

BLUE WATERS

SUSTAINED PETASCALE COMPUTING

December 3, 2013

Optimizing Applications on Blue Waters

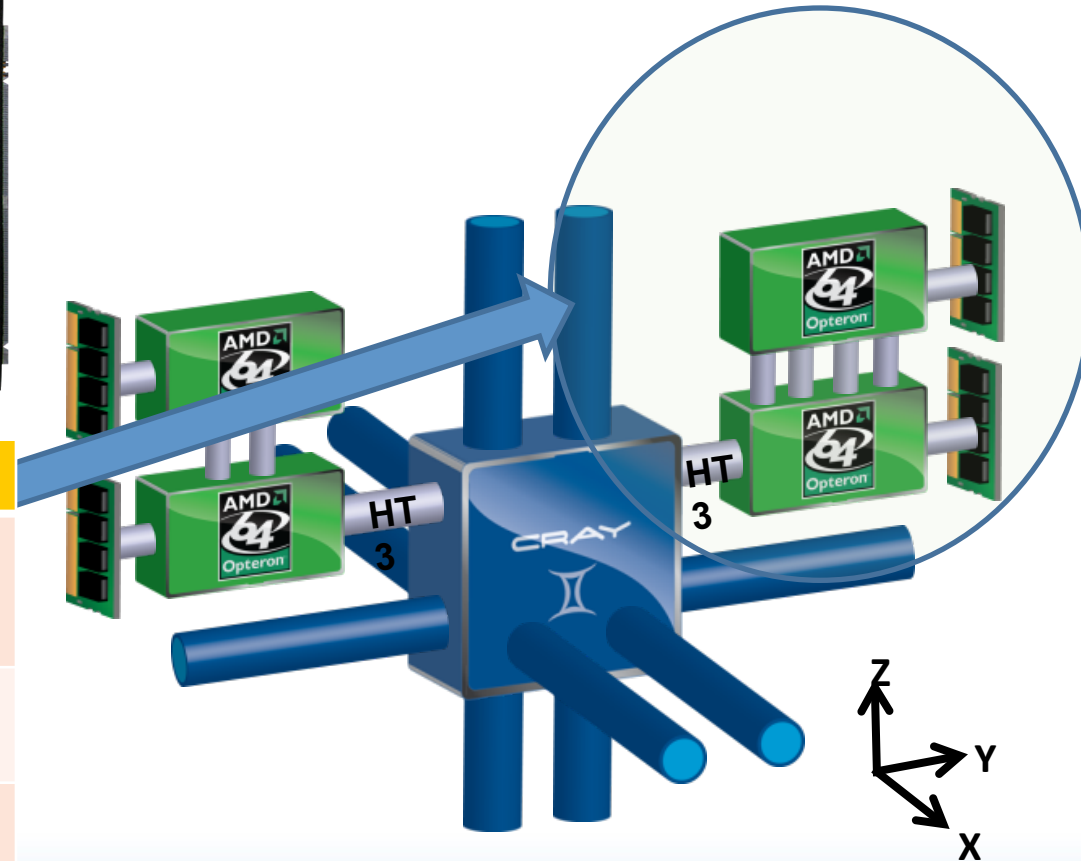
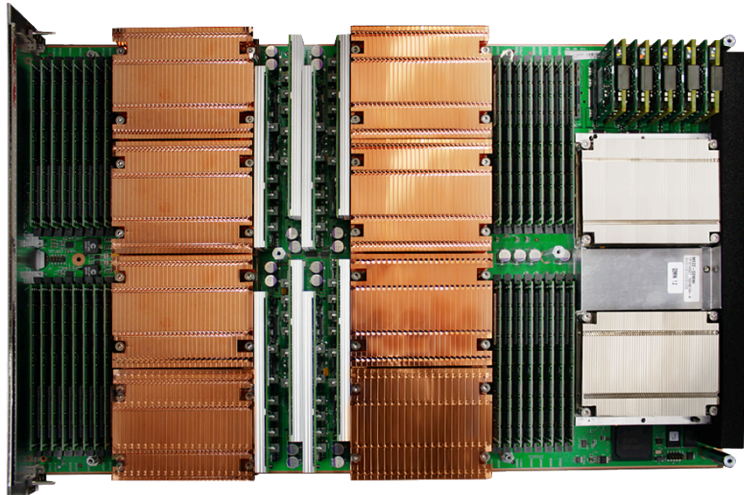
NCSA Science and Engineering Applications Support
Robert Brunner



GREAT LAKES CONSORTIUM
FOR PETASCALE COMPUTATION

CRAY®

Cray XE6 Blade and Node

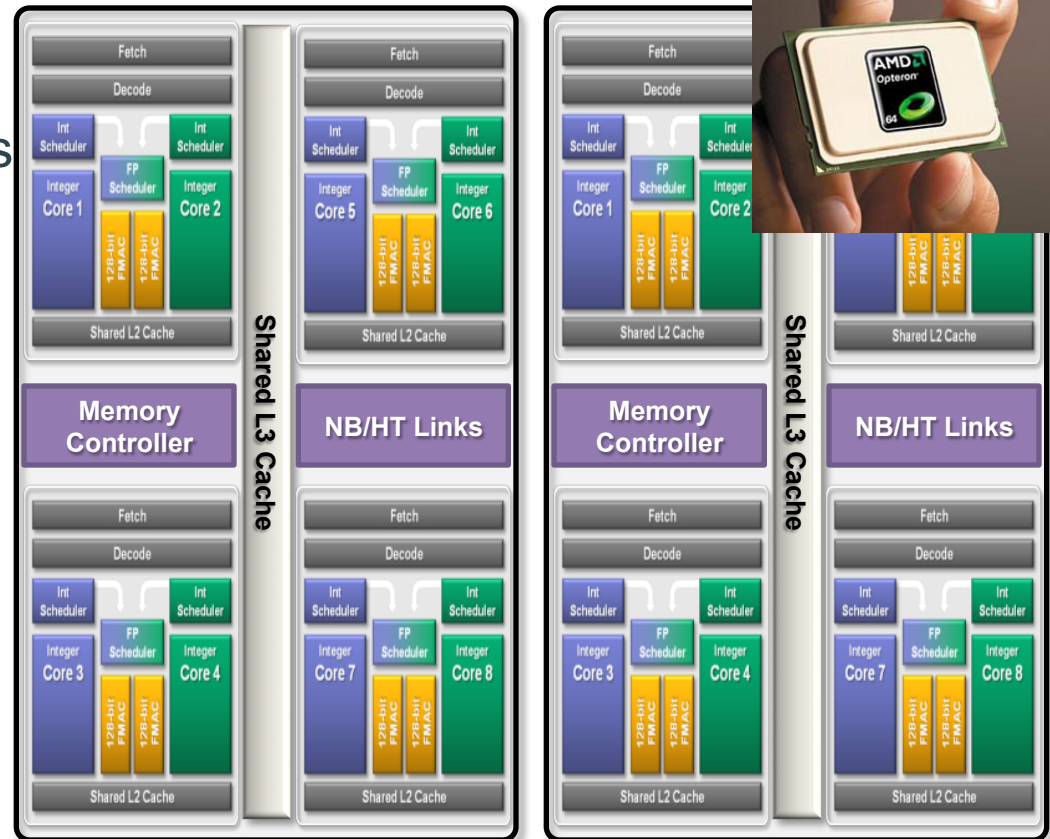


Node Characteristics

Number of Cores	16 Core modules (2 AMD 6276 Interlagos processors @2.3 GHz)
Peak Performance	313 Gflops/sec
Memory Size	4 GB per core-mod 64 GB per node
Memory Bandwidth (Peak)	102.4 GB/sec

AMD 6276 Interlagos Processor

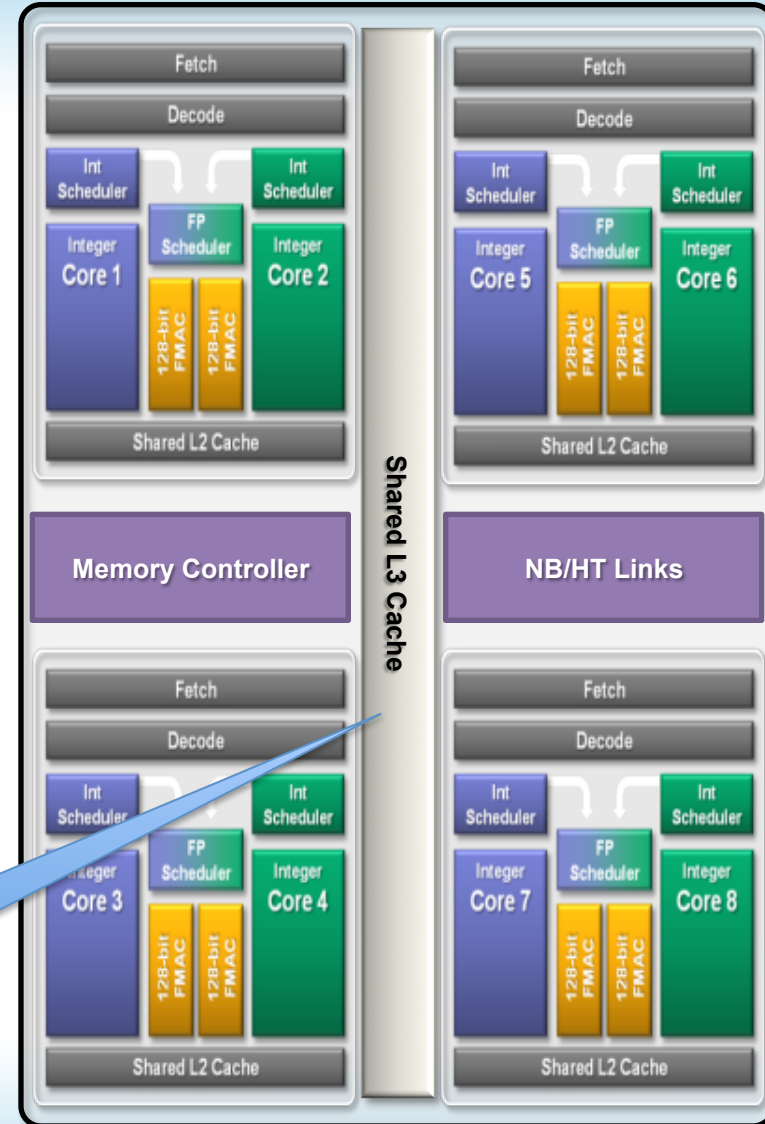
- Each processor die is composed of 4 core modules
- The 4 core modules share a memory controller and 8 MB L3 data cache on one die
- Two die are packaged on a multi-chip module to form a G34-socket Interlagos processor
- Package contains
 - 8 core modules
 - 16 MB L3 Cache
 - 4 DDR3 1600 memory channels



Interlagos

- Four Core Modules per die
- Two Integer cores and one FP core per Core Module
- OS treats each Interlagos as 16 cores (i.e. 32 per XE6 node)
- Each die shares L3 cache

1 of 2 dies



Compiler Options - Topics

- Available (Supported) Compilers
- Where to Start
- Compiler Choices – Relative Strength
- Compiler Options focused on
 - Optimization
 - Debugging

Available Compilers

- Cray Compilers (Cray Compiling Environment (CCE))
 - Provided additional support for Fortran 2003, Co-arrays, UPC, PGAS
- GNU Compiler Collection (GCC)
- Portland Group Inc (PGI) Compilers
- All provide Fortran, C, C++, OpenMP support
- UPC, PGAS, (limited) OpenACC support (Cray, PGI)
- So which compiler do I choose?
 - Experiment with various compilers
 - Work with your BW POC
 - Mixing libraries created by different compilers may cause issues

Where to Start

- Unless you have a very good reason, always use compiler wrappers
 - “module load PrgEnv-[cray,gnu,pgi]”
 - Compiler wrappers: ftn, cc, CC
 - Additional libraries are automatically linked in
 - Optimization targets automatically set
- For most applications, using default settings work very well
- The OpenMP threaded BLAS/LAPACK libraries are used
 - The serial version is used if “OMP_NUM_THREADS” is not set or set to 1

Use the Best Compiler

- The best compiler may not be the same for every application.
- Work with BW staff to compare compilers

Compiler Choices – Relative Strength

- CCE – Outstanding Fortran, Very good C and okay C++
 - Very good vectorization
 - Very good Fortran language support; only real choice for coarrays
 - C support is very good, with UPC support
 - Very good scalar optimization and automatic parallelization
 - Clean implementation of OpenMP 3.0 with tasks
 - Cleanest integration with other Cray tools (Performance tools, debuggers, upcoming productivity tools)
 - No inline assembly support
 - Excellent support from Cray (bugs, issues, performance etc)

Compiler Choices – Relative Strength

- PGI – Very good Fortran, okay C and C++
 - Good vectorization
 - Good functional correctness with optimization enabled
 - Good manual and automatic prefetch capabilities
 - Company focused on HPC market
 - Excellent working relationship with Cray, good bug responsiveness

Compiler Choices – Relative Strength

- GNU – so-so Fortran, outstanding C and C++ (If you ignore vectorization)
 - Obviously, the best gcc compatibility
 - Scalable optimizer was recently rewritten and is very good
 - Vectorization capabilities focus mostly on inline assembly
 - Few releases have been incompatible with each other and require recompilation of modules (4.3, 4.4, 4.5)

Recommended CCE Compilation Options

- Use default optimization levels
 - It's the equivalent of most other compilers `-O3` or `-fast`
- Use `-O3`, `fp3` (or `-O3 -hfp3` or some variation)
 - `-O3` gives slightly more than `-O2`
 - `-hfp3` gives a lot more floating point optimizations, esp 32 bit
- If an application is intolerant of floating point reassociation, try lower hfp number, try `hfp1` first, only `hfp0` if absolutely necessary
 - Might be needed for tests that require strict IEEE conformance
 - Or applications that have validated results from different compiler
- Do not suggest using `-Oipa5`, `-Oaggress` and so on; higher numbers are not always correlated with better performance
- Compiler feedback : `-rm` (fortran), `-hlist=m` (C)
- If don't want OpenMP : `-xomp` or `-Othread0` or `-hnoomp`
- Manpages : `crayftn`, `craycc`, `crayCC`

Loopmark : Compiler Feedback (CCE)

- Compiler can generate an filename.lst file
- Contains annotated listing of your source code with letter indicating important optimizations
- Loopmark legend

Primary Loop Type -----	Modifiers -----
A - Pattern matched	a - atomic memory operation
C - Collapsed	b - blocked
D - Deleted	c - conditional and/or computed
E - Cloned	f - fused
G - Accelerated	g - partitioned
I - Inlined	i - interchanged
M - Multithreaded	m - partitioned
V - Vectorized	n - non-blocking remote transfer
	p - partial
	r - unrolled
	s - shortloop
	w - unwound

Starting Point for PGI Compilers

- Suggested Option : -fast
- Interprocedural analysis allows the compiler to perform whole program optimizations : `-Mipa=fast(,safe)`
- If you can be flexible with precision, also try `-Mfprelaxed`
- Option `-Msmartalloc`, calls the subroutine `mallopt` in the main routine, can have a dramatic impact on the performance of program that uses dynamic allocation of memory
- Compiler feedback : `-Minfo=all`, `-Mneginfo`
- Manpages : `pgf90`, `pgcc`, `pgCC`

PGI Compiler Flags

- `-default64` : Fortran driver option for `-i8` and `-r8`
- `-i8, -r8` : Treats INTEGER and REAL variables in Fortran as 8 bytes (use `ftn -default64` option to link the right libraries)
- `-byteswapio` : Reads big endian files in fortran
- `-Mnomain` : Uses `ftn` driver to link programs with the main program (written in C or C++) and one or more subroutines (written in fortran)

PGI Compiler Flags

- It is possible to disable optimizations included with `-fast`, for example `-fast -Mnoire` enables `-fast` and then disables loop redundant optimizations
- `-Mconcur`, `-mprof=mpi`, `-Mmpi` and `-Mscalapack` are no more supported
- Fortran interfaces can be called from C program by inserting an underscore to the respective name
- Pass argument by reference rather than by value
- For example to call `dgetrf()`
- `Dgetrf_(&uplo, &M, &n,);`
- To debug an optimized code, the `-opt` flag will insert debugging information without disabling optimizations

PGI Compiler Flags

- Some compiler options that affect both performance and accuracy
- Lower accuracy is often higher performance, but it is also able to enforce accuracy
 - -Kieee : all floating point (FP) math strictly conforms to IEEE , off by default
 - -Ktrap : Turns processor trapping of FP exceptions
 - -Mdaz : Treat all denormalized numbers as zeros
 - Mflushz : Set SSE to flush-to-zero (on with -fast)
 - -Mfprelaxed : allow to use relaxed (reduced) precision to speed up some floating point optimizations
 - Some compilers turn this on by default, PGI chooses to favor accuracy to speed, by default

Starting Point for GNU Compilers

- `-O3 -ffast-math -funroll-loops`
- Compiler feedback : `-ftree-vectorizer-verbose=2`
- Manpages : `gfortran`, `gcc`, `g++`

Numerical Libraries Overview

- Many commonly-used packages are available on Blue Waters
- Typically can link with most or all combinations of compiler, language, and parallel programming model
- Use the “`module`” command to select a particular version
- Will try to accommodate special installation requests (*can't install “Everything under the Sun” due to scalability and other considerations*)

Cray Scientific Library (libsci)

- Contains optimized versions of several popular scientific software routines
- Available by default; can change versions with “`module avail`” and “`module load xt-libsci [/version]`”
 - BLAS, BLACS
 - LAPACK, ScaLAPACK
 - FFT, FFTW
- Unique to Cray (*affects portability*)
 - CRAFFT, CASE, IRT

PETSc (Argonne National Laboratory)

- Programmable, Extensible Toolkit for Scientific Computing
- Widely-used collection of many different types of linear and non-linear solvers
- Actively under development; very responsive team
- Can also interface with numerous optional external packages (e.g., SLEPC, HYPRE, ParMETIS, ...)
- Optimized version installed by Cray, along with many external packages
- Use “`module load petsc[/version]`”

Other Numerical Libraries

- ACML (AMD Core Math Library)
 - BLAS, LAPACK, FFT, Random Number Generators
- Trilinos (from Sandia National Laboratories)
 - Somewhat similar to PETSc, interfaces to a large collection of preconditioners, solvers, and other computational tools
- GSL (GNU Scientific Library)
 - Collection of numerous computational solvers and tools for C and C++ programs
- All available using “`module load`”

Optimization options

- Hybrid programming model (MPI+OpenMP, *et al*) is usually better
- Try 1, 2, 4, 16, 32 tasks per node

For 1024 nodes:

32 tasks+threads/node:

```
aprun -n 4096 -N 4 -d 8 ./myprog
```

16 tasks+threads/node:

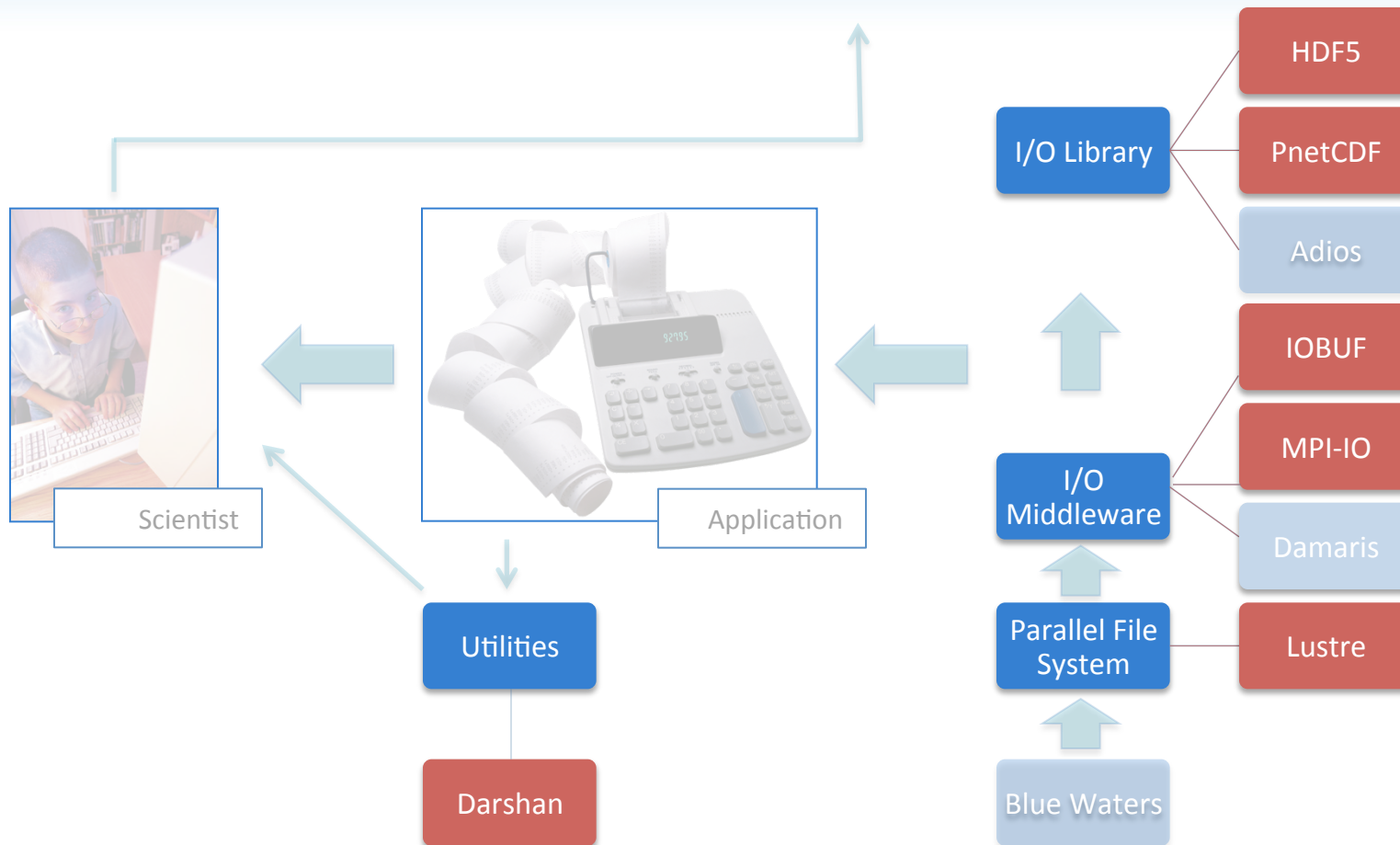
```
aprun -n 4096 -N 4 -d 4 \  
-cc 0,2,4,6:8,10,12,14:16,18,20,22:24,26,28,30 \  
./myprog
```

- Try using `-r 1` to reserve a core for the OS

```
aprun -n 4096 -N 4 -d 7 -r 1 \  
-cc 0-6:8-14:16-22:24-30 ./myprog
```

- Test different compilers, flags
- Use accelerators

This Talk



Common I/O Usage

- Checkpoint files
 - Write-close
 - Size varies
 - Must be written to disk
- Log / history / state files
 - Simple appends
 - Small writes (~kb - ~MB)
 - Can be buffered
- Write-read not very common

- Optimize for write
- Synchronous write

- Optimize for write
- Asynchronous write
- Explicit buffer management or
- Use a library

Available File Systems

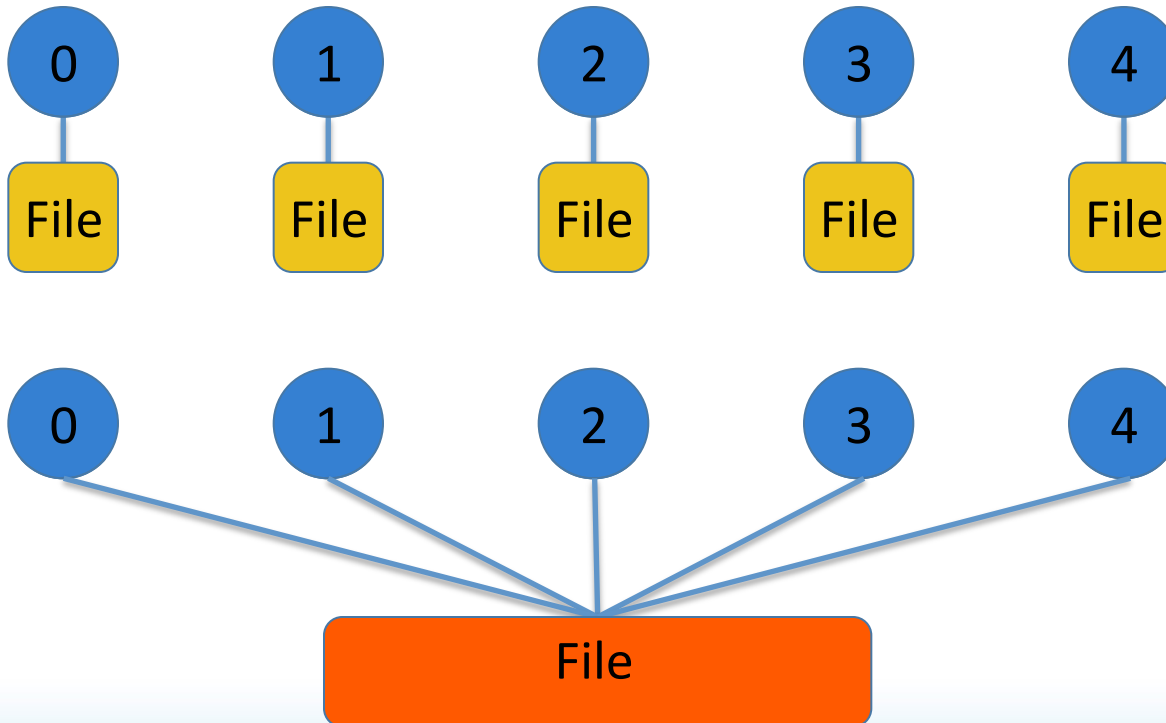
- home
 - 2.2 PB
 - 1TB quota
 - project
 - 2.2 PB
 - 3TB quota
 - scratch
 - 22 PB
 - 500 TB quota
- Three separate file systems
 - Three separate metadata servers
 - User operations in home won't interfere with application IO
 - Project space controlled by the PI

Application I/O: Big Picture Considerations

- Maximize both client I/O and communication bandwidth (without breaking things)
- Minimize management of an unnecessarily large number of files
- Minimize costly post-processing
- Exploit parallelism in the file system
- Maintain portability

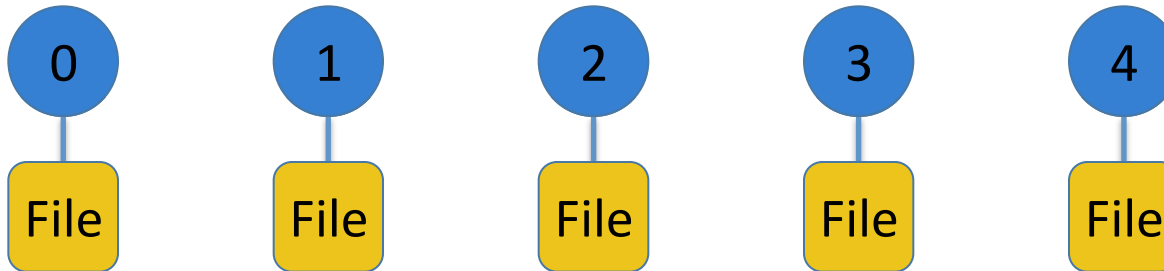
Large Scale I/O in Practice

- Serial I/O is limited by both the I/O bandwidth of a single process as well as that of a single OST
- Two ways to increase bandwidth:



File-Per-Process

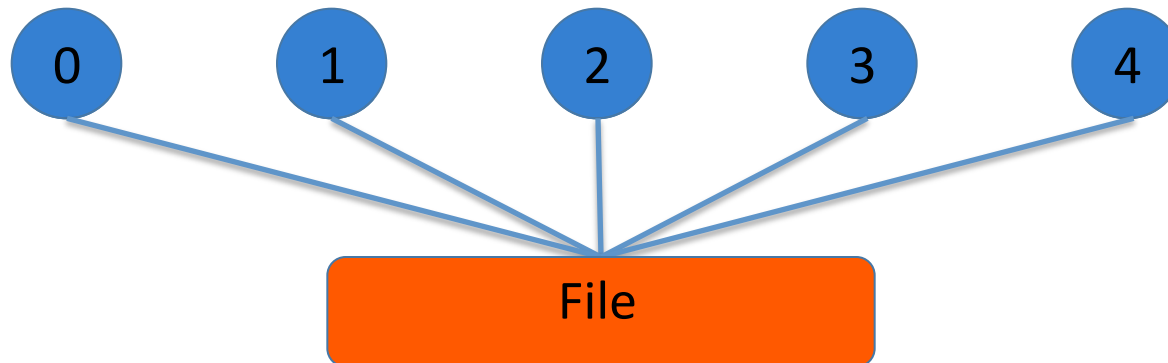
- Each process performs I/O on its own file



- Advantages
 - Straightforward implementation
 - Typically leads to reasonable bandwidth quickly
- Disadvantages
 - Limited by single process
 - Difficulty in managing a large number of files
 - Likely requires post processing to acquire useful data
 - Can be taxing on the file system metadata and ruin everybody's day

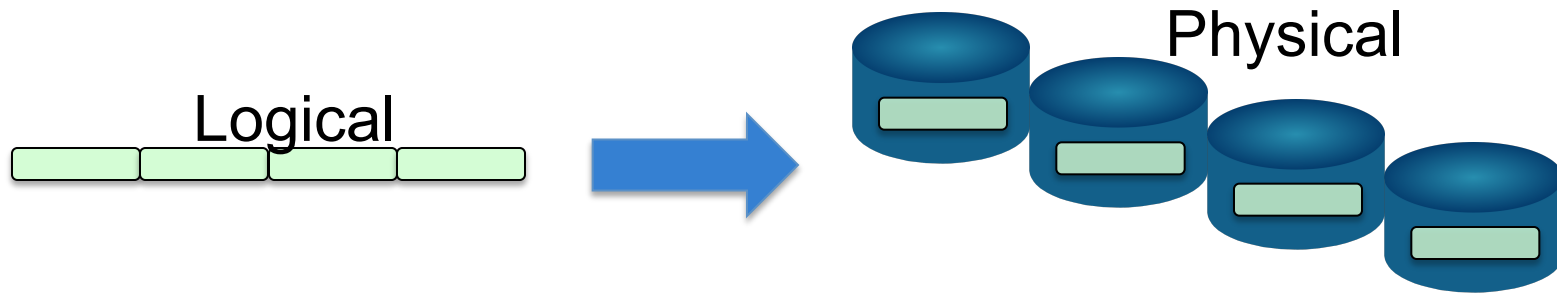
Shared-File

- There is one, large file shared among all processors which access the file concurrently



- Advantages
 - Results in easily managed data that is useful with minimal preprocessing
- Disadvantages
 - Likely slower than file-per-process, if not used properly
 - Additional (one-time!) programming investment

Lustre File System: Striping



- **File striping:** single files are distributed across a series of OSTs
 - File size can grow to the aggregate size of available OSTs (rather than a single disk)
 - Accessing multiple OSTs concurrently increases I/O bandwidth

Performance Impact: Configuring File Striping

- `lfs` is the Lustre utility for viewing/setting file striping info
 - **Stripe count** – the number of OSTs across which the file can be striped
 - **Stripe size** – the size of the blocks that a file will be broken into
 - **Stripe offset** – the ID of an OST for Lustre to start with, when deciding which OSTs a file will be striped across
- Configurations should focus on stripe count/size
- Blue Waters defaults:

```
$> touch test
$> lfs getstripe test
test
lmm_stripe_count:    1
lmm_stripe_size:    1048576
lmm_stripe_offset:  708
      obdidx          objid          objid          group
      708            2161316        0x20faa4        0
```

Setting Striping Patterns

```
$> lfs setstripe -c 5 -s 32m test
```

```
$> lfs getstripe test
```

```
test
```

```
lmm_stripe_count:    5
```

```
lmm_stripe_size:    33554432
```

```
lmm_stripe_offset:  1259
```

obdidx	objid	objid	group
1259	2162557	0x20ff7d	0
1403	2165796	0x210c24	0
955	2163063	0x210177	0
1139	2161496	0x20fb58	0
699	2161171	0x20fa13	0

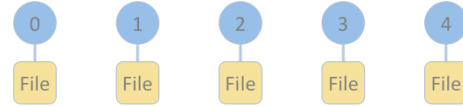
- Note: a file's striping pattern is permanent, and set upon creation
 - `lfs setstripe` creates a new, 0 byte file
 - The striping pattern *can* be changed for a directory; every *new* file or directory created within will inherit its striping pattern
 - Simple API available for configuring striping – portable to other Lustre systems

Striping Case Study

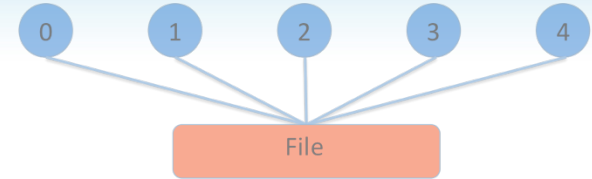
- | Function | Stripe Count = 1 | Stripe Count = 64 | Improvement |
|----------------|------------------|-------------------|-------------|
| Total | 4551.620s | 268.209s | 94.1% |
| loadKernel | 4296.118s | 85.331s | 98.0% |
| loadDamp | 33.767s | 6.144s | 81.8% |
| loadDamp_bycol | 30.085s | 5.712s | 81.0% |

- Code is now CPU bound instead of I/O bound
- Optimization “effort”: `lfs setstripe -c 64`

Striping, and You

- When to use the default stripe count of 1
 - Serial I/O or small files
 - Inefficient use of bandwidth + overhead of using multiple OSTs will degrade performance
 - File-per-process I/O Pattern 
 - Each core interacting with a single OST reduces network costs of hitting OSTs (which can eat your lunch at large scales)
- Stripe size is unlikely to vary performance unless unreasonably small/large
 - Err on the side of small
 - This helps keep stripes **aligned**, or within single OSTs
 - Can lessen OST traffic
 - Default stripe size should be adequate

- Large shared files:
 - Processes ideally access exclusive file regions
 - Stripe size
 - Application dependent
 - Should maximize stripe alignment (localize a process to an OST to reduce contention and connection overhead)
 - Stripe count
 - Should equal the number of processes performing I/O to maximize I/O bandwidth
 - Blue Waters contains 1440 OSTs, the maximum possible for file stripe count is currently 160 (likely to increase soon pending a software update)

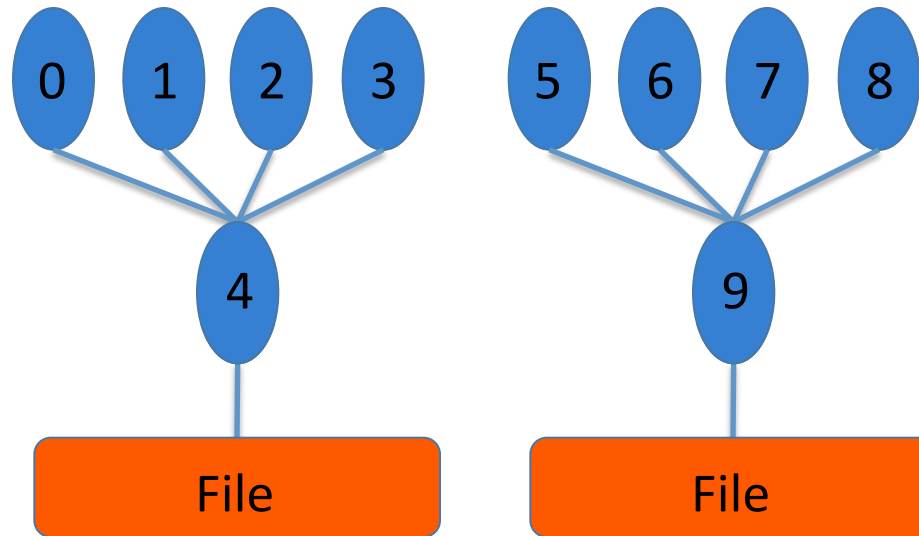


```
$> lfs osts
OBDS
0: snx11001-OST0000_UUID ACTIVE
1: snx11001-OST0001_UUID ACTIVE
.....
1438: snx11003-OST059e_UUID ACTIVE
1439: snx11003-OST059f_UUID ACTIVE
```

And the Winner is... Neither?

- Both patterns increase bandwidth through the addition of I/O processes
 - There are a limited number of OSTs to stripe a file across
 - The likelihood of OST contention grows with the ratio of I/O processes to OSTs
 - Eventually, the benefit of another I/O process is offset by added OST traffic
- Both routinely use all processes to perform I/O
 - A small subset of a node's cores can consume a node's I/O bandwidth
 - This is an inefficient use of resources
- The answer? It depends... but,
 - Think aggregation, a la *file-per-node*

I/O Delegates



- Advantages
 - More control - customize per job size
 - Ex: One file per node, one file per OST
- Disadvantages
 - Additional (one-time!) programming investment

MPI-IO & IOBUF

I/O MIDDLEWARE

Why use I/O Middleware?

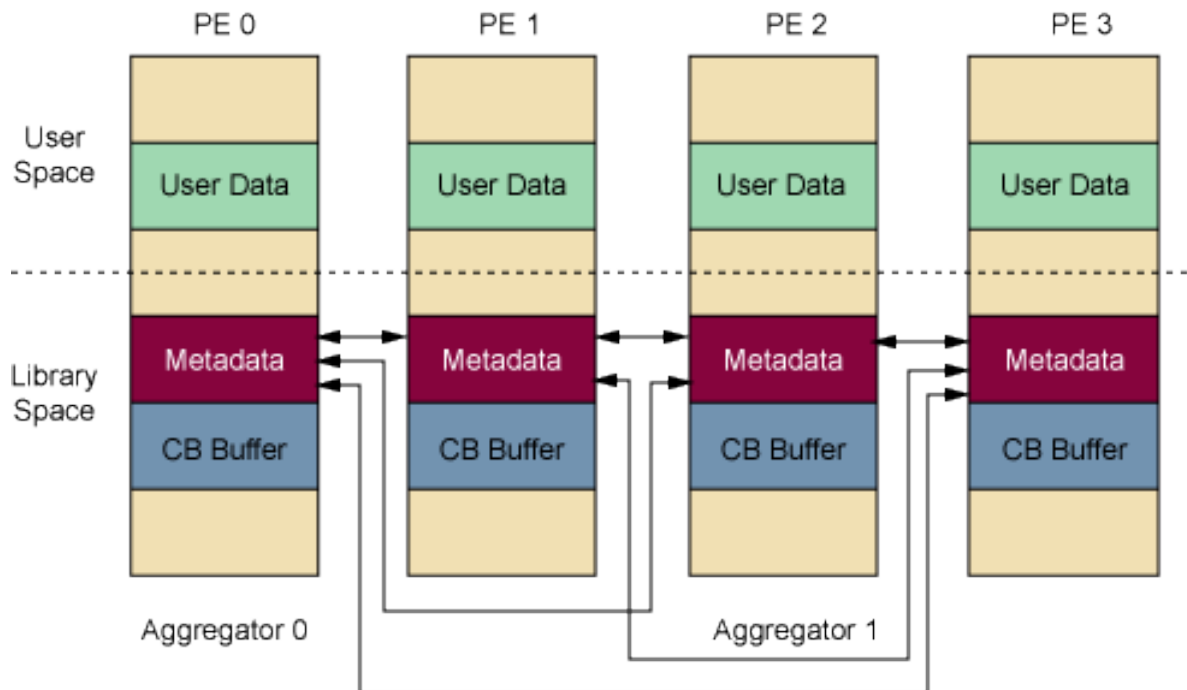
- Derived data types
- Easy to work with shared files
- Derived types + shared files
 - Data is now a series of objects, rather than a number of files
 - On restart from checkpoint, the number of processors need not match the number of files
- Easy read-write of non-contiguous data
- Optimizations possible with little effort

I/O Middleware: MPI-IO

- MPI standard's implementation of **collective** I/O (shared-file)
 - A file is opened by a group of processes, partitioned among them, and I/O calls are collective among all processes in the group
 - Files are composed of native MPI data types
 - Non-collective I/O is also possible
- Uses **collective buffering** to consolidate I/O requests
 - All data is transferred to a subset of processes and aggregated
 - Use `MPICH_MPIIO_CB_ALIGN=2` to enable Cray's collective buffering algorithm
 - *automatic* Lustre stripes alignment & minimize lock contention
 - May not be beneficial when writing small data segments
 - Verified to deliver 25% improvement on BlueWaters for a 1000 rank job
- Use `MPICH_MPIIO_XSTATS [0, 1, 2]` to obtain MPI-IO statistics
- I/O optimizations in high level libraries are often implemented here – be sure any monkeying is careful monkeying

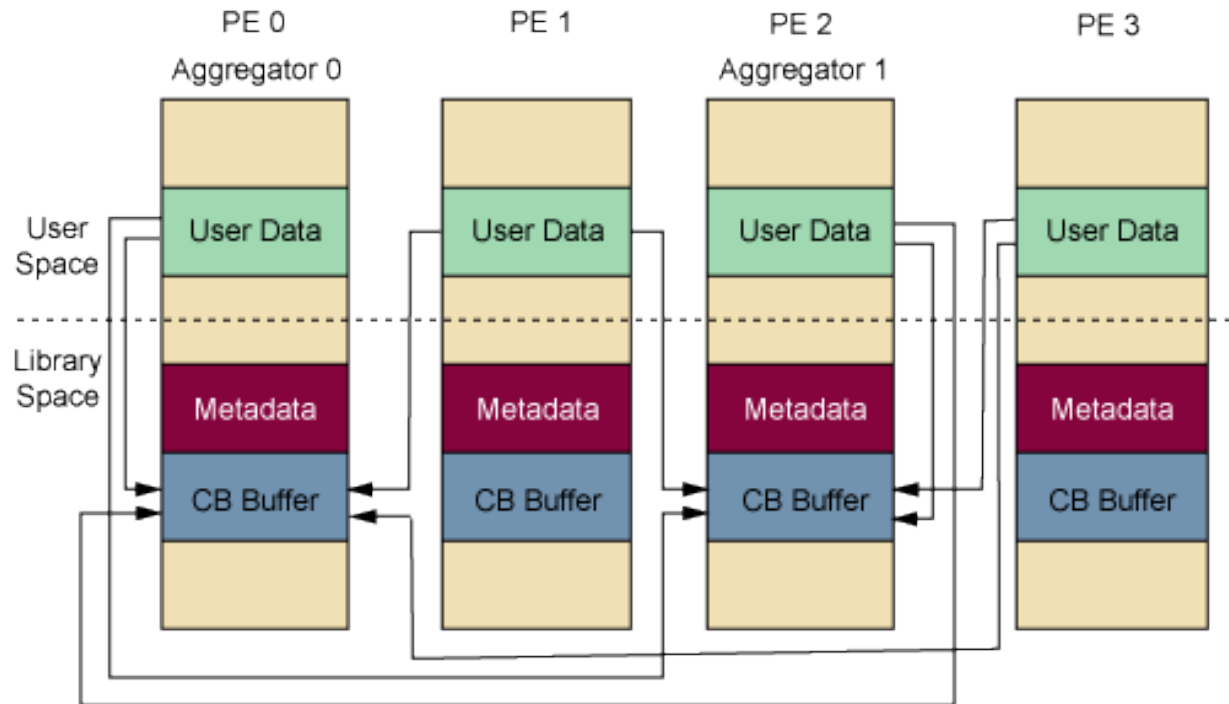
Collective Buffering (1)

- Exchange metadata



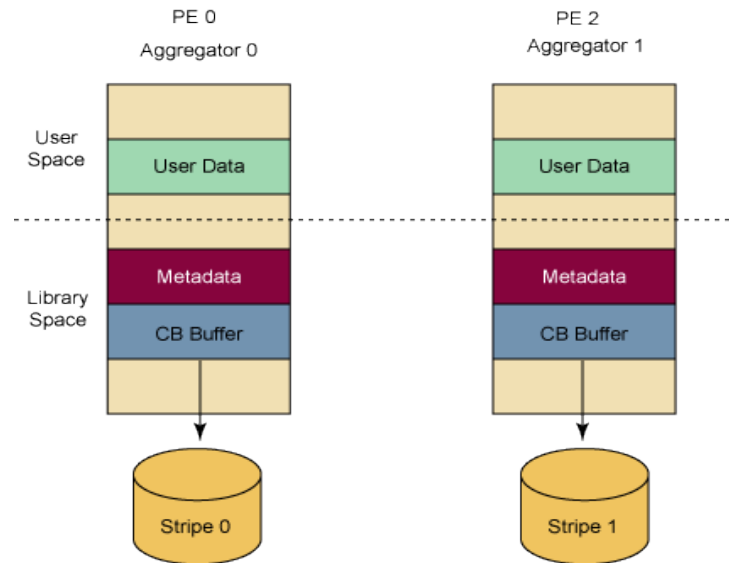
Collective Buffering (2)

- Copy user/application data



Collective Buffering (3)

- Aggregators write to disk



Tuning MPI-IO: CB Hints

- Hints are specified in application code [`MPI_Info_set()`] or as environment variables (`MPICH_MPIIO_HINTS`)
- Collective buffering hints

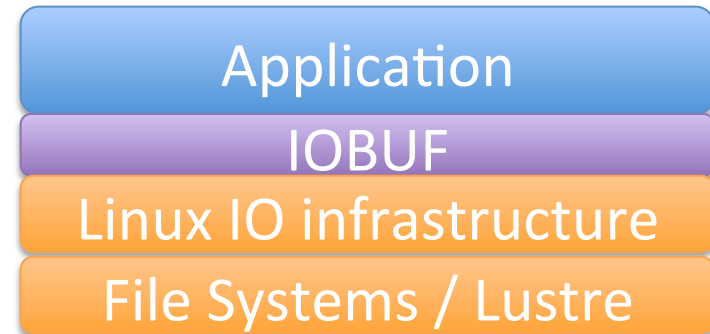
Hint	Description	Default
<code>cb_buffer_size</code>	set the maximum size of a single I/O operation	4MB
<code>cb_nodes</code>	set maximum number of aggregators	stripe count of file
<code>romio_cb_read</code> <code>romio_cb_write</code>	enable or disable collective buffering	automatic
<code>romio_no_indep_rw</code>	<ul style="list-style-type: none"> • if <code>true</code>, MPI-IO knows all I/O is collective • Only aggregators will open files 	<code>false</code>
<code>cb_config_list</code>	a list of independent configurations for nodes	N/A
<code>striping_factor</code>	Specifies the number of Lustre stripes	File system
<code>striping_unit</code>	Specifies the size of the Lustre stripe	File system

Other Useful Hints

Hint	Description	Default
<code>romio_lustre_co_ratio</code>	tell MPI-IO the maximum number of processes (clients, here) that will access an OST	1
<code>romio_lustre_coll_threshold</code>	Turns off collective buffering when transfer sizes are above a certain threshold	0 (never)
<code>mpich_mpiio_hints_display</code>	when true a summary of all hints to stderr each time a file is opened	false

IOBUF – I/O Buffering Library

- Optimize I/O performance with minimal effort
 - Asynchronous prefetch
 - Write back caching
 - stdin, stdout, stderr disabled by default
- No code changes needed
 - Load module
 - Recompile & relink the code
- Ideal for sequential read or write operations



IOBUF – I/O Buffering Library

- Globally (dis)enable by (un)setting IOBUF_PARAMS
- Fine grained control
 - Control buffer size, count, synchronicity, prefetch
 - Disable iobuf per file
- Some calls in C, C++ can be enabled using iobuf.h, use the compiler macro, USE_IOBUF_MACROS

export

```
IOBUF_PARAMS='*.in:count=4:size=32M,*.out:count=8:size=64M:preflush=1'
```

IOBUF – MPI-IO Sample Output

IOBUF parameters: file="outc-iob.4":size=1048576:count=4:vbuffer_count=4096:prefetch=1:verbose

PE 0: File "outc-iob.2"

	Calls	Seconds	Megabytes	Megabytes/sec	Avg Size
Open	1	0.000756			
Close	1	0.000318			
Buffers used	1 (1 MB)				

PE 0: File "outc-iob.1"

	Calls	Seconds	Megabytes	Megabytes/sec	Avg Size
Read	1	0.000663	0.065536	98.841390	65536
Open	1	0.000710			
Close	1	0.000361			
Buffer Read	1	0.000445	0.065536	147.308632	65536
I/O Wait	1	0.000474	0.065536	138.268565	
Buffers used	1 (1 MB)				

PE 0: File "outc-iob.3"

	Calls	Seconds	Megabytes	Megabytes/sec	Avg Size
Read	1	0.000694	0.065536	94.427313	65536
Open	1	0.000844			
Close	1	0.000189			
Buffer Read	1	0.000433	0.065536	151.364486	65536
I/O Wait	1	0.000460	0.065536	142.497619	
Buffers used	1 (1 MB)				

IOBUF parameters: file="outc-iob.2":size=1048576:count=4:vbuffer_count=4096:prefetch=1:verbose



HDF5 & PnetCDF

I/O LIBRARIES

Benefits of I/O Libraries

- There are many benefits to using higher level I/O libraries
 - They provide a well-defined, base structure for files that is self-describing and organizes data intuitively
 - Has an API that represents data in a way similar to a simulation
 - Often built on MPI-IO and handle (some) optimization
 - Easy serialization/deserialization of user data structures
 - Portable
- Currently supported: (Parallel) HDF5, (Parallel) netCDF, Adios

I/O Libraries – Some Details

- Parallel netCDF
 - Derived from and compatible with the original “Network Common Data Format”
 - Offers collective I/O on single files
 - Variables are typed, multidimensional, and (with files) may have associated attributes
 - Record variables – “unlimited” dimensions allowed if dimension size is unknown
- Parallel HDF5
 - “Hierarchical Data Format” with data model similar to PnetCDF, and also uses collective I/O calls
 - Can use compression (only in serial I/O mode)
 - Can perform data reordering
 - Very flexible
 - Allows some fine tuning, e.g. enabling buffering

Example Use on Blue Waters

- Under PrgEnv-cray:

```
$> module avail hdf5
```

```
----- /opt/cray/modulefiles -----  
hdf5/1.8.7      hdf5/1.8.8(default)  hdf5-parallel/1.8.7  hdf5-parallel/1.8.8  
(default)
```

```
$> module load hdf5-parallel
```

```
$> cc Dataset.c
```

```
$> qsub -I -lnodes=1:ppn=16 -lwalltime=00:30:00
```

```
$> aprun -n 2 ./a.out
```

```
Application 1293960 resources: utime ~0s, stime ~0s
```

```
$> ls *.h5
```

```
SDS.h5
```

- Dataset.c is a test code from the HDF Group:

<http://www.hdfgroup.org/ftp/HDF5/examples/parallel/Dataset.c>



Darshan

I/O UTILITIES

Example I/O Utility: Darshan

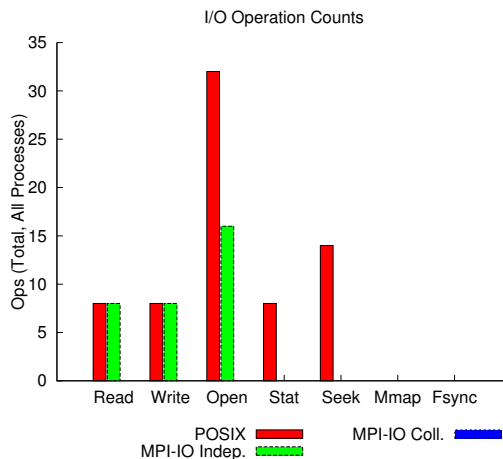
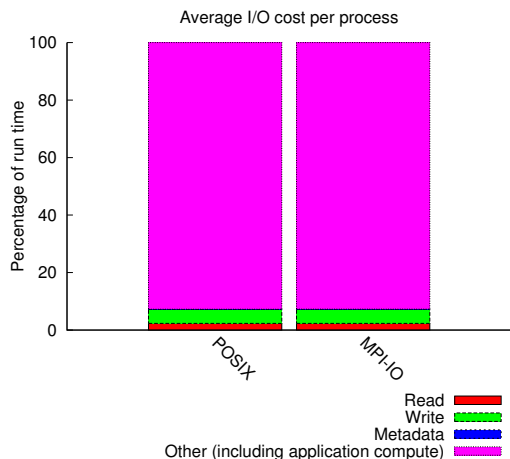
- We will support tools for I/O Characterization
 - Sheds light on the intricacies of an application's I/O
 - Useful for application I/O debugging
 - Pinpointing causes of extremes
 - Analyzing/tuning hardware for optimizations
- Darshan was developed at Argonne, and
- is “a scalable HPC I/O characterization tool... designed to capture an accurate picture of application I/O behavior... with minimum overhead”
- <http://www.mcs.anl.gov/research/projects/darshan/>

Darshan Specifics

- Darshan collects per-process statistics (organized by file)
 - Counts I/O operations, e.g. unaligned and sequential accesses
 - Times for file operations, e.g. opens and writes
 - Accumulates read/write bandwidth info
 - Creates data for simple visual representation
- More
 - Requires no code modification (only re-linking)
 - Small memory footprint
 - Includes a job summary tool

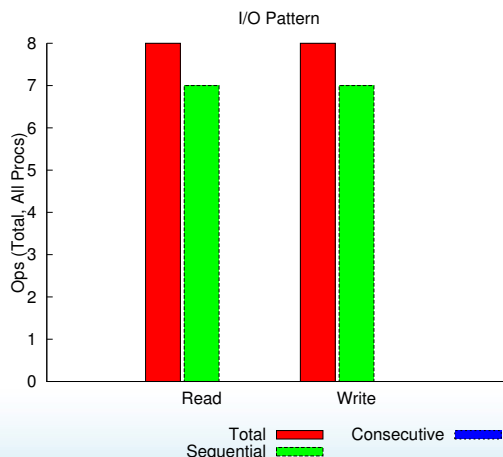
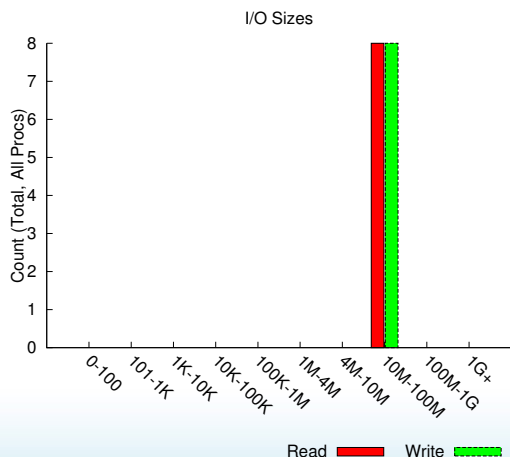
Summary Tool Example Output

jobid: 3406 uid: 1000 nprocs: 8 runtime: 1 seconds



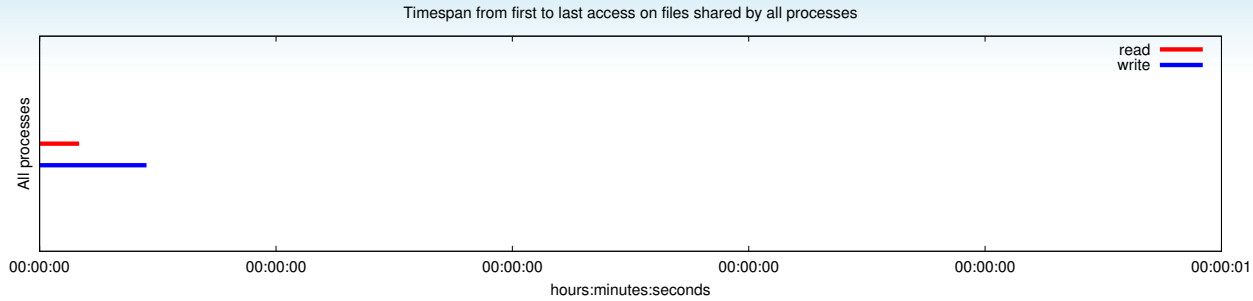
File Count Summary

type	number of files	avg. size	max size
total opened	1	128M	128M
read-only files	0	0	0
write-only files	0	0	0
read/write files	1	128M	128M
created files	0	0	0



Most Common Access Sizes

access size	count
16777216	16



Average I/O per process

	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0.000000	0.000000
Independent writes	0.000000	0.000000
Independent metadata	0.000000	N/A
Shared reads	0.023298	16.000000
Shared writes	0.049300	16.000000
Shared metadata	0.000019	N/A

Data Transfer Per Filesystem

File System	Write		Read	
	MiB	Ratio	MiB	Ratio
/	128.00000	1.00000	128.00000	1.00000

Variance in Shared Files

File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...test.out	8	0	0.041998	32M	2	0.111384	32M	0.0246	0

Two slides left.

THE SUMMARY

Good Practices, Generally

- Opening a file for writing/appending is expensive, so:
 - If possible, open files as *read-only*
 - Avoid large numbers of small writes

```
while (forever) {    open("myfile");  
    write(a_byte); close("myfile"); }
```

- Be gentle with metadata (or suffer its wrath)
 - limit the number of files in a single directory
 - Instead opt for hierarchical directory structure
 - `ls` contacts the metadata server, `ls -l` communicates with every OST assigned to a file (for all files)
 - Avoid wildcards: `rm -rf *`, expanding them is expensive over many files
 - It may even be more efficient to pass metadata through MPI than have all processes hit the MDS (calling `stat`)
 - Avoid updating last access time for *read-only* operations (`NO_ATIME`)

Lessons Learned

- **Avoid unaligned I/O and OST contention!**
- Use large data transfers
 - Don't expect performance with non-contiguous, small data transfers. Use buffering when possible
- Consider using MPI-IO and other I/O libraries
 - Portable data formats vs. unformatted files
- Use system specific hints and optimizations
- Exploit parallelism using striping
 - Focus on stripe alignment, avoiding lock contention
- Move away from one-file-per-process model
 - Use aggregation and reduce number of output files
- Talk to your POC about profiling and optimizing I/O

The End

Thanks to: Victor Anisimov, Galen Arnold, Kalyana Chadalavada, Tom Cortese, Manisha Gajbe, Rob Sisneros