Collective IO for Petascale Programming

Zhao Zhang, Allan Espinosa, Kamil Iskra, Ioan Raicu,
Ian Foster, Michael Wilde

Presented by M. Wilde, wilde@mcs.anl.gov

Computation Institute
University of Chicago
and Argonne National Laboratory

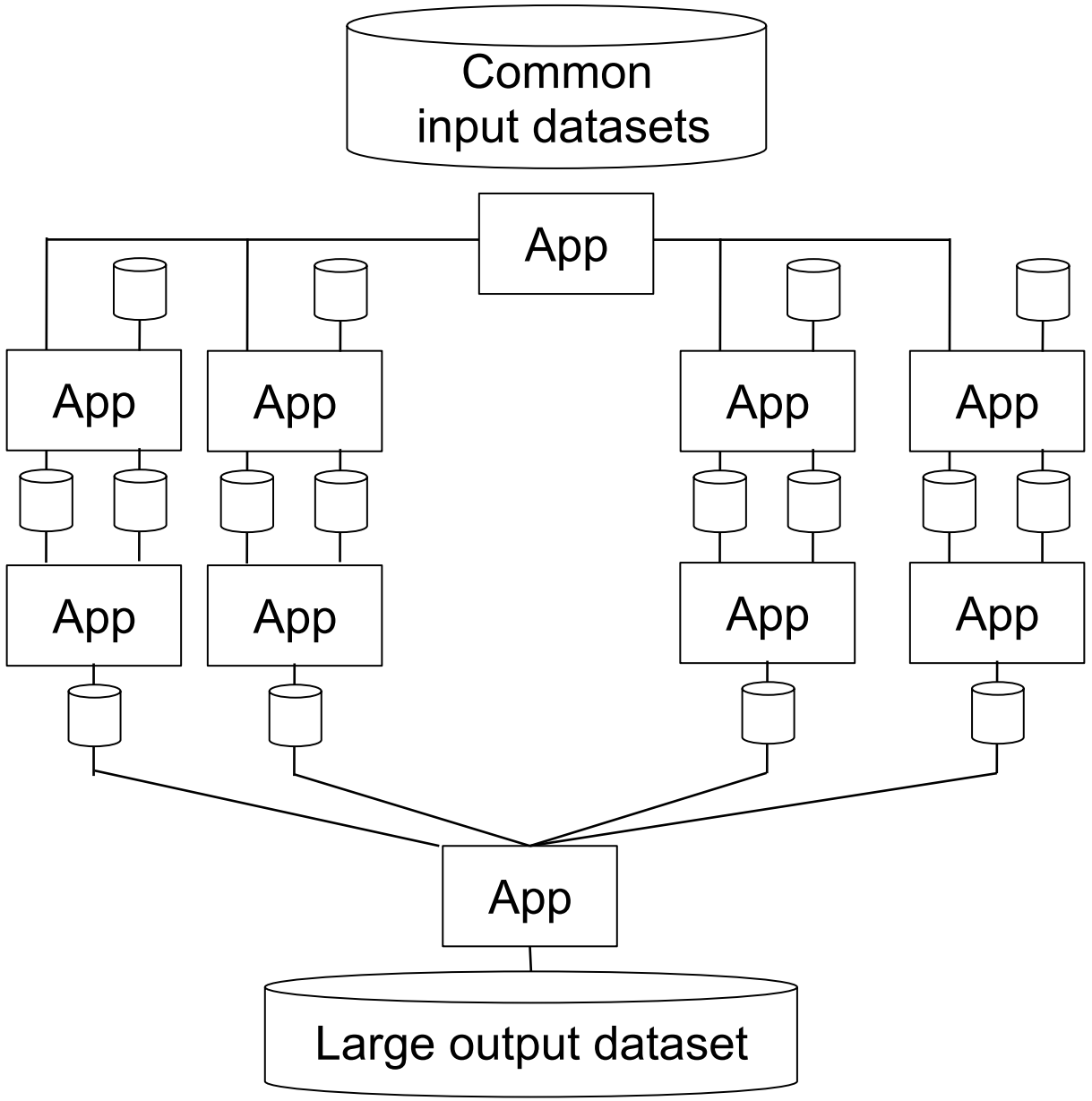
www.ci.uchicago.edu

Computing Models

- Tightly coupled
  - Shared memory, SMP >> OpenMP
  - Distributed memory, message passing >> MPI
  - Takes significant programming effort; sometimes obtained from parallel libraries
- Loosely coupled
  - Vast array of existing applications to reuse
  - Re-coupling apps to do new things – **via scripts**
  - Scripting is powerful:

    Enable scripting for petascale systems

Loosely coupled application dataflow patterns



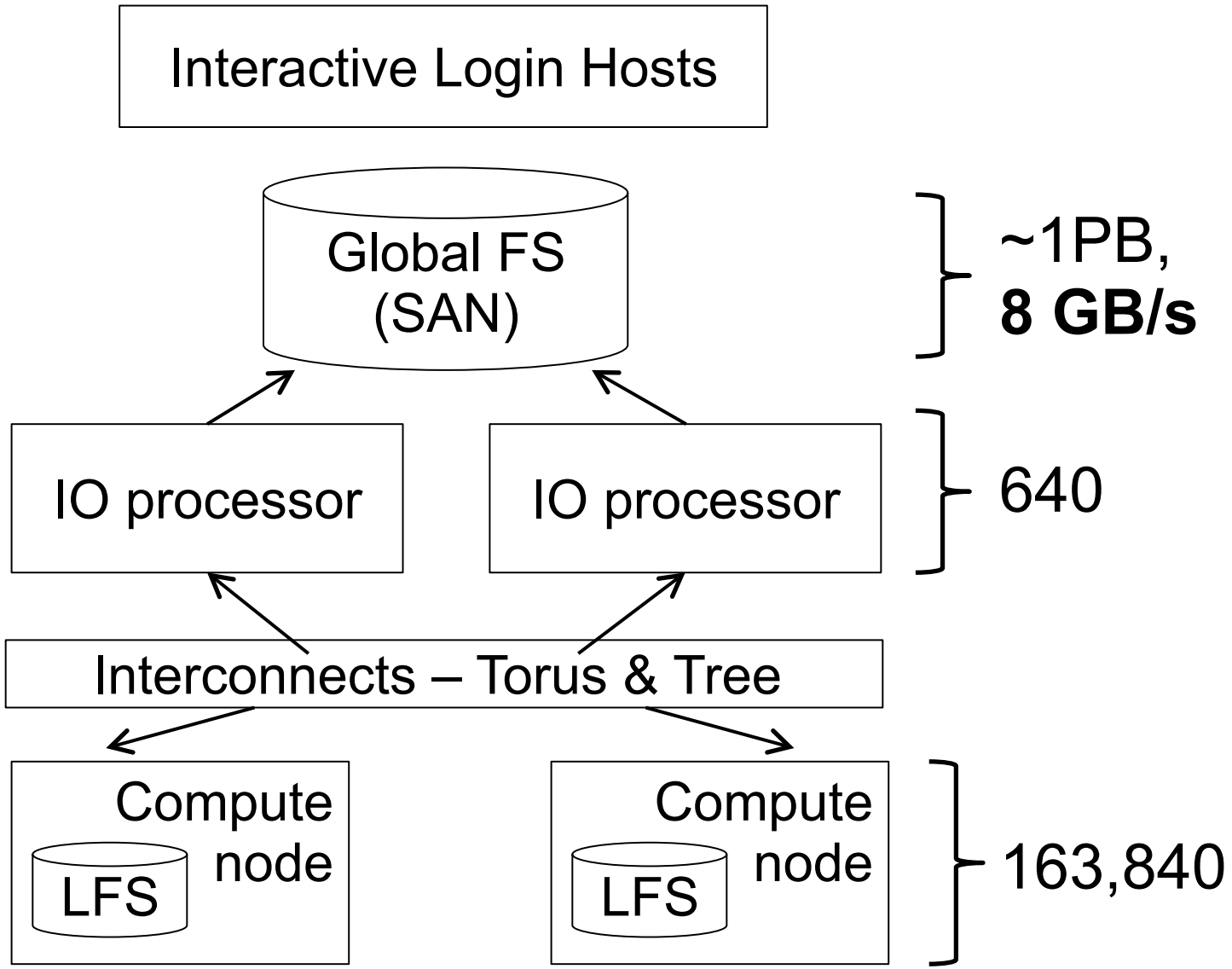
Why loose coupling?

- Applications as functions – create powerful capabilities by linking in new patterns
- All the benefits of scripting amplified by petascale resource levels
- Simple to reuse a lot of existing functionality
- Can spread load across petascale machines
- Can use machines specifically tailored to specific applications (e.g. visualization)
- Can do loose coupling of TC apps

Motivation

- Easier to adapt workload to changing processor availability
- Utilize *provisioned resources* for repeated simulation and analysis
  - Diverse tasks in collaborative sessions
- Great fault tolerance than tight coupling
- Don’t couple tightly when algorithm or performance needs don’t demand it
  - Algorithm should drive coupling mode

Typical target: ALCF BG/P “Intrepid”



Intrepid GPFS Global File Systems

- DataDirect 9550 SAN (4)
  - 1.1 PB Raw Disk (combined)
  - 320 500GB SATA HDD (each)
- IBM x3655 File Servers (24)
  - 12 GB RAM
  - 2.6 GHz Dual Core CPU (2)
  - Myri10G NIC (2)
  - 4X SDR InfiniBand NIC (1)
  - **7.5 GB/sec read, 8.1 GB/sec write**

https://wiki.alcf.anl.gov/index.php/Filesystem_Info,
as of Jul 2008; significant expansion in progress)

FS Name	Servers	Write GB/s	Read GB/s
gpfs1	16	5.6	5.3
home	8	2.5	2.3

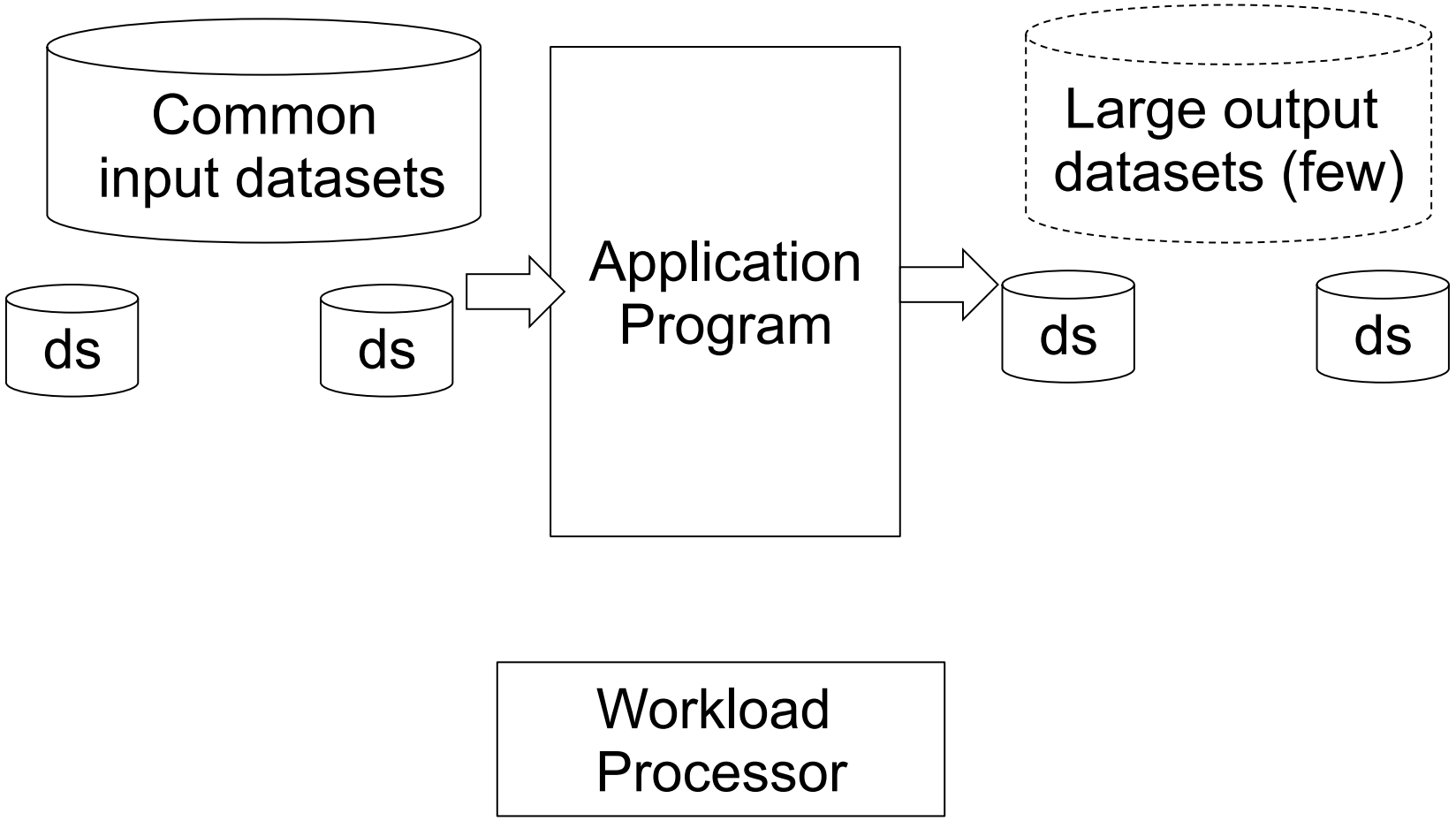
Costs of Loose Coupling

- Loose coupling is easy with a global file system, BUT:
  - Object management – create, delete is expensive
  - Locking and bandwidth limits cause contention
  - Application IO block sizes may be poor – stress on a distributed IO subsystem

Costs of Loose Coupling

- Even with local filesystems there are hard issues:
  - Local file space increasingly limited on petascale systems
  - Local space not persistent (nodes booted for jobs)
  - Access issues: data may not be where needed, moving may be costly

Application IO Patterns



Application Profiles

- MARS – petroleum refining econ model
  - KB files in, KB files out
- OOPS – protein folding
  - KB + 50MB common in, MB files out
- DOCK – protein-ligand docking
  - KB + 50MB in, MB files out
- BLAST – sequence alignment/search
  - KB + 6GB common in; KB files out
- All need multi-stage analysis/reduction

The Collective IO Model

- Provide fast pools of intermediate storage
- Use local storage wherever possible, and stage in and out
- Broadcast input
- Batch and gather output

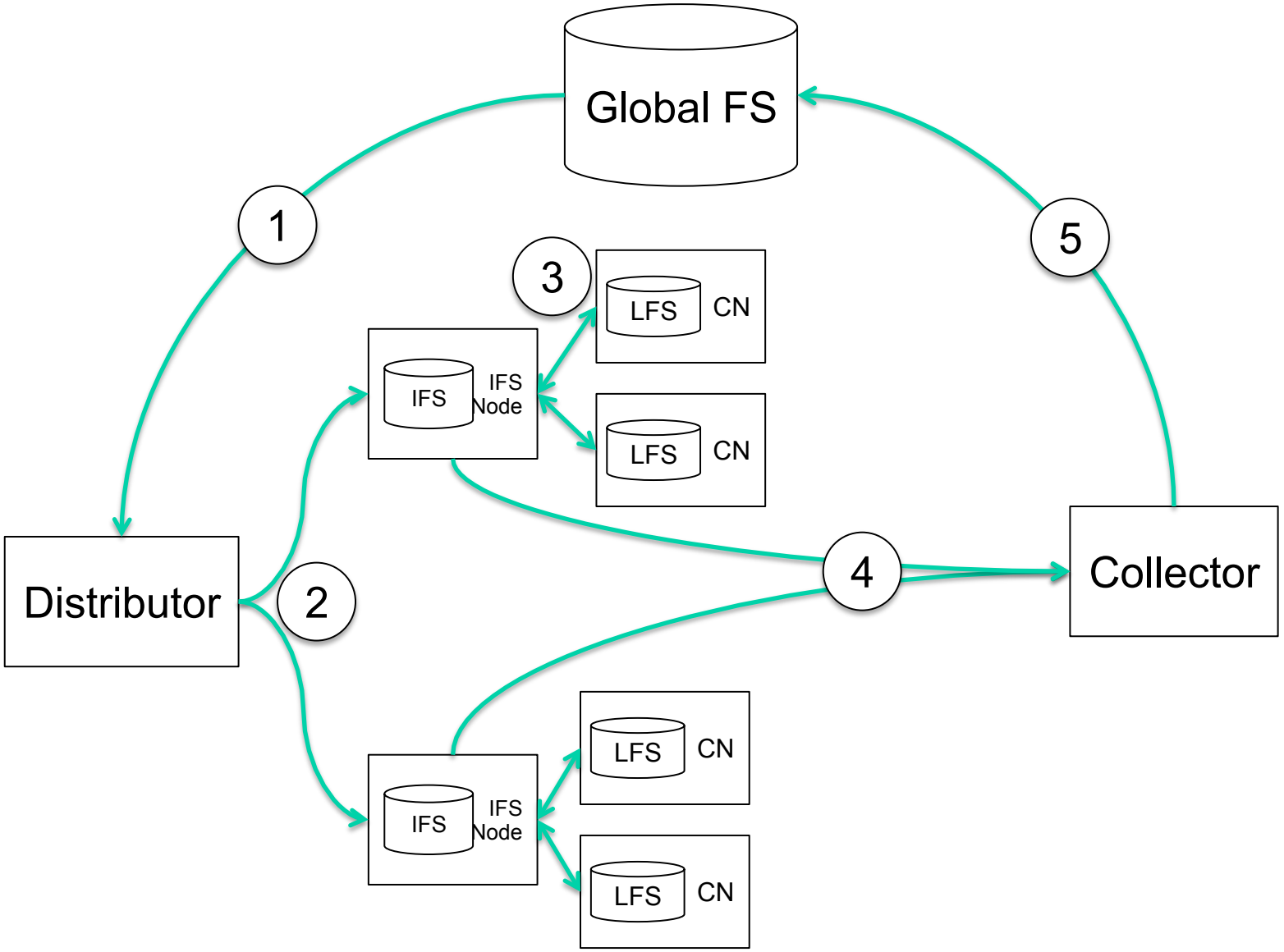
Input processing

- Small datasets staged from GFS to LFS of CN that will read them
- Larger datasets placed on intermediate file system (striped for capacity and speed)
- Datasets read by multiple tasks distributed by “broadcast”

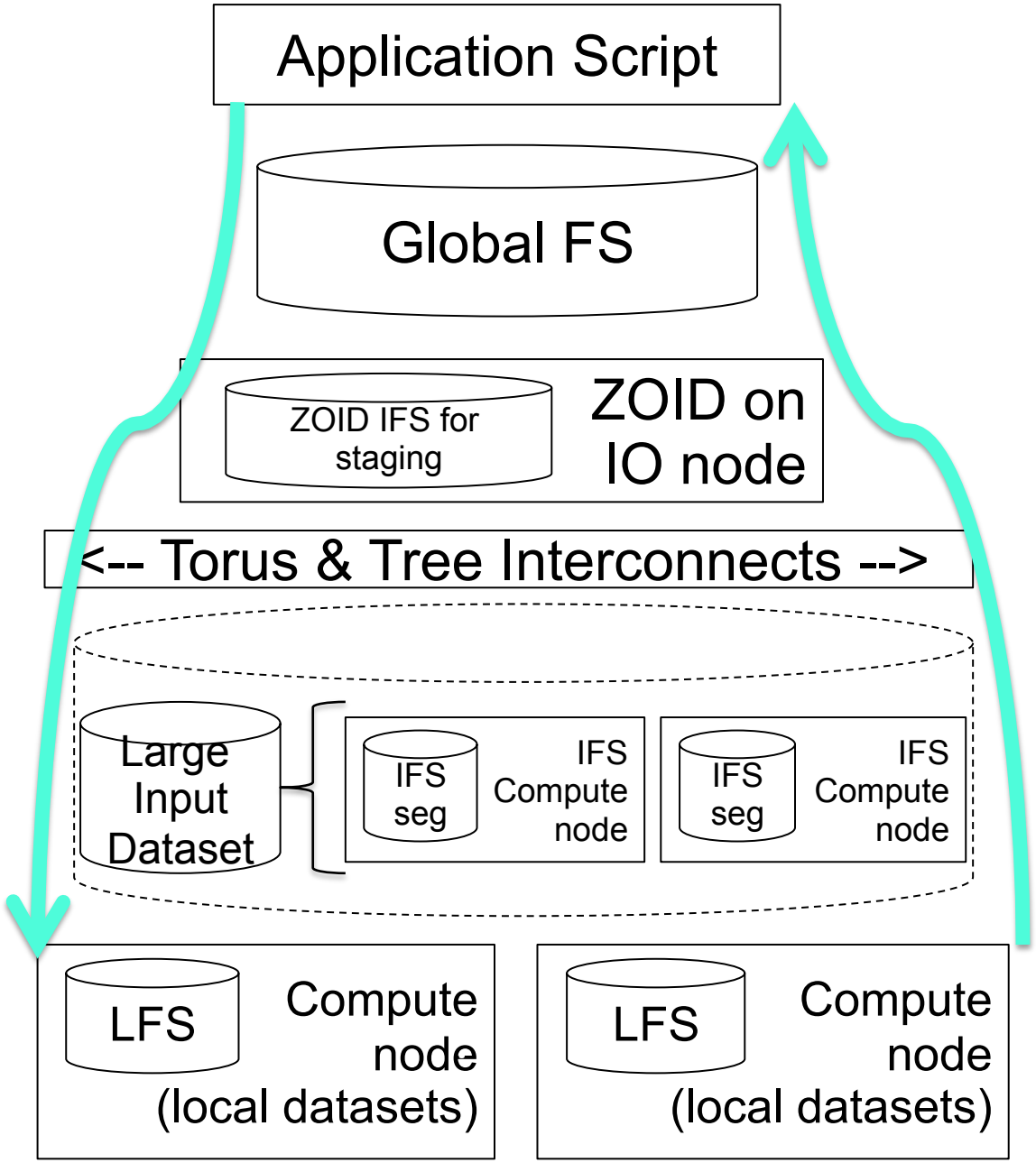
Output processing

- Gather small output files periodically from multiple CNs
- Aggregate into larger files for efficient staging to GFS
- Stage data asynchronously to let tasks finish quickly
- Use LFS or IFS for staging, as needed

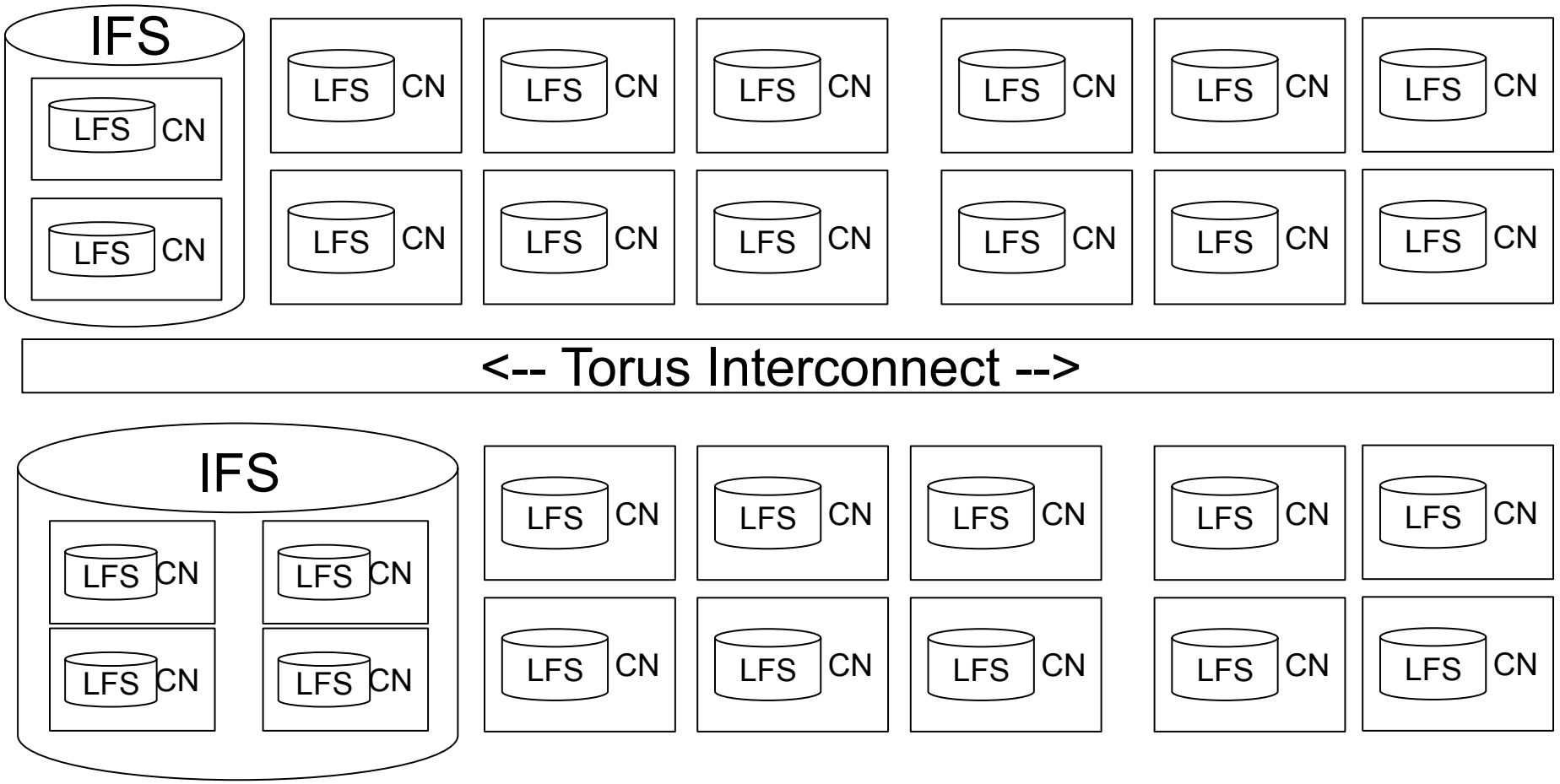
Distributor and collector



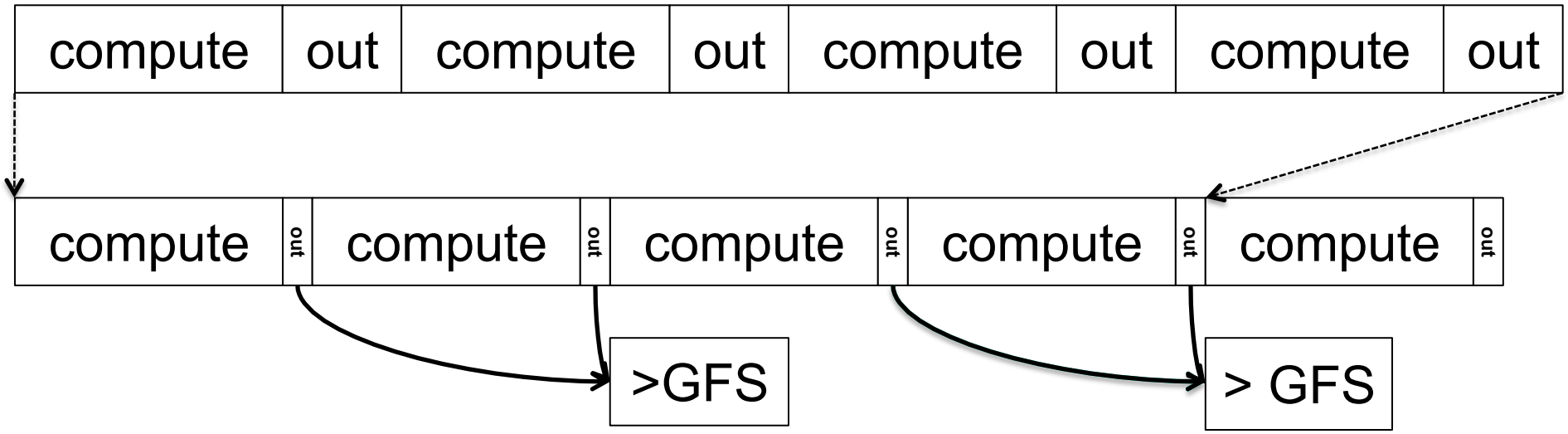
LCP Collective IO Model



Mapping Compute Nodes to IFS's



Asynchronous Output Staging



Simple prototype implementation

- Performance sanity check, few automated heuristics
- Simple scripts the hard-code much of the eventual logic
- Using MosaStore, Chirp and FUSE for file system mechanisms
- Goal when mature is to integrate into parallel scripting and workflow systems like Swift
- Several things not yet implemented

Reading from IFS

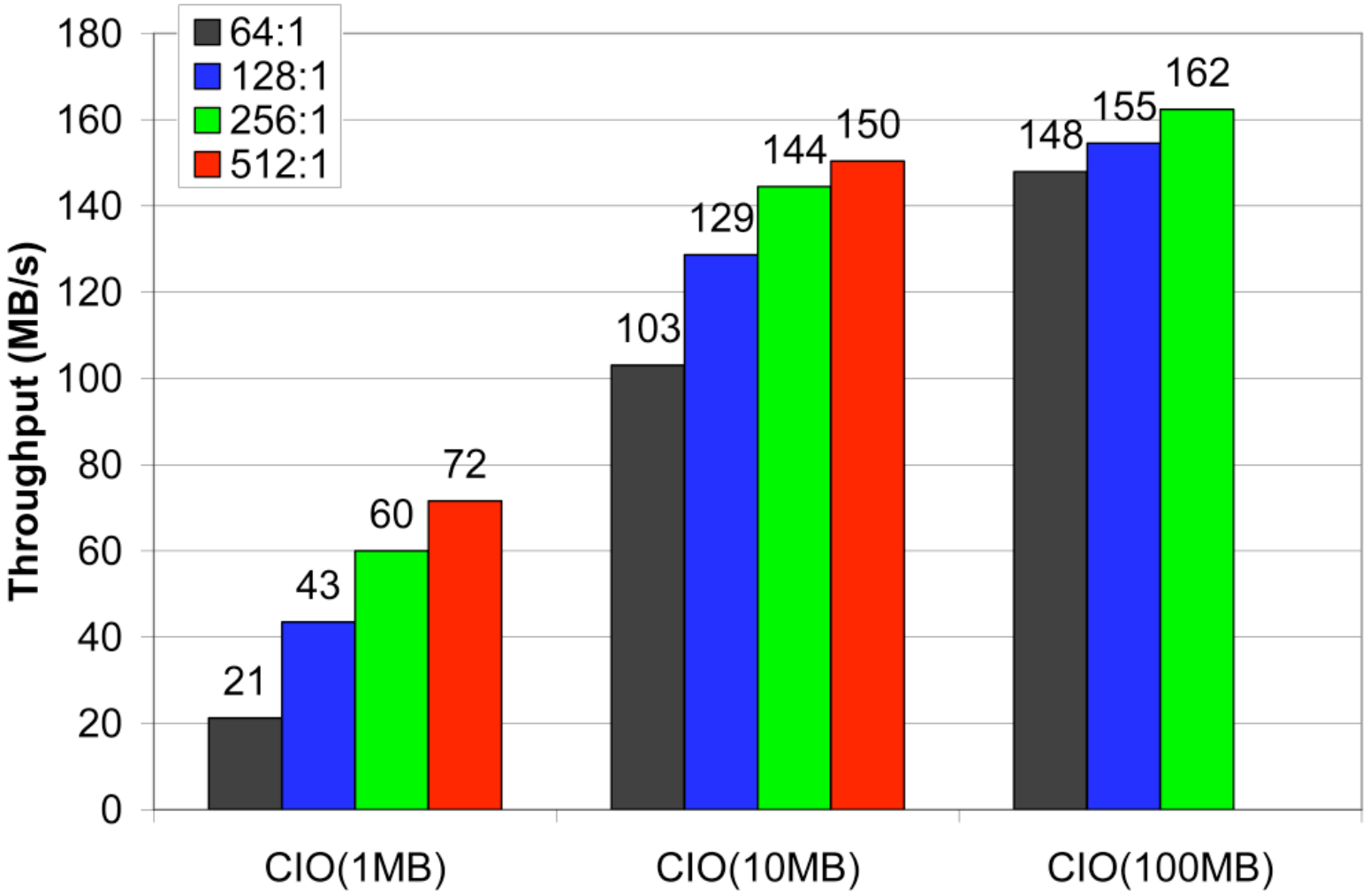


Figure 11: Read performance while varying the ratio of LFS to IFS from 64:1 to 512:1 using the Torus network.

Read performance from IFS

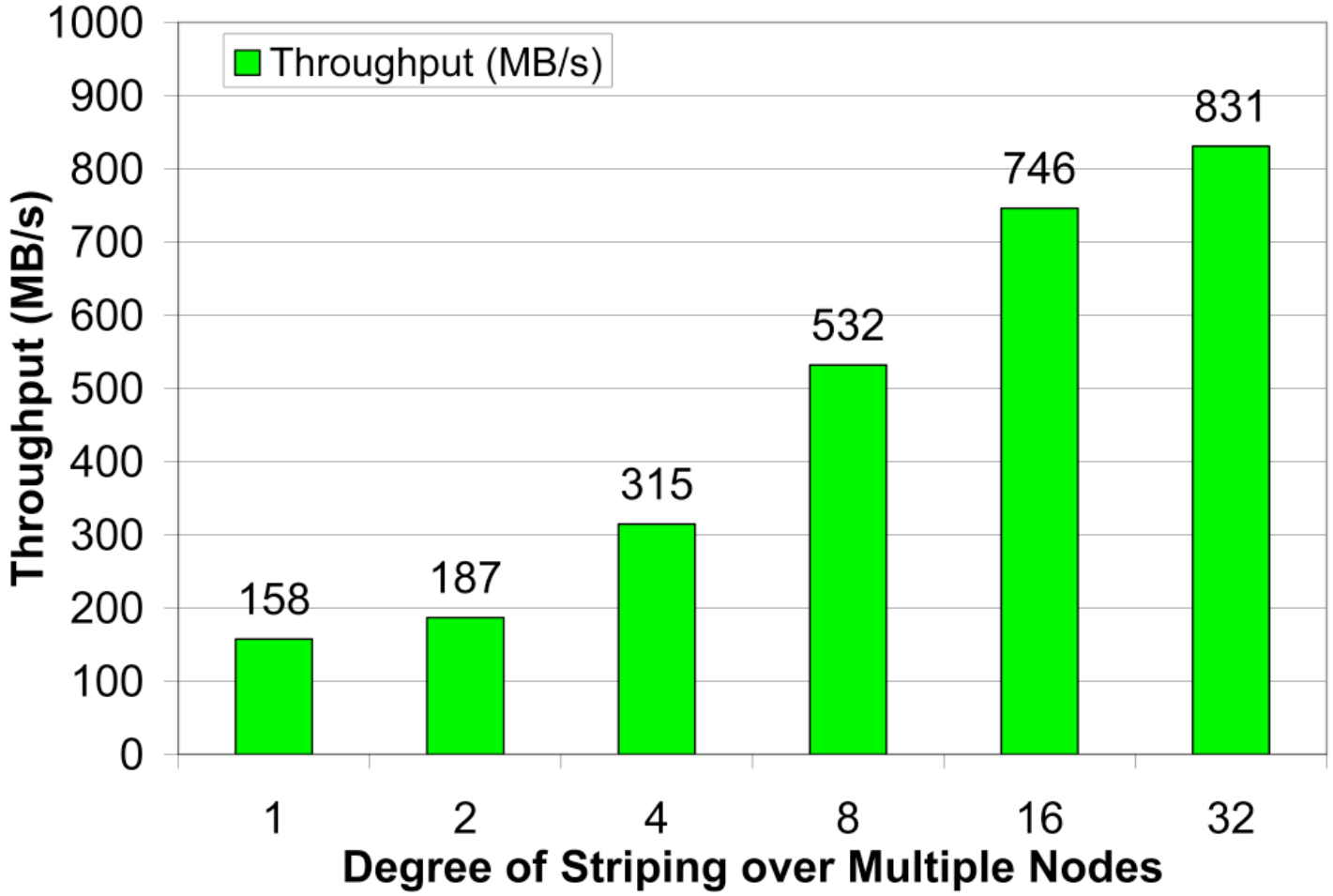
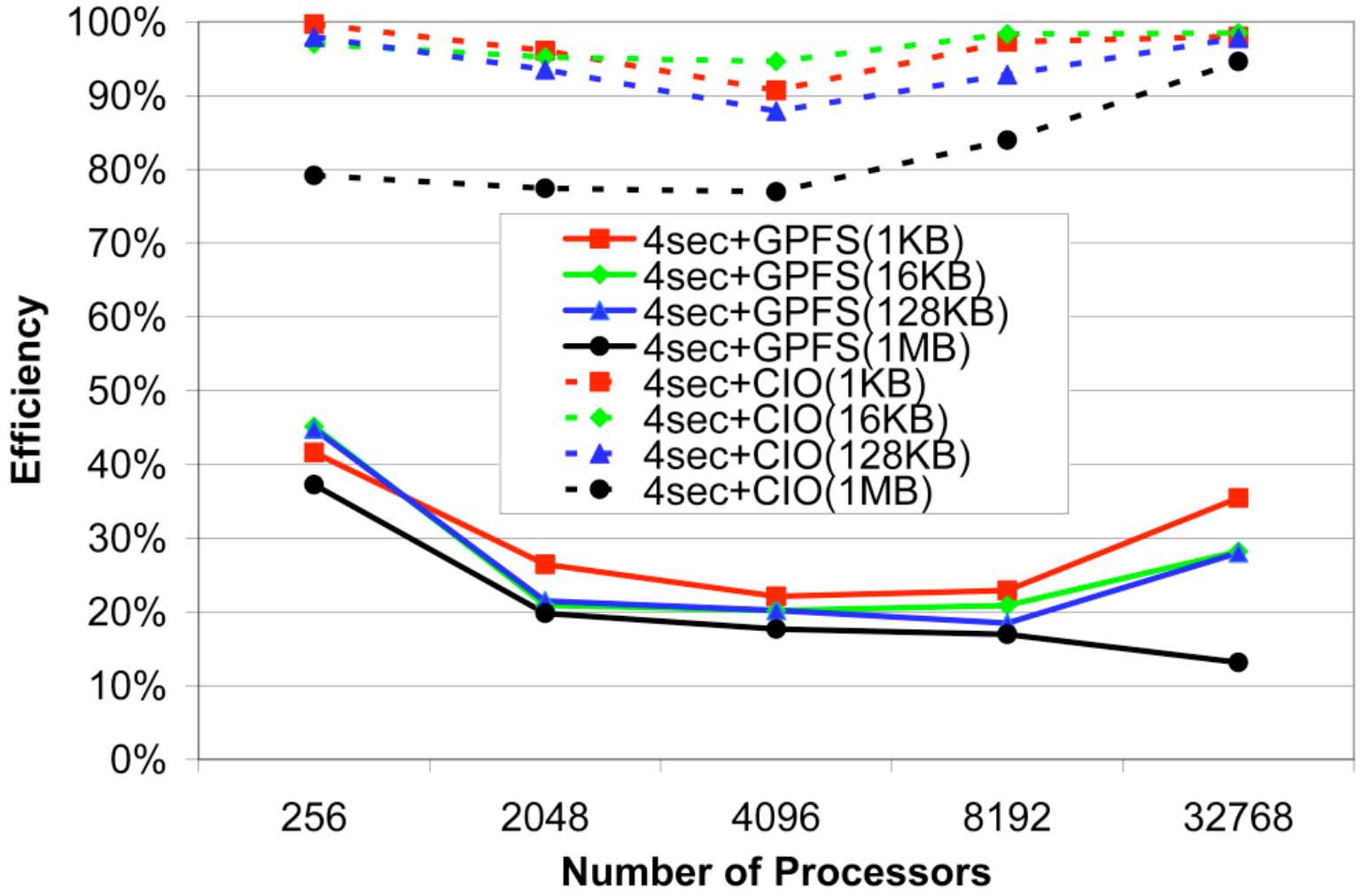


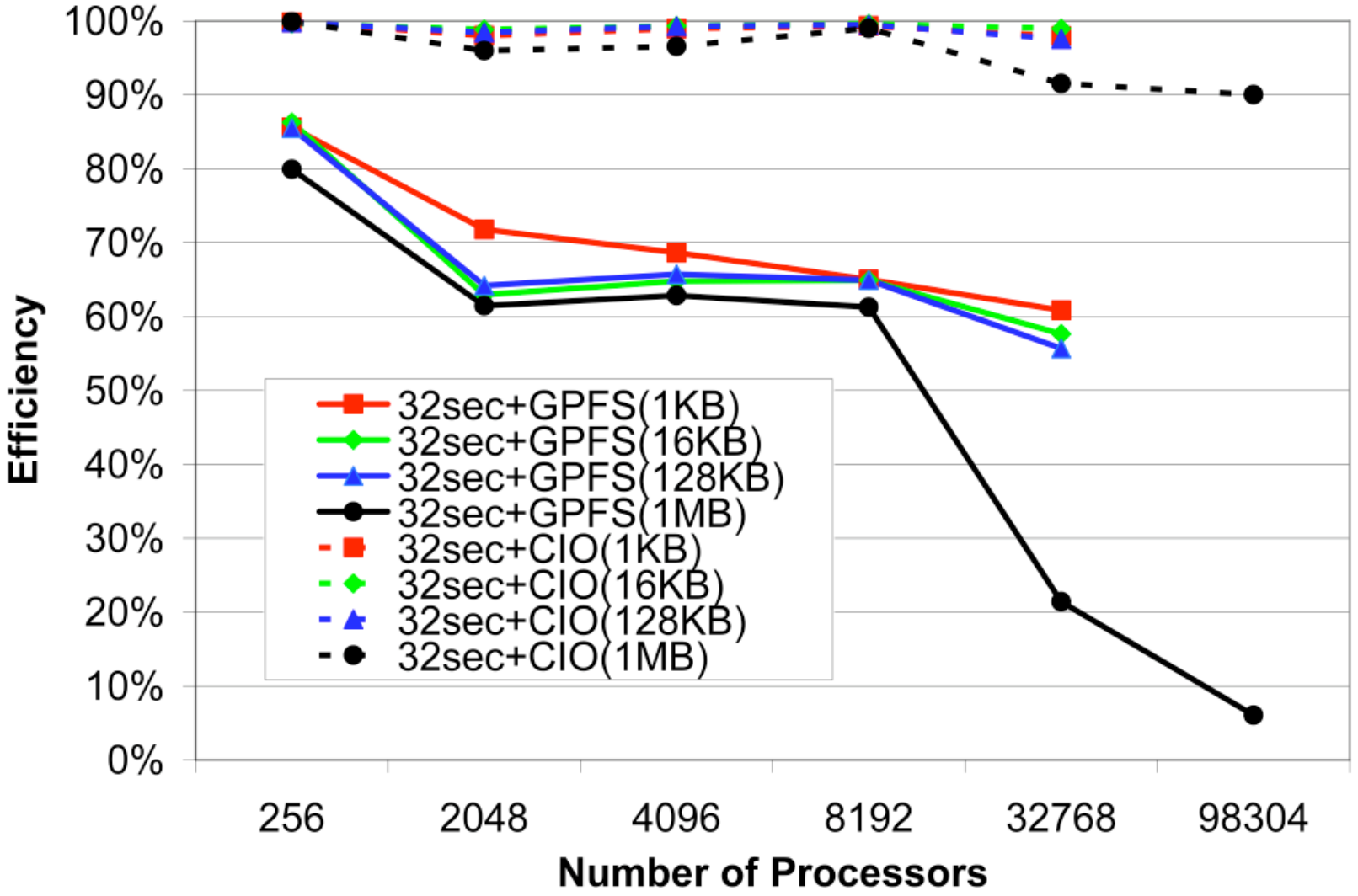
Figure 12: Read performance, varying the degree of striping of data across multiple nodes from 1 to 32 using the torus network

CIO vs. GFS output efficiency



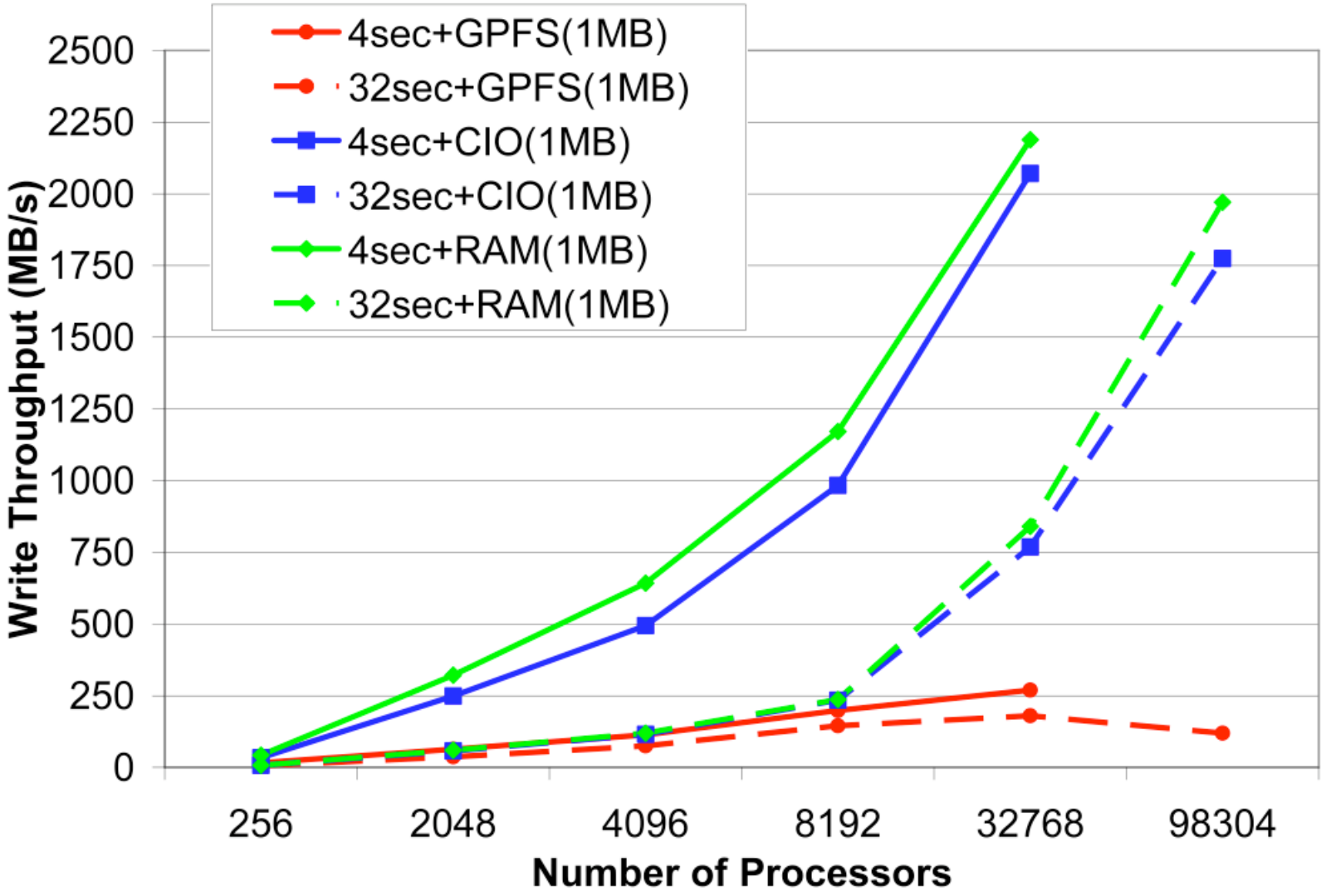
- **Figure 14: CIO vs. GFS efficiency for 4 second tasks, varying data size (1KB to 1MB) on 256 to 32K processors**

CIO vs. GFS output efficiency



- **Figure 15: CIO vs GPFS efficiency for 32 second tasks, varying data size (1KB to 1MB) for 256 to 96K processors.**

CIO collection write performance



- **Figure 16: CIO collection write performance compared to RAM and GPFS write performance on up to 96K processors**

Multi-stage DOCK6 workload

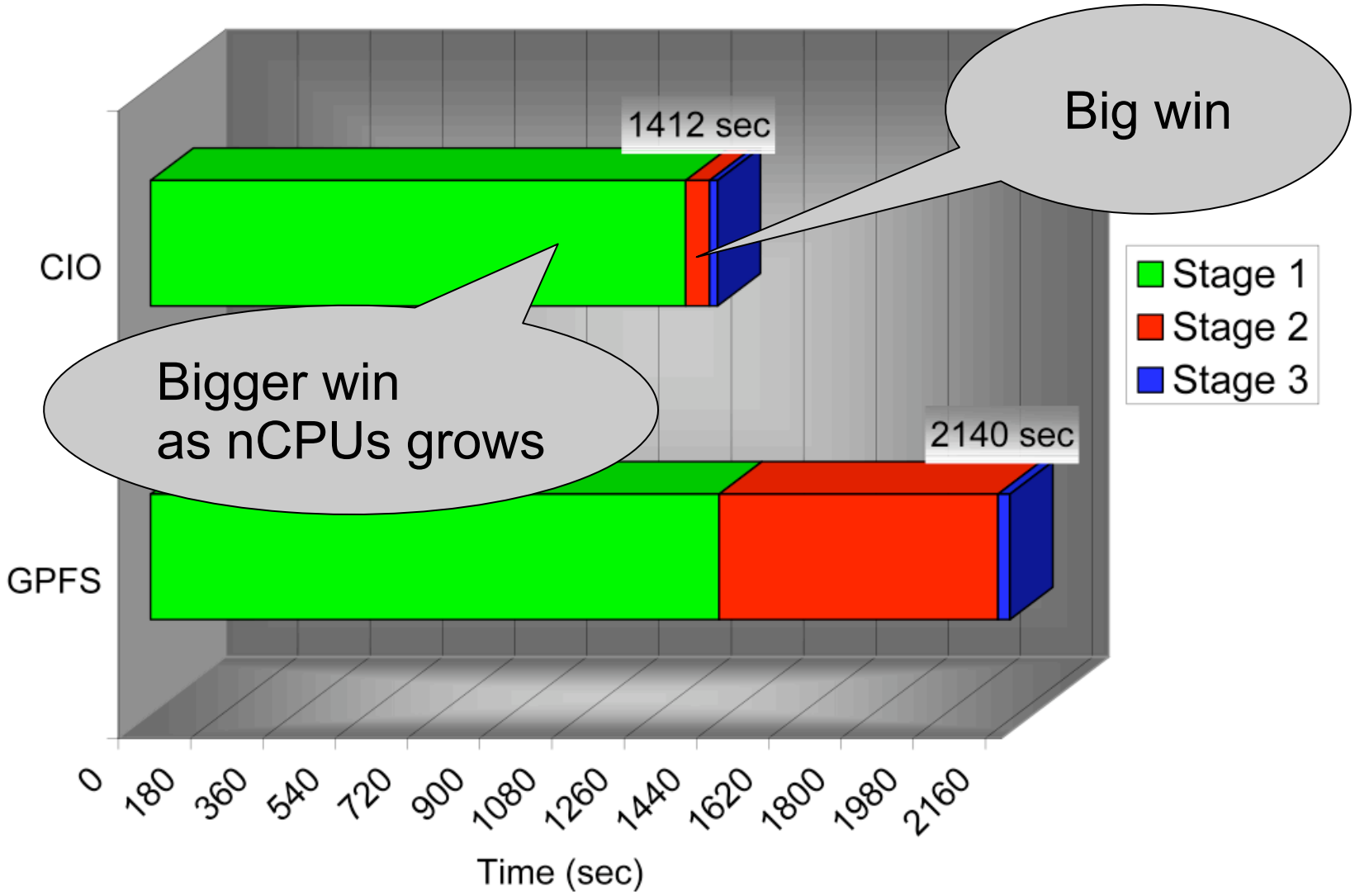


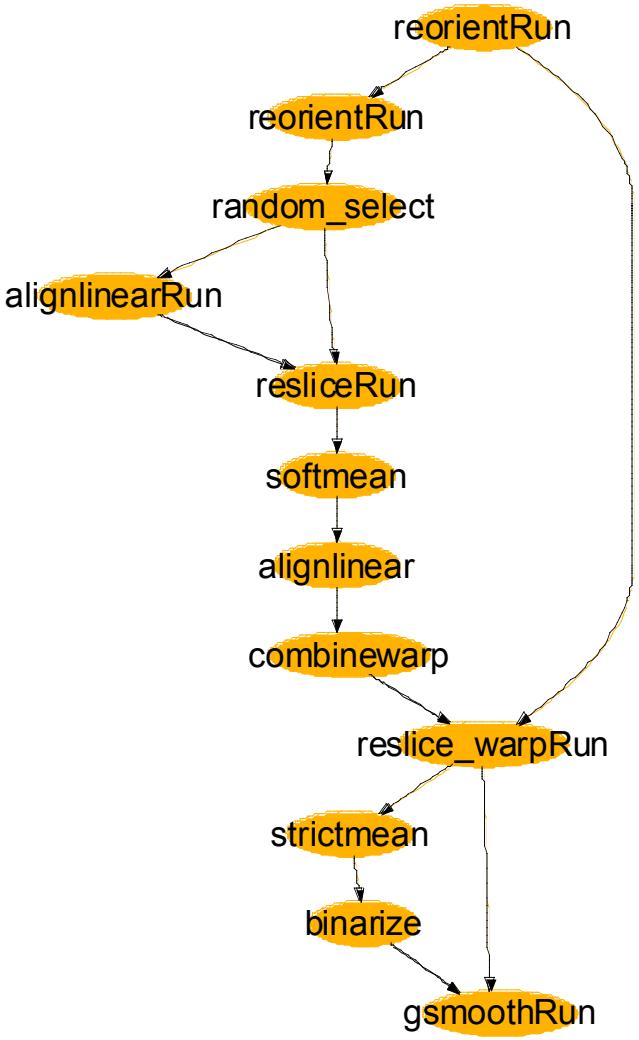
Figure 17: DOCK6 application summary with 15K tasks on 8K processor comparing CIO with GPFS

Next Steps

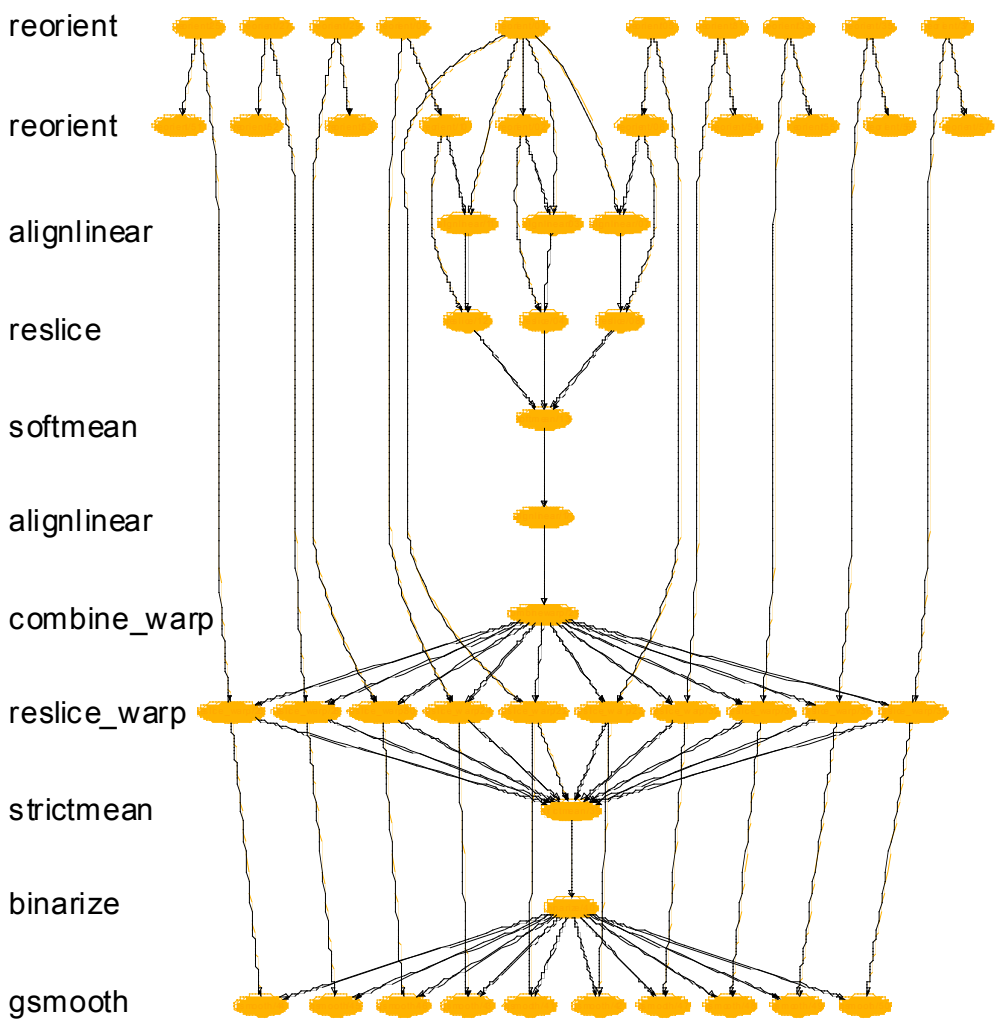
- Integration
  - Automatically do CIO within Swift workflows
- Algorithms
  - Optimal ration of IFS nodes to CNs (wf dep?)
  - Use IFS CN's to compute?
  - Optimal data placment: LFS vs. IFS vs. GFS
  - Learn from prior runs of a workflow or app
  - Automate caching for downstream processing
  - lifetime management of cached data
- Implementation
  - MPI for broadcast; xar; choice of tech.
- Application and measurement
  - BLAST with large databases; many others

Automated image registration for spatial normalization

AIRSN workflow:



AIRSN workflow expanded:



Collaboration with James Dobson, Dartmouth [SIGMOD Record Sep05]

AIRSN Program Definition

```
(Run snr) functional ( Run r, NormAnat a,
                       Air shrink ) {
    Run yroRun = reorientRun( r , "y" );
    Run roRun = reorientRun( yroRun , "x" );
    Volume std = roRun[0];
    Run rndr = random_select( roRun, 0.1 );
    AirVector rndAirVec = align_linearRun( rndr, std, 12, 1000, 1000, "81 3 3" );
    Run reslicedRndr = resliceRun( rndr, rndAirVec, "o", "k" );
    Volume meanRand = softmean( reslicedRndr, "y", "null" );
    Air mnQAAir = alignlinear( a.nHires, meanRand, 6, 1000, 4, "81 3 3" );
    Warp boldNormWarp = combinewarp( shrink, a.aWarp, mnQAAir );
    Run nr = reslice_warp_run( boldNormWarp, roRun );
    Volume meanAll = strictmean( nr, "y", "null" )
    Volume boldMask = binarize( meanAll, "y" );
    snr = gsmoothRun( nr, boldMask, "6 6 6" );
}
```

```
(Run or) reorientRun (Run ir,
                      string direction) {
    foreach Volume iv, i in ir.v {
        or.v[i] = reorient(iv, direction);
    }
}
```

Conclusion

- Loosely-coupled programming offers great scientific benefit on petascale systems but challenges the IO subsystems of such machines
- Collective IO operations can make loosely coupled programming practical and efficient
- Much work remains to make it fast *and* transparent
- File and message passing are similar

Acknowledgements

- NSF Grant OCI-0721939
  NASA Ames GSRP NNA06CB89H
  US DOE Contract DE-AC02-06CH11357 Argonne LDRD Program Argonne LDRD
- Chirp (Notre Dame, D. Thain et.al) and MosaStore (UBC, Samer Al-Kiswany, Matei Ripeanu et.al)
- Samer Al-Kiswany (UBC), Kazutomo Yoshii (Argonne), Argonne Leadership Computing Facility team (BG/P access and support), Mike Kubal, UChicago for the DOCK application)

For more info…

Swift parallel scripting system

www.ci.uchicago.edu/swift

Falkon lightweight task scheduler

www.ci.uchicago.edu/falkon

ZeptoOS Compute Node Kernel

www.zeptoos.org