

Parallel I/O Algorithms

Outline

Cost of Parallel I/O

The Naive Algorithm

Some Ad-hoc Optimizations

Collective I/O Algorithms

- Access Scheduling: Grouping & Disk-directed I/O
- Collective Buffering

Summary



Cost of Parallel I/O

Key to parallel I/O performance: Parallelism

Cost breakdown:

- Disk access time (# of disk accesses, seek time)
- Message transmission time (# of msgs, total bytes)
- Resource contention effects (# of active agents)

Factors affecting performance:

- Application behavior
- System and filesystem architecture

The key factor is data access size



Experimental Method

Parameters Varied

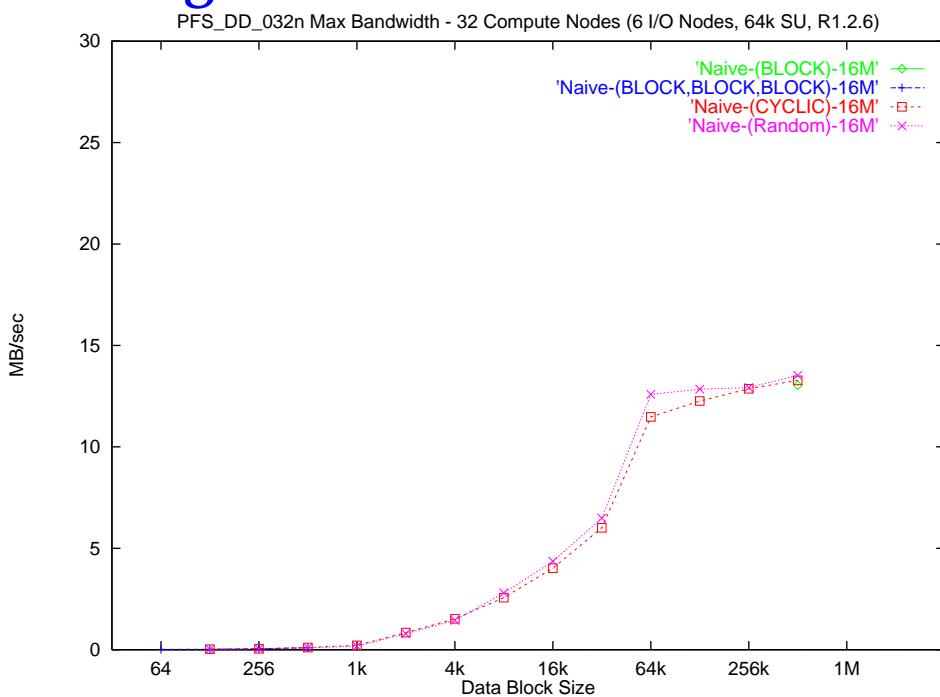
- Number of compute nodes / compute:I/O ratio
- Data distribution (block, cyclic, 3D block, random)
- File access size (128 bytes to 1 MB)
- File size (small 16 MB and large 128-192 MB)
- System architecture (Paragon PFS and SP2 PIOFS)

Results reported

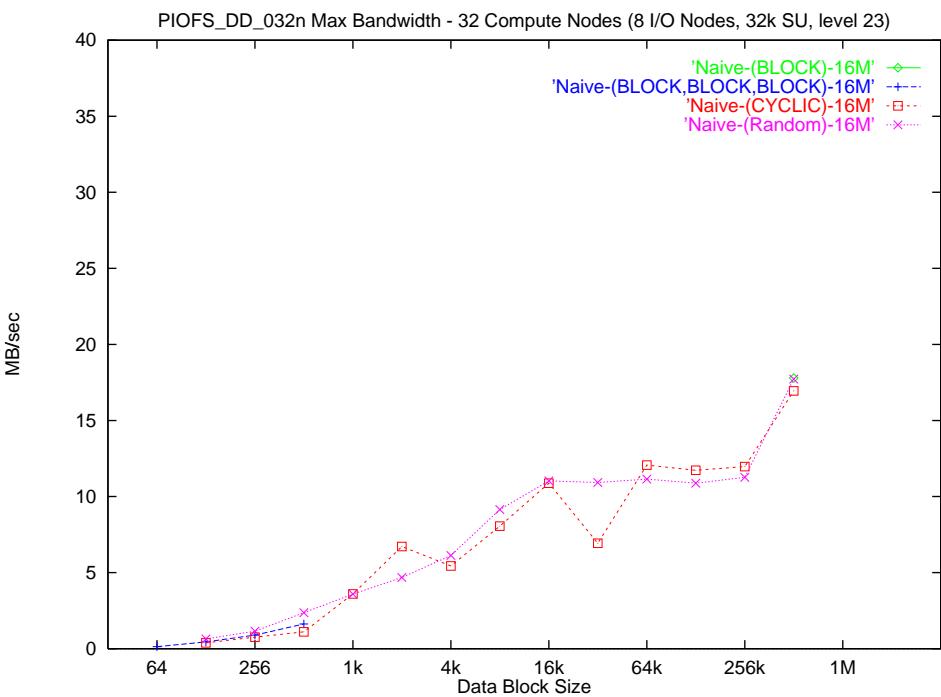
- Best of multiple (3-50) runs
- Averages were noisy (20% error/90% confidence)



Paragon PFS Write Performance



SP2 PIOFS Write Performance



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 5

Performance Factors

Data access size (data block size) is the single most important factor affecting performance.

Small data accesses are slow not only for multi-dimensional distributions but for others as well (CYCLIC and Random)

Data distribution yields at most a 2x performance difference (at the low end), but data access size yields orders-of-magnitude differences.



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 6

The “Naive” Algorithm

Our name for the simple approach

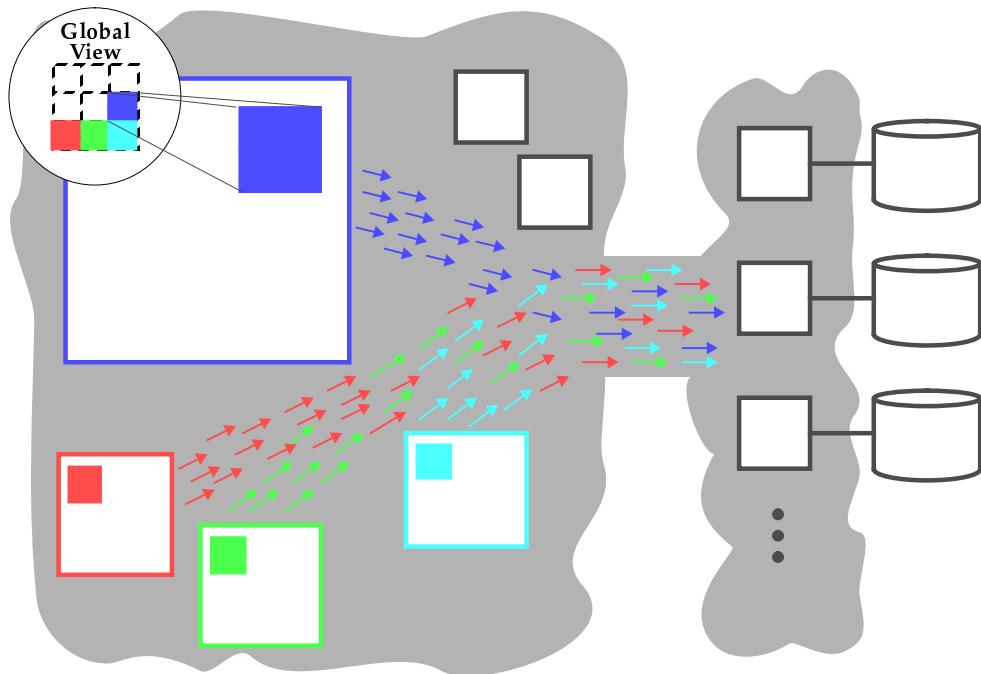
- Straightforward implementation of file I/O
- Uses existing interfaces (e.g. UNIX)

Compute node pseudocode:

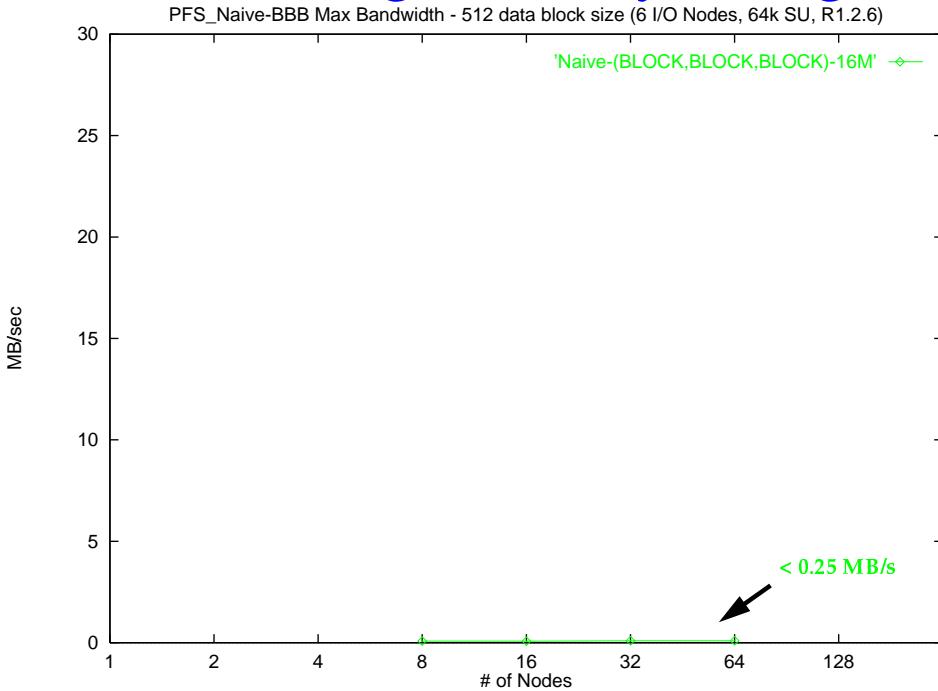
```
for each data block (db) in local data chunk:  
    compute offset (o) of db in the file  
    write db at offset o
```



Naive Algorithm



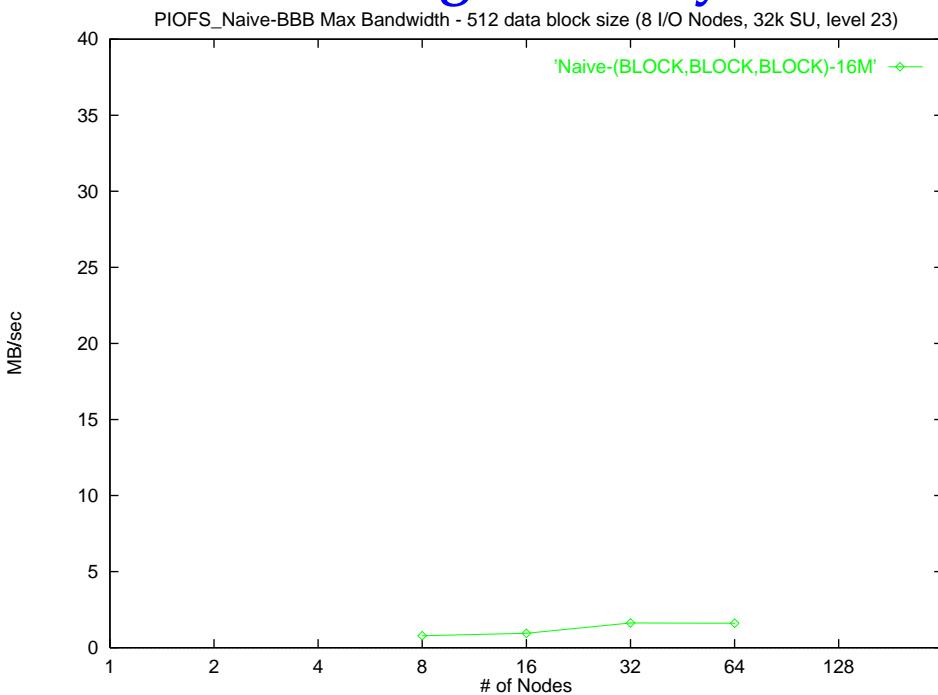
Naive: Writing 3D Array (Paragon)



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 9

Naive: Writing 3D Array (SP2)



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 10

Naive Algorithm Summary

Advantages

- Simple to implement
- Highly portable

Disadvantages

- Abysmal performance for small data blocks (e.g. multi-dimensional distributions)
- May cause thrashing
- Cumbersome offset calculations



Ad-hoc Optimizations

Don't do I/O

Every node writes to a separate file, then post-process

- Postprocessing may require computing resources commensurate with writing the canonical format
- Administrative nightmare (100s of files)

One node does all I/O

- serialization point, but not bad
- a special case of "collective buffering"

Hand-code exotic optimizations

Nothing is portable!



Collective I/O

iPSC/860 CFS Study [Nitzberg '92]:

- Small accesses are particularly slow
- Obtaining maximum performance requires significant programmer effort

Scientific applications are already synchronized:

```
Initialize
Repeat
    Compute
    Communicate
    Perform I/O
End
```

Motivated applying the **collective approach to parallel I/O**



Collective I/O

Collective I/O is coordinated (synchronized) I/O which uses global knowledge for intelligent data transfer.

Coordination provides global knowledge of

- data distribution
- file layout
- machine architecture

Access to the global data structure is logically performed as a single operation rather than many individual operations.

- exploits inherent synchronization at I/O points
- significant potential for optimizations



Collective Algorithms

Cost of I/O on a Parallel System

- Disk access time (# of disk accesses, seek time)
- Message transmission time (# of msgs, total bytes)
- Resource contention effects (# of active agents)

Access Scheduling

- Grouping (R)
- Disk-directed I/O (D)

Collective Buffering

- BLOCK, File Layout, CYCLIC (D)
- Scatter / Gather (M)



Grouping

**Eliminate thrashing caused by resource contention effects
(due to OS bugs, hardware anomalies, other “features”)**

- node grouping — limit control parallelism
- data grouping — limit data parallelism

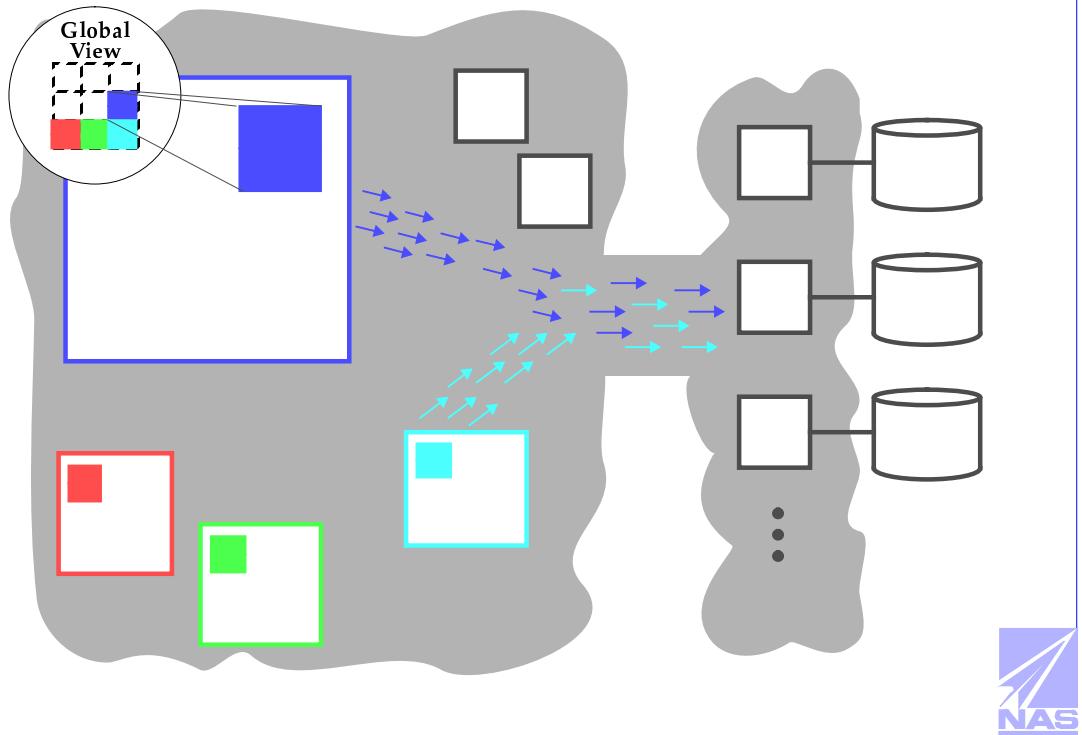
Indicated by negative scaling in performance

Node grouping pseudocode (N nodes):

```
k = process_rank();
if (k >= GroupSize)
    Wait for GO signal;
Perform I/O Operation;
if (k < N - GroupSize)
    Send GO signal to process (k + GroupSize);
```



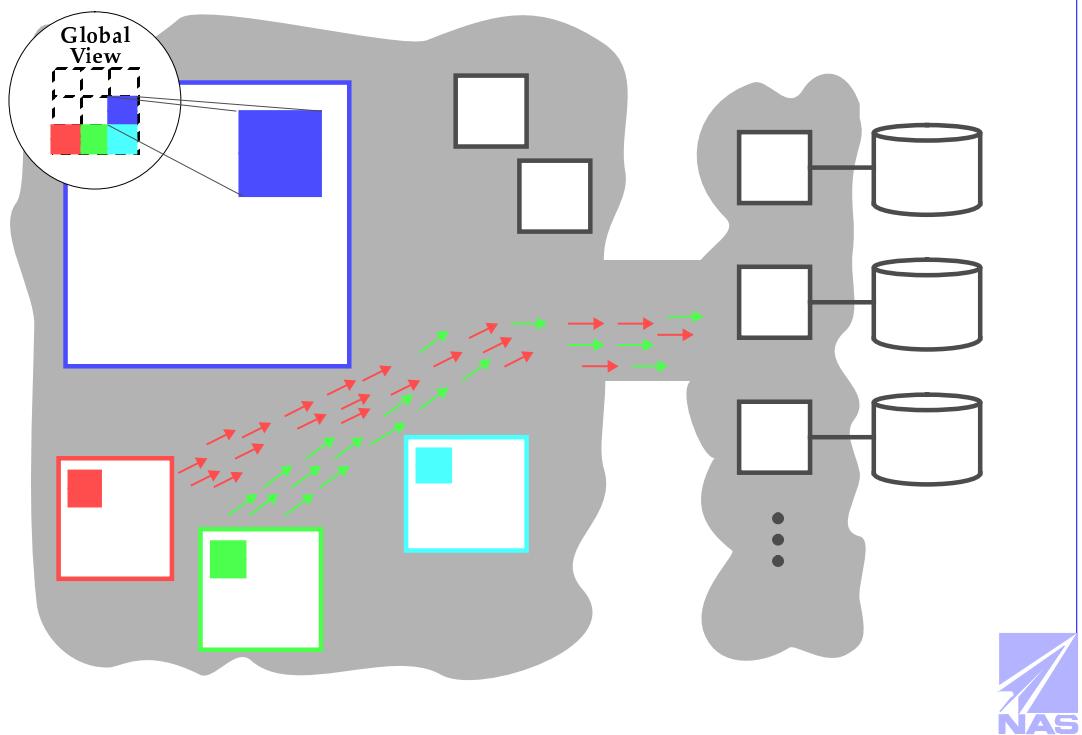
Grouping—First set



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 17

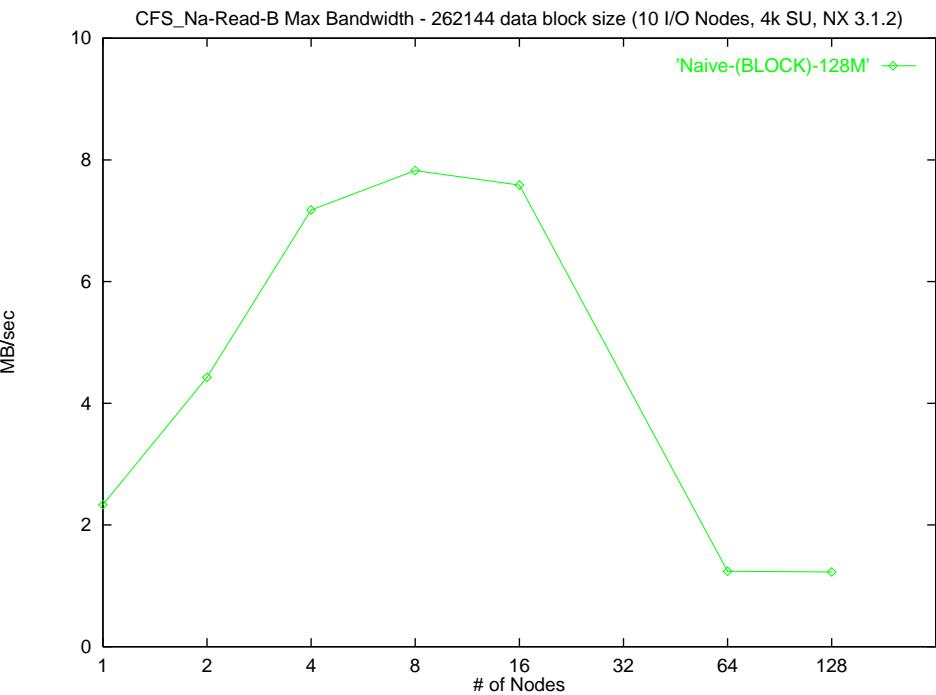
Grouping—Next set



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 18

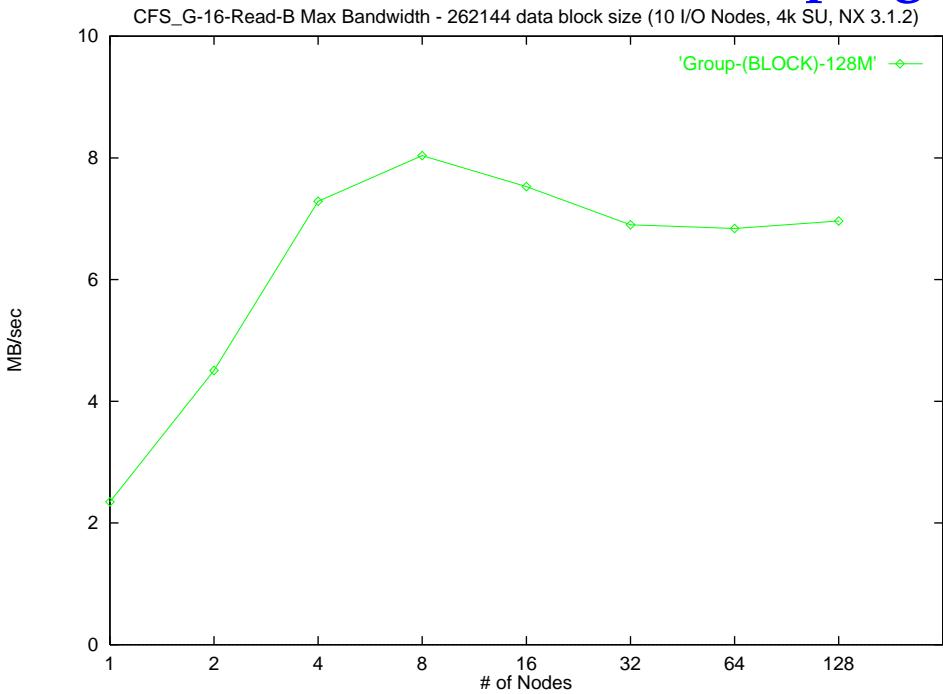
CFS Read Performance: Naive



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 19

CFS Read Performance: Grouping



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 20

Grouping Summary

Advantages

- 8x performance improvement on iPSC/860 CFS
- No need to determine the cause of the thrashing
- Group sizes can be determined empirically

Disadvantages

- Reduces parallelism
- Grouping is a “hack”; better solution is to fix the cause of the thrashing



Disk-directed I/O

Minimize disk access time by both reducing the number of disk accesses and reducing the seek time at every access.

Developed by David Kotz, Dartmouth, '94

I/O nodes collect requests, determine which local disk blocks need to be accessed, access the blocks in sorted order, and transfer each block's requested data.

Similar techniques:

“Server-Directed Collective I/O”, Seamons+, Illinois, '95
“Generalized Sliding Window”, Nitzberg, Oregon, '95



Disk-directed I/O Pseudocode for Read

Compute Nodes:

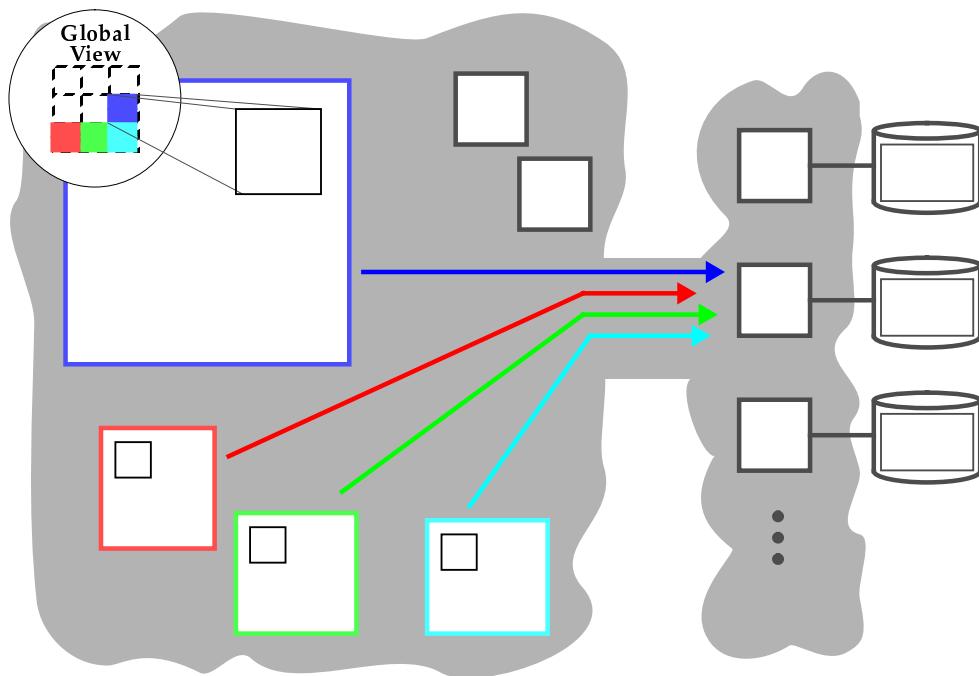
```
setup asynchronous receives for data blocks;  
barrier_synchronize();  
One node:  
    send request for all data blocks to all I/O nodes  
wait for all asynchronous receives to complete;
```

I/O Nodes:

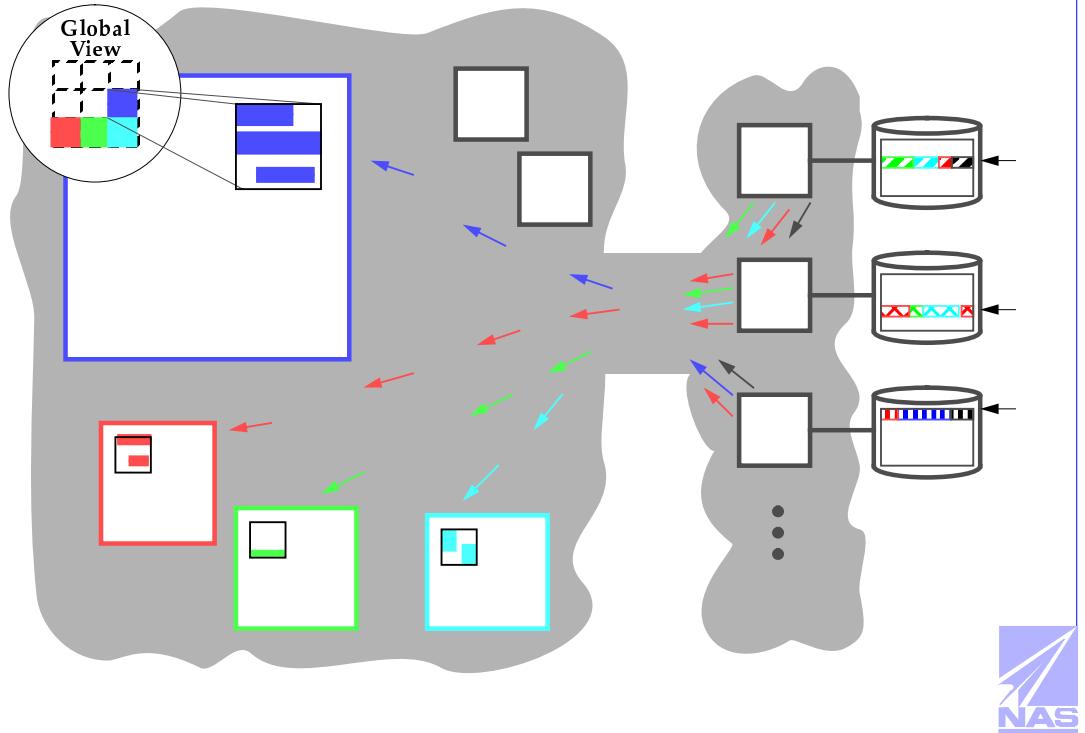
```
create list of disk blocks containing requested data;  
sort list of disk blocks;  
for each disk block (double buffering)  
    read the block from disk;  
    send pieces of the block to appropriate nodes;
```



Disk-directed I/O—Send Request



Disk-directed I/O—Transfer Data



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 25



Disk-directed I/O Summary

Performance (1D and 2D simulations only)

- ~90% of peak, up to 16 times faster than Naive

Advantages

- maximizes parallelism at the I/O nodes and disks
- minimizes disk access time (# of accesses & seeks)
- minimal buffer space (2 blocks per disk)
- extendable to support a workload

Disadvantages

- requires OS support
- best with remote memory access & scatter/gather



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 26

Collective Buffering (CB) Algorithms

**Data is permuted prior to performing I/O operations,
minimizing the number of disk operations**

Extension of serial file buffering

- masks grain size mismatch between request size and optimal access size
- evens out the bursty (uncoordinated) nature of I/O requests
- data must travel through the network twice
- requires precious resources (nodes or memory)

Target of the permutation is optimized for disk access



Collective Buffering Target

Three parameters for collective buffering target

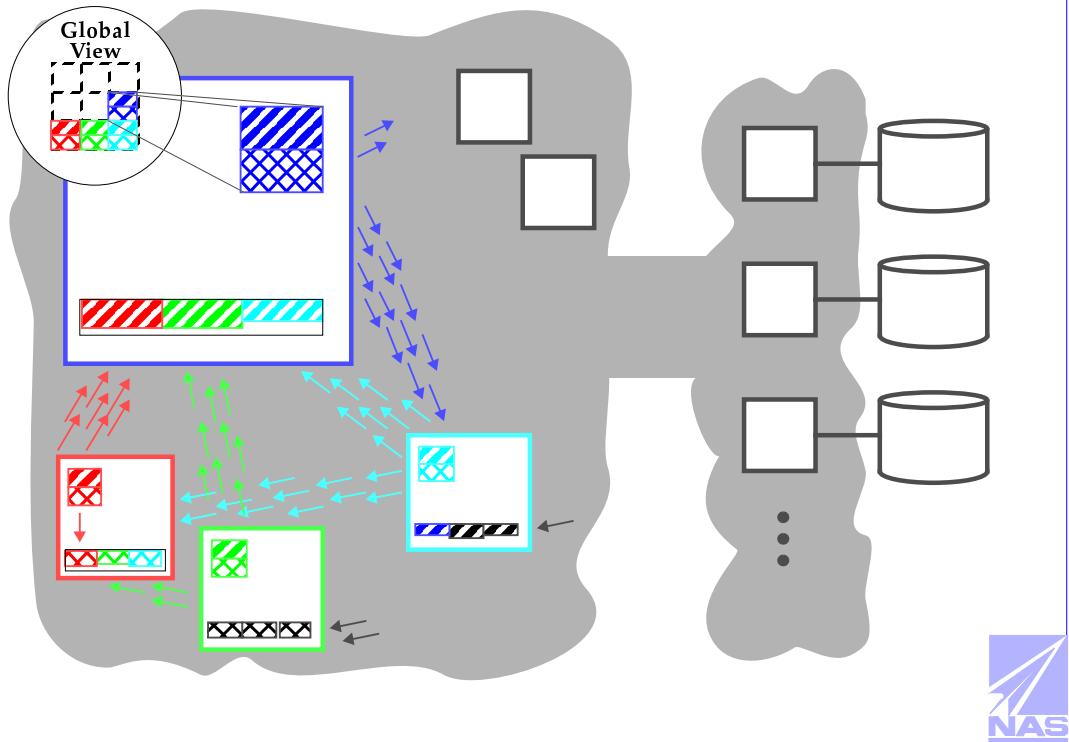
- number and identity of target nodes
- amount of target buffer space
- target data distribution

Write Pseudocode

```
if (target node)
    allocate buffer space;
forall local data blocks db:
    send db to appropriate target node
        (based on target distribution);
if (target_node)
    save data to file;
```



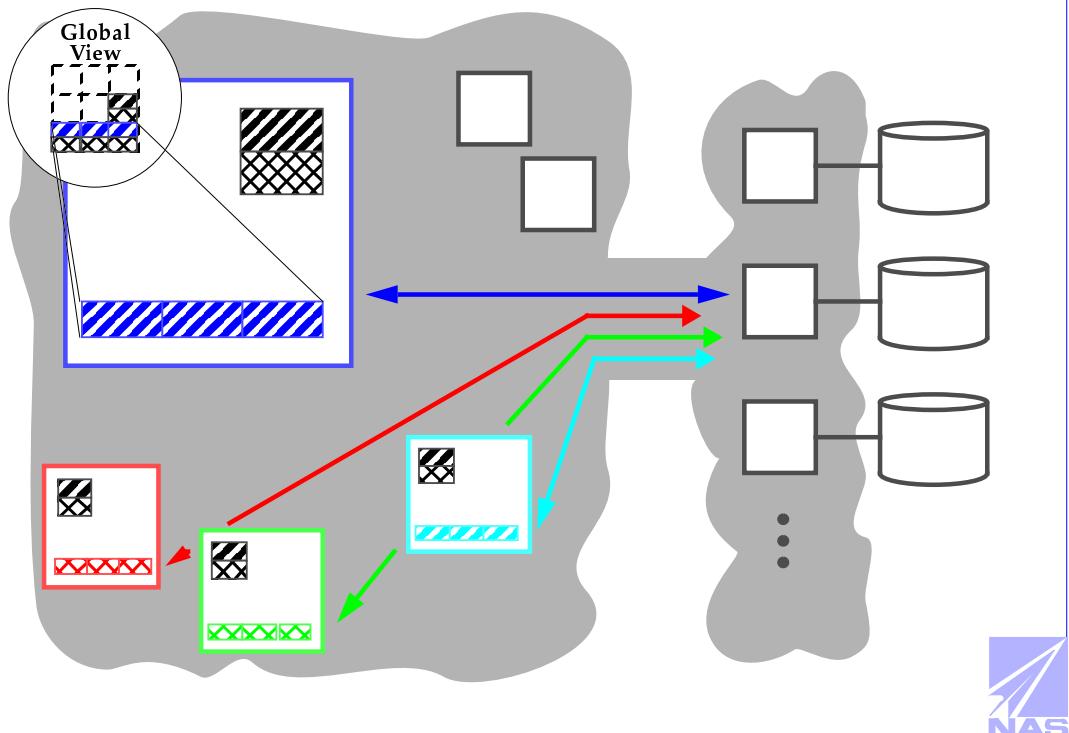
Data is Permuted



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 29

Data is Written (Read)



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 30

BLOCK Target

Target nodes:

All nodes of the application

Target buffer space:

Same as global data structure size

Target distribution:

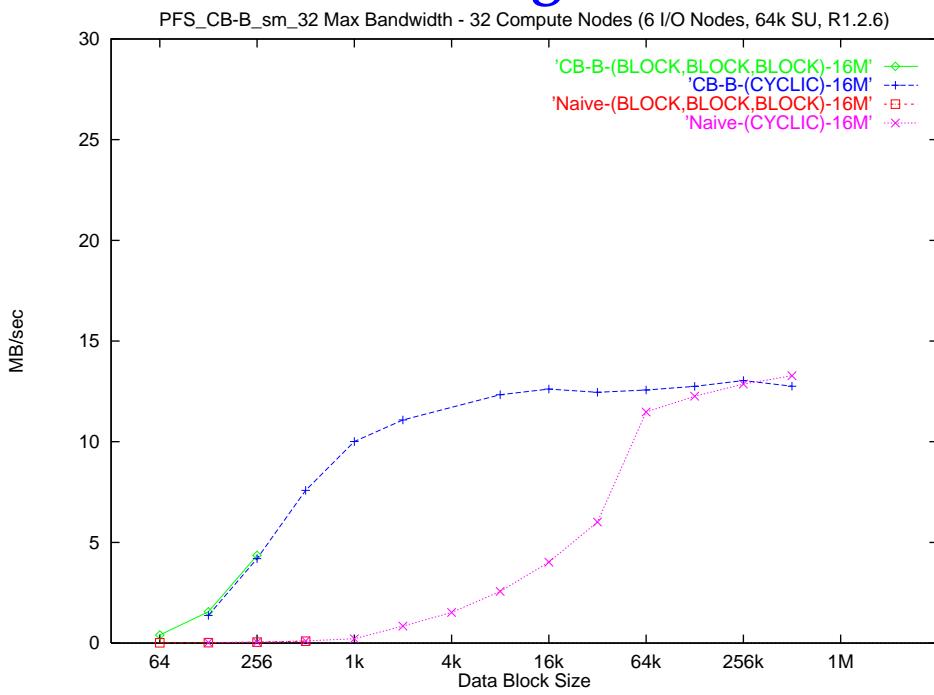
HPF BLOCK

Pseudocode

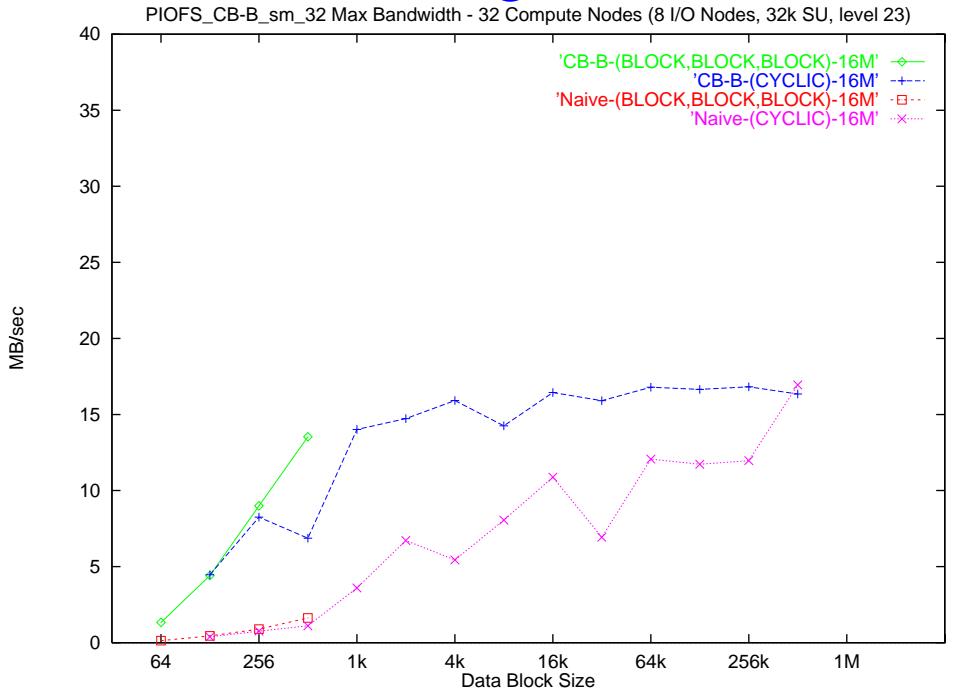
```
k = process_rank();
allocate buffer space;
set up a receive for target data;
barrier_synchronize();
foreach local data block db
    send db to the proper node;
wait for all messages to arrive;
write out the buffer;
```



BLOCK Target: PFS



BLOCK Target: PIOFS



BLOCK Target Summary

Advantages

- Good performance for > 4 KB data block sizes
- Order-of-magnitude improvements for small sizes

Disadvantages

- Requires significant resources (memory or nodes)
- Potential resource contention on I/O nodes



File Layout Target

Target nodes: Pio app. nodes ($\text{Pio} = \# \text{ I/O nodes}$)

Target buffer space: 1M

Target distribution: HPF CYCLIC(fbs)

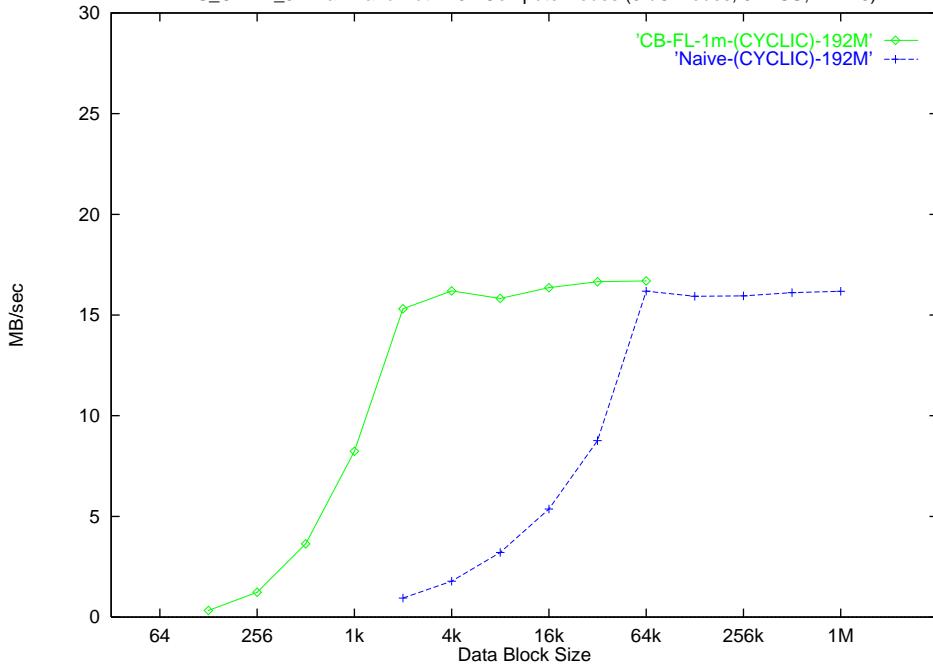
Pseudocode:

```
k = process_rank();
if (k < Pio)
    allocate buffer space;
for (r = 0; r < nrounds; r++)
    barrier_synchronize();
foreach local data block db in this round
    send db to the proper node;
if (k < Pio)
    wait for all messages to arrive;
    write out the buffer;
```

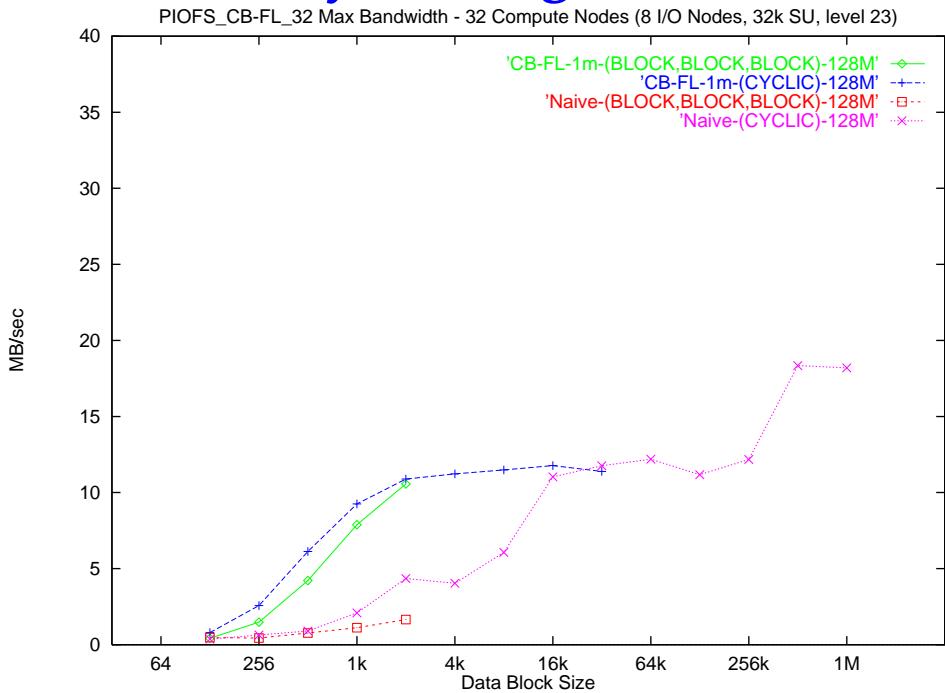


File Layout Target: PFS

PFS_CB-FL_32 Max Bandwidth - 32 Compute Nodes (6 I/O Nodes, 64k SU, R1.2.6)



File Layout Target: PIOFS



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 37

File Layout Target Summary

Advantages

- Similar performance to BLOCK target
- Less buffer space required



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 38

Generic CYCLIC Target

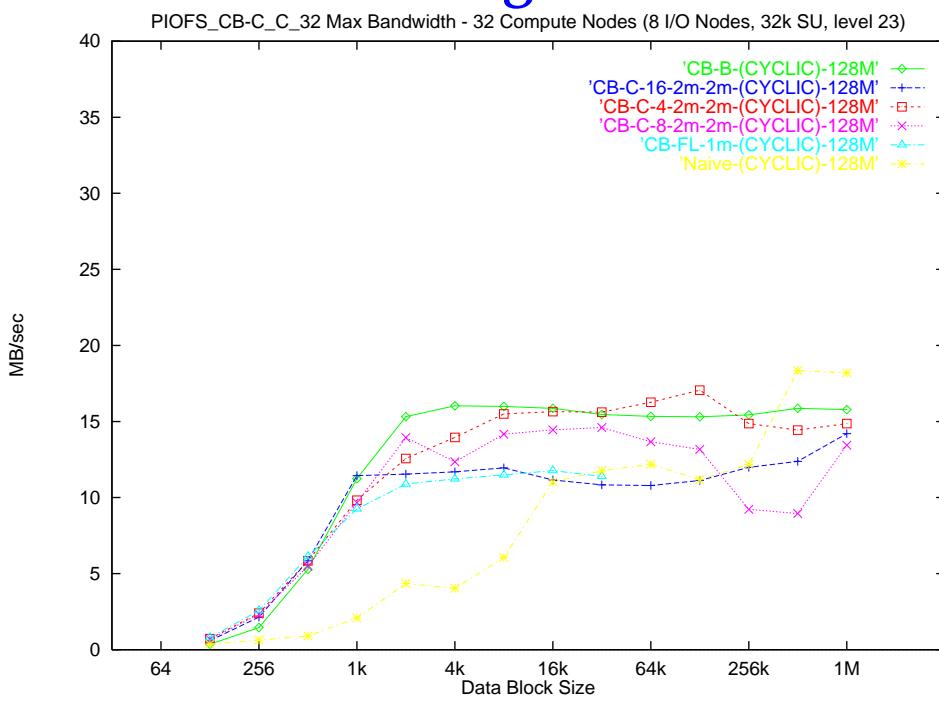
Target nodes: Variable
Target buffer space: Variable
Target distribution: HPF CYCLIC(variable)

Advantages

- CYCLIC target subsumes BLOCK and File Layout
- Provides CB performance with limited resources
- CB target parameters can be set at run time



CYCLIC Target: PIOFS



Reducing Message Overhead

Performance drop-off for small data block sizes (< 4 KB) results from message transmission overhead (latency).

Small data blocks result in numerous small messages between every compute node and every I/O node

Use scatter/gather message passing to combine messages

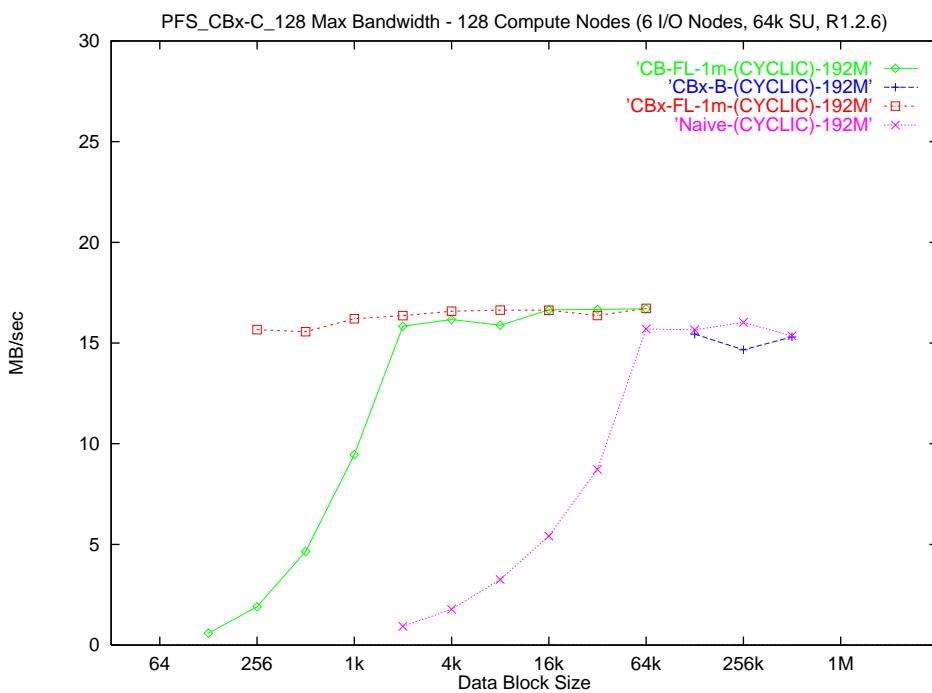
- requires OS/hardware support

Simulated results show:

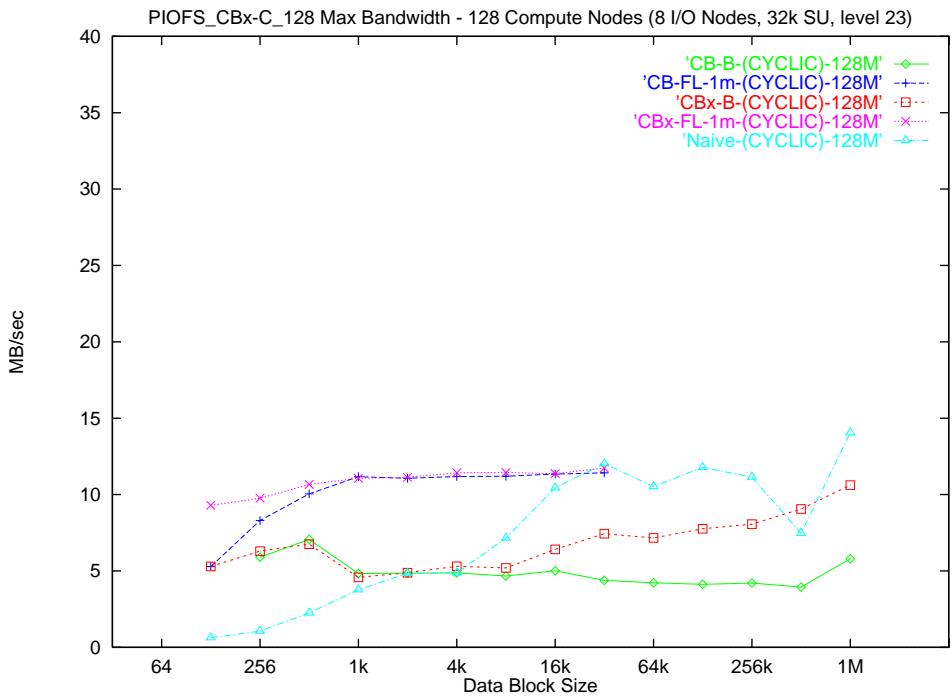
Peak performance is achieved across all data block sizes



Scatter/Gather CB: PFS



Scatter/Gather CB: PIOFS



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 43

Other CB Algorithms

Two-phase I/O, Bordawekar, Syracuse, '93

- CB-B, reading 2D arrays on Delta CFS

Collective Local I/O Optimization, Bennett+, Maryland, '94

- Auxiliary node CB, accessing 2D arrays on SP1

BTIO, Naik, IBM Research, '94

- Multi-buffering, writing 3D array on SP2

Extended Two-phase I/O, Thakur, Syracuse, '95

- Maximizes compute node parallelism (Delta CFS)

Cray T3D I/O primitives use CB, '95

PMPIO (MPI-IO), Wong+, NASA Ames, '95



SC '95 Parallel I/O Tutorial, B. Nitzberg and S. Fineberg

Algorithms, p. 44

Future Algorithms

Better Collective Algorithms

- Maximize both compute and I/O node parallelism simultaneously (Extended two-phase?)
- Dynamic self tuning

Algorithms which consider the whole scientific workload

- Disk-directed I/O?

New file layouts

- Panda?



Summary

**The straightforward Naive algorithm performs abysmally
Collective I/O permits significant optimizations**

Access scheduling

- Grouping yields 8x improvement
- Disk-directed I/O yields 16x improvement

Collective buffering

- Yields orders-of-magnitude improvement
- With scatter/gather, performance is uniform

Need a portable interface: compiler or library

