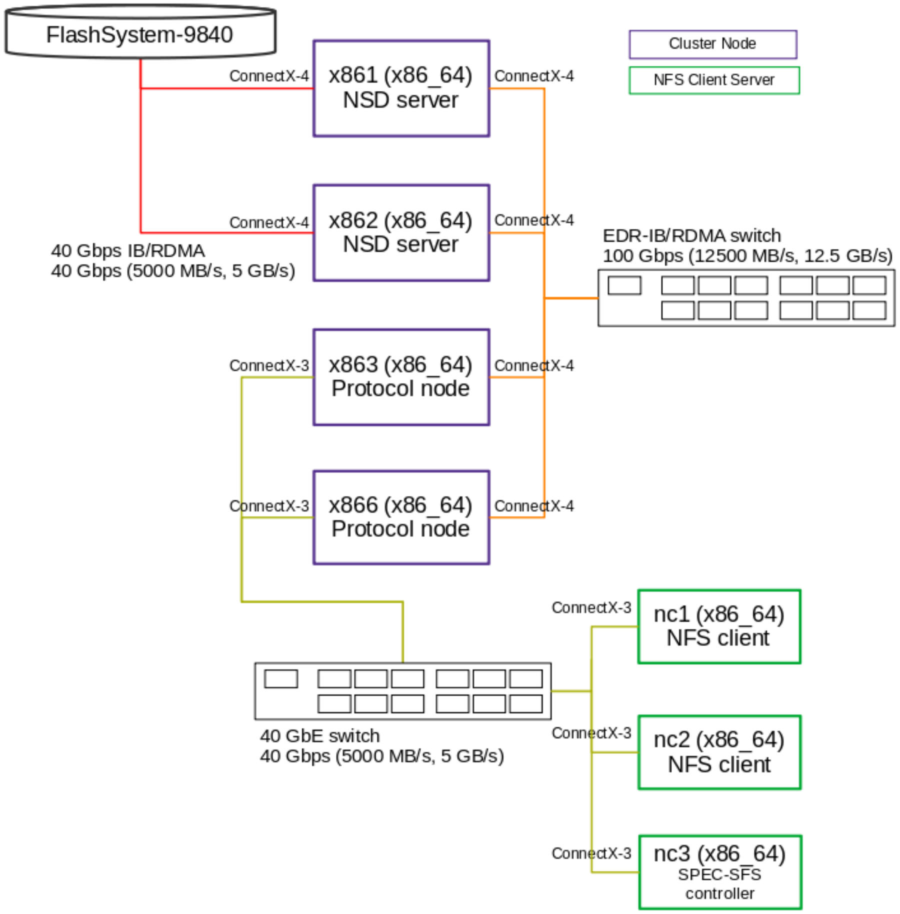
The evaluation compares previous GA code GPFS 5.1.9 + Ganesha 4.3 against current GPFS 5.2.0 + Ganesha 5.7.

Ganesha 4.3 = 4.3-ibm073.09
Ganesha 5.7 = 5.7-ibm017.00

Relevant configuration parameters:

maxFilesToCache 8M
maxStatCache 10M

The evaluation was done with clients:
NFSv3, NFSv4.0, NFSv4.1.



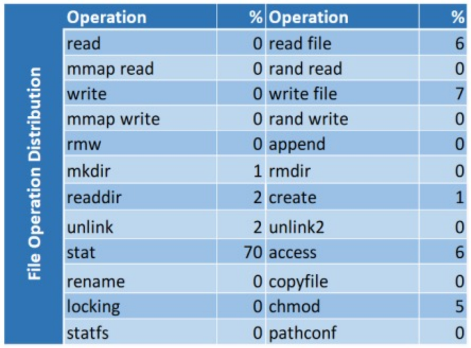
NFS performance improvements – Summary

During 5.2.0 NFS performance regression evaluation we observed significant IOPS improvement in all supported NFS clients with SPEC-SFS SWBUILD benchmark. Most noticeably with NFSv4.1.

The main contributor of this improvement is the NFS-Ganesha Meta Data Cache component (aka MDCACHE), which was revised in Ganesha 5. MDCACHE provides the basic file-handle cache as well as attribute and directory entry caching. Reference: https://github.com/nfs-ganesha/nfs-ganesha/wiki/Stacked-FSAL#mdcache-fsal

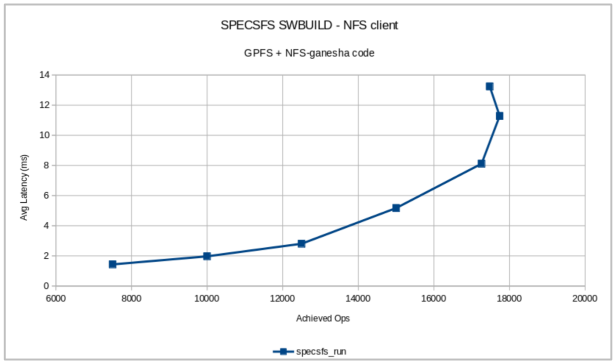
The software build (SWBUILD) workload type is a classic meta-data intensive build workload. Conceptually, these tests are similar to running Unix 'make' against several tens of thousands of files. This workload consumes a lot of CPU and memory. The runs of this benchmark usually have a similar curve.

The overall procedure consist in applying a controlled amount of load and increase it, until the system shows signs of overload (average response time exceeding 10 ms).



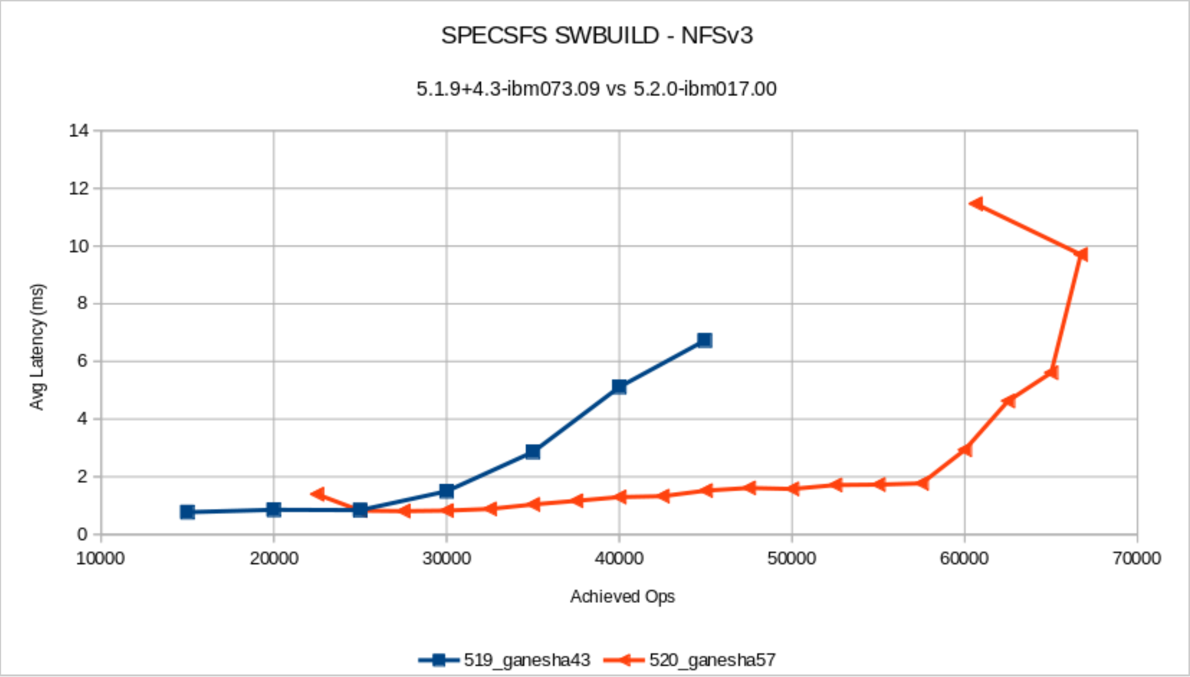
File Operation Distribution

Operation	%	Operation	%
read	0	read file	6
mmap read	0	rand read	0
write	0	write file	7
mmap write	0	rand write	0
rmw	0	append	0
mkdir	1	rmdir	0
readdir	2	create	1
unlink	2	unlink2	0
stat	70	access	6
rename	0	copyfile	0
locking	0	chmod	5
statfs	0	pathconf	0



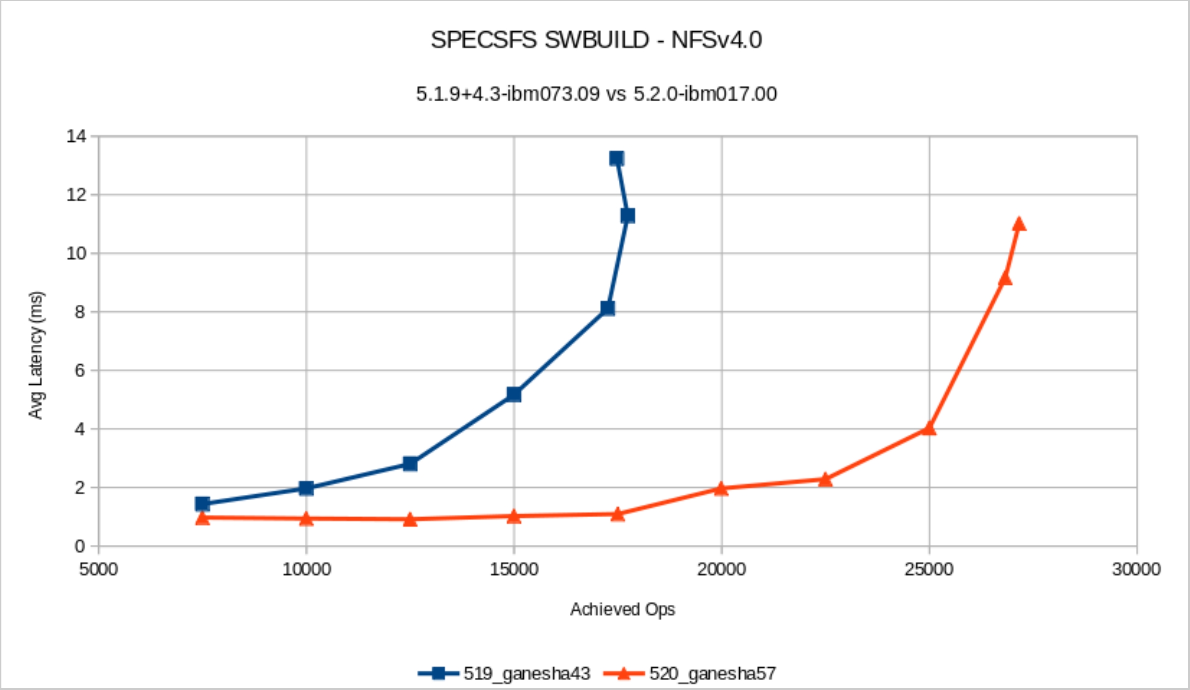
NFS performance improvements – SWBUILD with NFSv3 client

With NFSv3, the maximum IOPS for SWBUILD when running GPFS 5.1.9 + 4.3-ibm073.09 (blue line in the graphic) was 45,000 IOPS (around this metric is when runs started being invalid); with GPFS 5.2.0 + 5.7-ibm017.00 (red line in the graphic) the maximum IOPS for SWBUILD is 67,500 IOPS. That is 50% performance improvement.



NFS performance improvements – SWBUILD with NFSv4.0 client

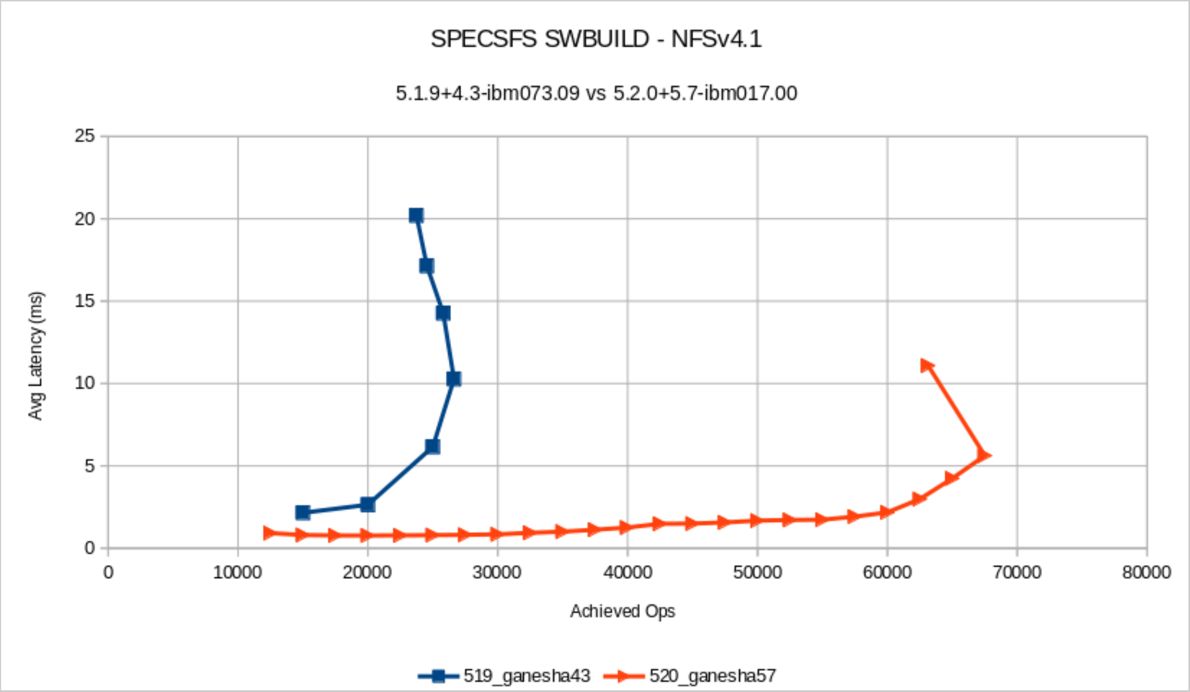
With NFSv4.0, the maximum IOPS for SWBUILD when running GPFS 5.1.9 + 4.3-ibm073.09 (blue line in the graphic) was 17,500 IOPS (around this metric is when runs started being invalid); with GPFS 5.2.0 + 5.7-ibm017.00 (red line in the graphic) the maximum IOPS for SWBUILD is 27,500 IOPS. That is 57% improvement.



NFS performance improvements – SWBUILD with NFSv4.1 client

With NFSv4.1, the maximum IOPS for SWBUILD when running GPFS 5.1.9 + 4.3-ibm073.09 (blue line in the graphic) was 25,000 IOPS (around this metric is when runs started being invalid); with GPFS 5.2.0 + 5.7-ibm017.00 (red line in the graphic) the maximum IOPS for SWBUILD is 70,000 IOPS. That is 180% performance improvement.

Note: NFSv4.1 is better or at least on par with NFSv3.



Thanks!