Parallel I/O

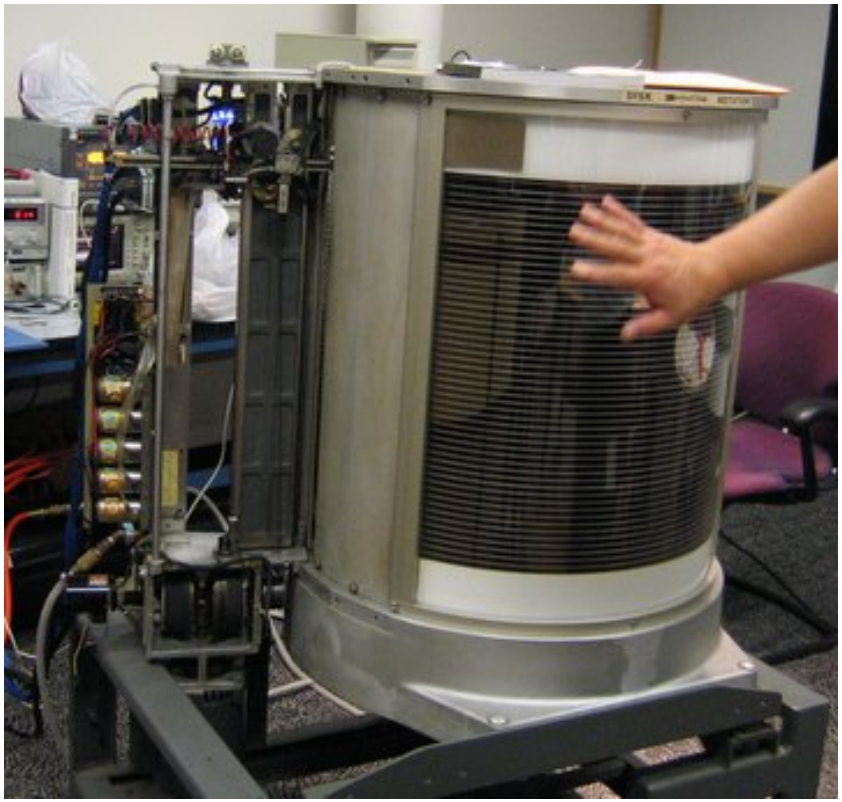
M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Behold, Magnetic Storage

First commercial disk drive, IBM RAMAC 350, introduced in 1956, leased for about 2500$/month (in 1956 $), had 50 24-inch disks:



Held 5 million 8-bit characters (and had a latency of about 1 second) ...

Storage is Slower Still ...

Capacities have obviously increased enormously - individual consumer magnetic storage devices are now over 3TB in size, but *performance* is still a serious issue:

- Typical read/write speed is on the order of 10-100 MB/s (low end for random access, higher for large sequential access)
- Latencies on the order of milliseconds (better for solid state drives)
- Disks typically lag memory performance by several orders of magnitude
- Have to access remote (rather than directly attached) disk? Gets even worse in terms of latencies and bandwidth, and the storage protocols are anything but well established ...

Parallelism on Several Levels

Parallelism gets applied on several levels to the input/output (I/O) problem:

- Hardware - RAID (Redundant Arrays of Independent Disks)
- Hardware - Dedicated Storage Interconnect
- Software - High-level libraries (that we directly access through applications)
- Software - Filesystems (Parallel? High Performance?)

Storage Capacity for HPC

How much storage is needed for a shared resource? Aside from specialized data needs (e.g. the Large Hadron Collider at CERN is estimated to generate 10-20 PB/year), a rule of thumb for HPC uses the relative mix of compute cycles (F=Flop/s) to main memory (M=Bytes) to total storage (S=Bytes).

- Desirable: M/F = 1 Byte/Flop/s (UB/CCR: about 0.2-0.4)
- Desirable: S/M = 20 (UB/CCR: total M is about 44TB, so S should be about 900TB; in fact, UB/CCR has 325TB shared in /projects and /user, and 183TB shared in /panasas, and each node locally also has S/M of 10-20)

The Need for Speed ...

Storage is slow - very slow relative to memory (which, as we have seen, is itself lagging far behind CPU performance). The slow nature of I/O (input/output) to disk is greatly aggravated in HPC, as it will quickly degrade the performance of any parallel calculation. Consider:

- Your application (weather model, CFD, stellar evolution, etc.) needs 1TB of distributed memory in which to run and keep track of its state variables (that is only about 256 cores at 4GB/core)
- You need to checkpoint (save all the variables of interest) every M minutes in order to restart your calculation later (in case of failure, analysis, etc.)
- Therefore you need to be able to output/input 1TB of data every M time units; say M=10m=600s, how fast does your storage need to be?
- The write time has to be $<<$ M (otherwise you are not going to get much done other than reading/writing!), let's say 0.01*M=6s, which means your storage needs to serve data > 167 GB/s.

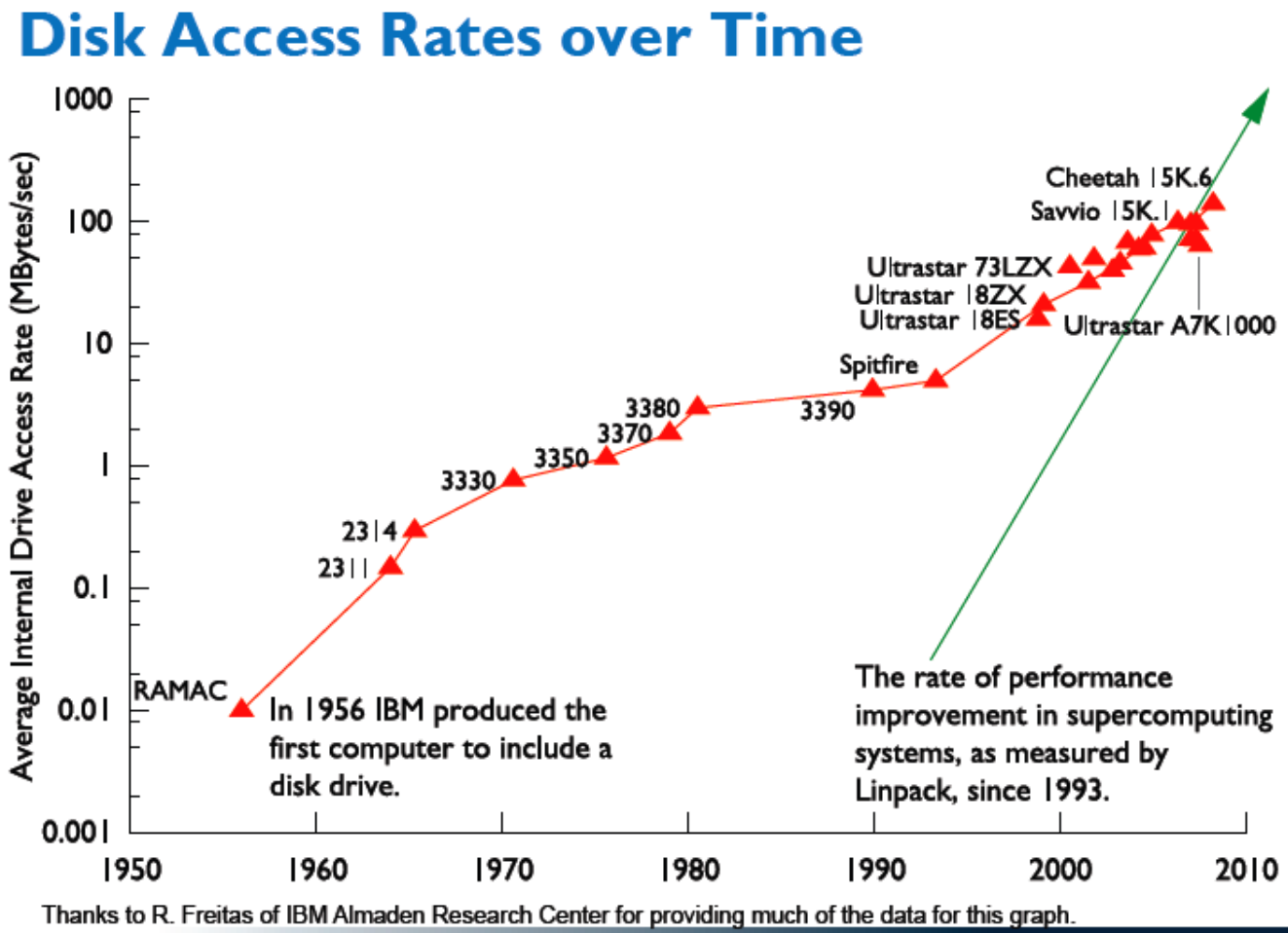
HPC I/O Challenges

Common challenges in I/O for HPC:

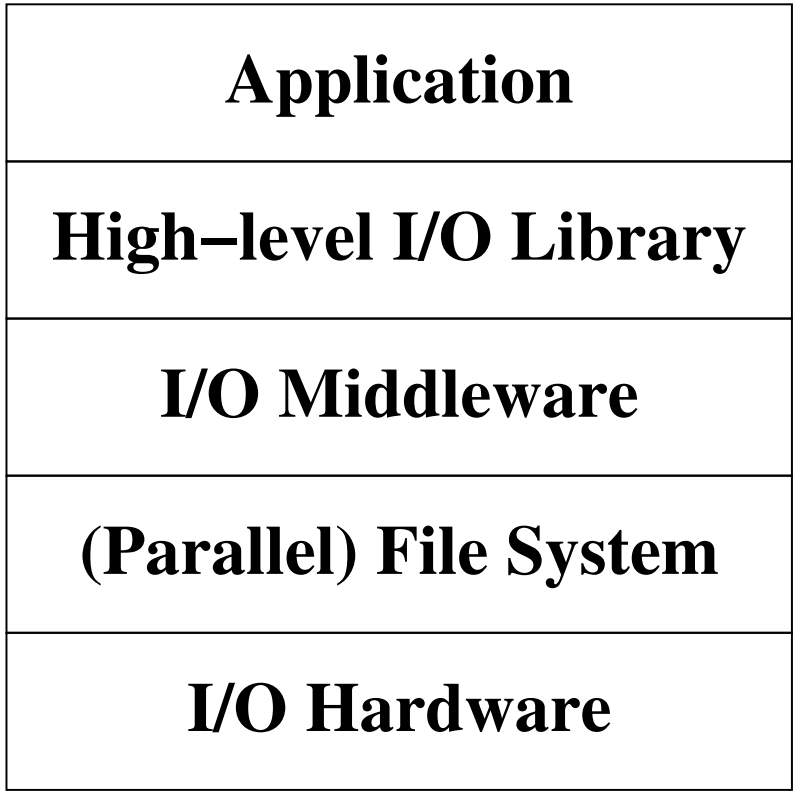
- Applications (even experimental instruments) producing increasing quantity of data - extreme example, Large Hadron Collider expected to produce 10-20 PB/year
- Applications have complicated and widely varying data structures - from hierarchical meshes to image data
- Storage devices have rather simple data layouts - trees of files into directories ...
- Trasfer rates are in even worse shape than capacities - at 50MB/s (typical high speed gigabit Ethernet link) it takes more than 5.5 hours to transfer even 1TB ...

Not Just a Network Problem

Single disk transfer rates:

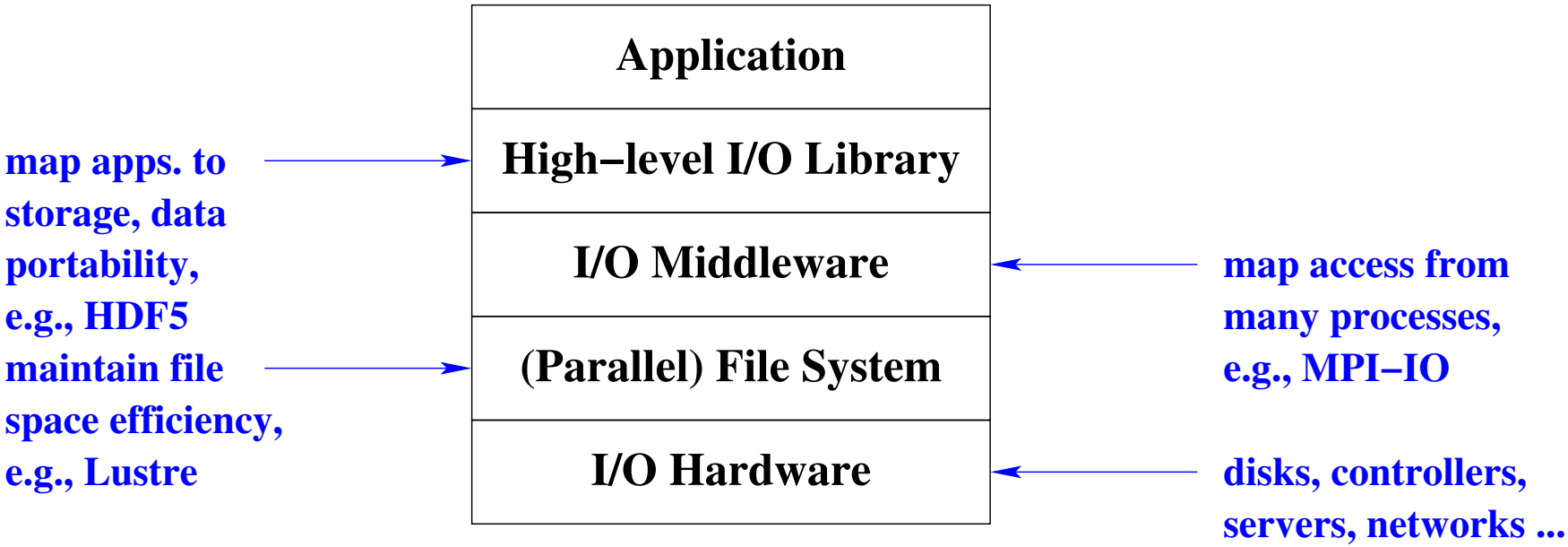


Layers for Parallel I/O



- What most concerns us: High-level libraries that can effectively utilize parallel hardware and file system resources to speed up large parallel I/O.

Layers for Parallel I/O (cont'd)



- All too often the applications simply utilize POSIX I/O operations that are entirely unsuited to a large scale parallel computation, but better high-level libraries exist and can be leveraged.

Example: Argonne BG/P

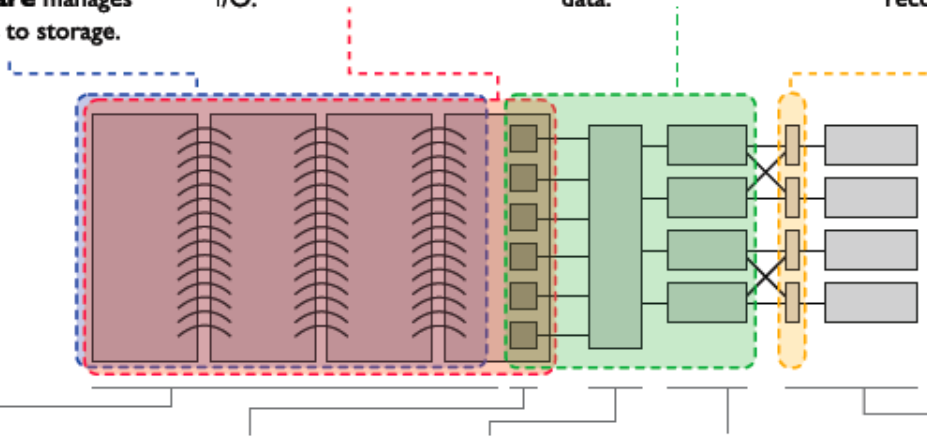
I/O Hardware and Software on Blue Gene/P

High-level I/O libraries execute on compute nodes, mapping application abstractions into flat files, and encoding data in portable formats.
I/O middleware manages collective access to storage.

I/O forwarding software runs on compute and gateway nodes, bridges networks, and provides aggregation of independent I/O.

Parallel file system code runs on gateway and storage nodes, maintains logical storage space and enables efficient access to data.

Drive management software or firmware executes on storage controllers, organizes individual drives, detects drive failures, and reconstructs lost data.



Compute nodes
40,960 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Gateway nodes
640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Commodity network
900+ port 10 Gigabit Ethernet Myricom switch complex

Storage nodes
136 two dual core Opteron servers with 8 Gbytes of RAM each

Enterprise storage
17 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Architectural diagram of the 557 TFlop IBM Blue Gene/P system at the Argonne Leadership Computing Facility.

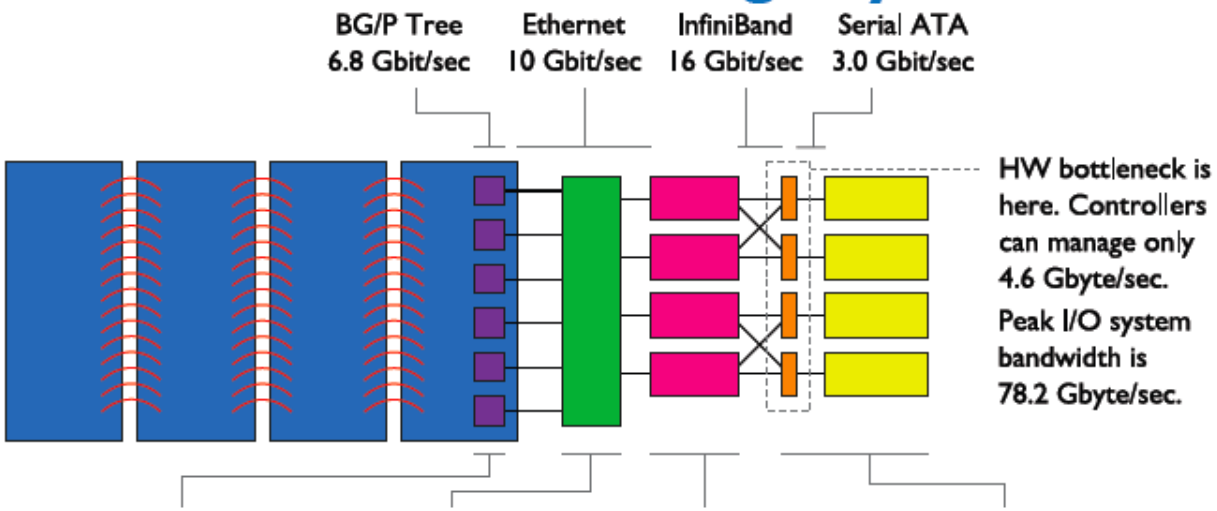
Argonne NATIONAL LABORATORY

18

panasas

Example: Argonne BG/P

Blue Gene/P Parallel Storage System



Gateway nodes run parallel file system client software and forward I/O operations from HPC clients.

640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Commodity network primarily carries storage traffic.

900+ port 10 Gigabit Ethernet Myricom switch complex

Storage nodes run parallel file system software and manage incoming FS traffic from gateway nodes.

136 two dual core Opteron servers with 8 Gbytes of RAM each

Enterprise storage controllers and large racks of disks are connected via InfiniBand or Fibre Channel.

17 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Architectural diagram of the 557 TFlop IBM Blue Gene/P system at the Argonne Leadership Computing Facility.

Argonne NATIONAL LABORATORY

34

panasas

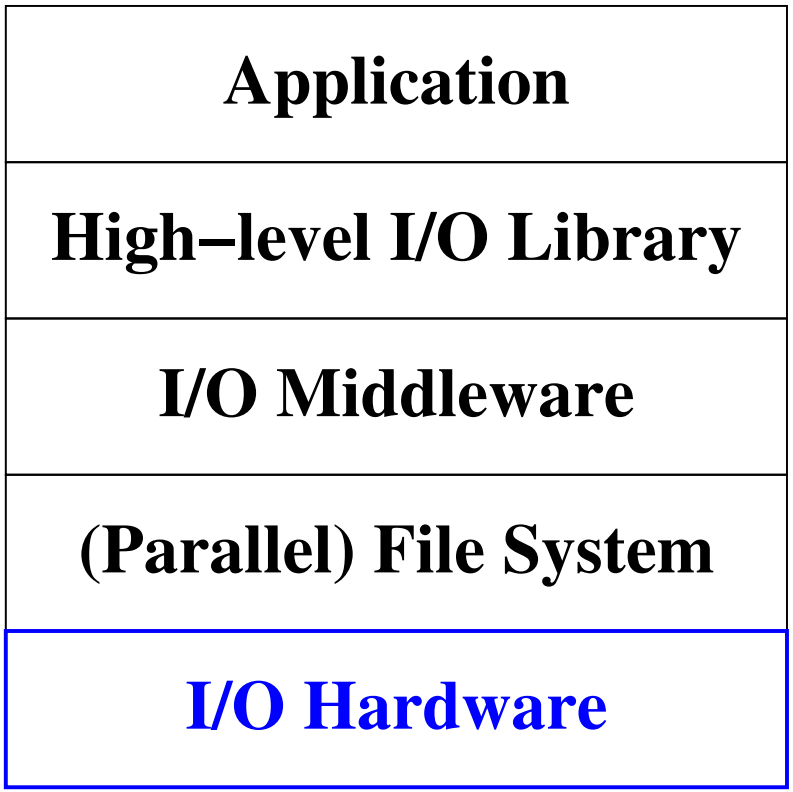
Example: Sequoia (LLNL)

Storage system architecture for LLNL Sequoia 20PFlop/s:

http://www.opensfs.org/wp-content/uploads/2011/11

Look at the "Installation-of-LLNL's-Sequoia-File-System-Marc-Stearman-Lawrence-Livermore-National-Laboratory" presentation.

Hardware



Hardware
is really a separate discussion, but let us briefly talk about some relevant issues. I/O hardware has gone parallel just as CPUs have:

- RAID (Redundant Arrays of Inexpensive/Independent Disks)
- Multiple storage controllers
- Multiple storage servers (with attached storage controllers)

It is usually left up to the parallel file system to map the hardware to the middleware (if you have any!) and the high-level libraries.

RAID

RAID (Redundant Arrays of Inexpensive Disks) has done much to alleviate the bandwidth (not the latency) bottleneck to storage. There is a lot more to RAID than we need to cover here, but the basic idea is to leverage parallel hardware for performance and redundancy

- RAID-0, pure "striping" of data across disk blocks of multiple drives (no redundancy, so actually AID and not RAID)
- RAID-1, mirrored data, duplicated across drives
- RAID-5, striping with distributed parity, can survive loss of single drive in array
- RAID-6, striping with dual distributed parity, can surive loss of two drives

RAID can be implemented in software as well as hardware ...

File Systems, Generally

- File systems serve two key purposes:
  1. Organize and maintain file space(s)
  2. Storing contents (files, directories)
- Local file system used by a single operating system/machine/client has direct access to local disk(s)
- Network file systems provide access to (many) clients through file sharing (e.g., NFS, AFS, CIFS, samba, etc.) in which data storage is not necessarily local (directly attached)
  - CCR's U2 cluster uses an NFS system from Isilon/EMC that utilizes clustered file servers to several thousand clients (/projects/ and /user filesystems)
- Parallel file systems are special; high performance network file systems that also provide the ability to concurrently access files

File System Issues

Network file systems must address some issues that are recognizable based on usual contention issues:

- Same problems as local filesystems (allocations, attributes, policies, error correction)
- Additional requirements:
  - Cache coherency - not CPU cache, but now memory-based cache of file system changes in clients/servers
  - High availability - without it, lose your file server and lose your cluster
  - Scalability - how many clients can you satisfy?

NAS versus SAN

In network-attached storage (NAS):

- File server exports local file system using network protocol (e.g., NFS/CIFS/http)
- Scalability limited by file server hardware (some companies optimize considerably)
- "Islands" of storage created when file server runs out of bandwidth to clients or to back-end storage
- Can be clustered for resiliency (still load balance issues and no concurrent access to files)

Storage area networks (SANs):

- Common access/management for storage using storage protocols
- Usually over dedicated network (e.g., fibre channel, 10 gigabit Ethernet)
- Typically fronted by multiple servers, can be exported by NAS in a hybrid arrangement
- Very expensive

NAS versus SAN Illustrated

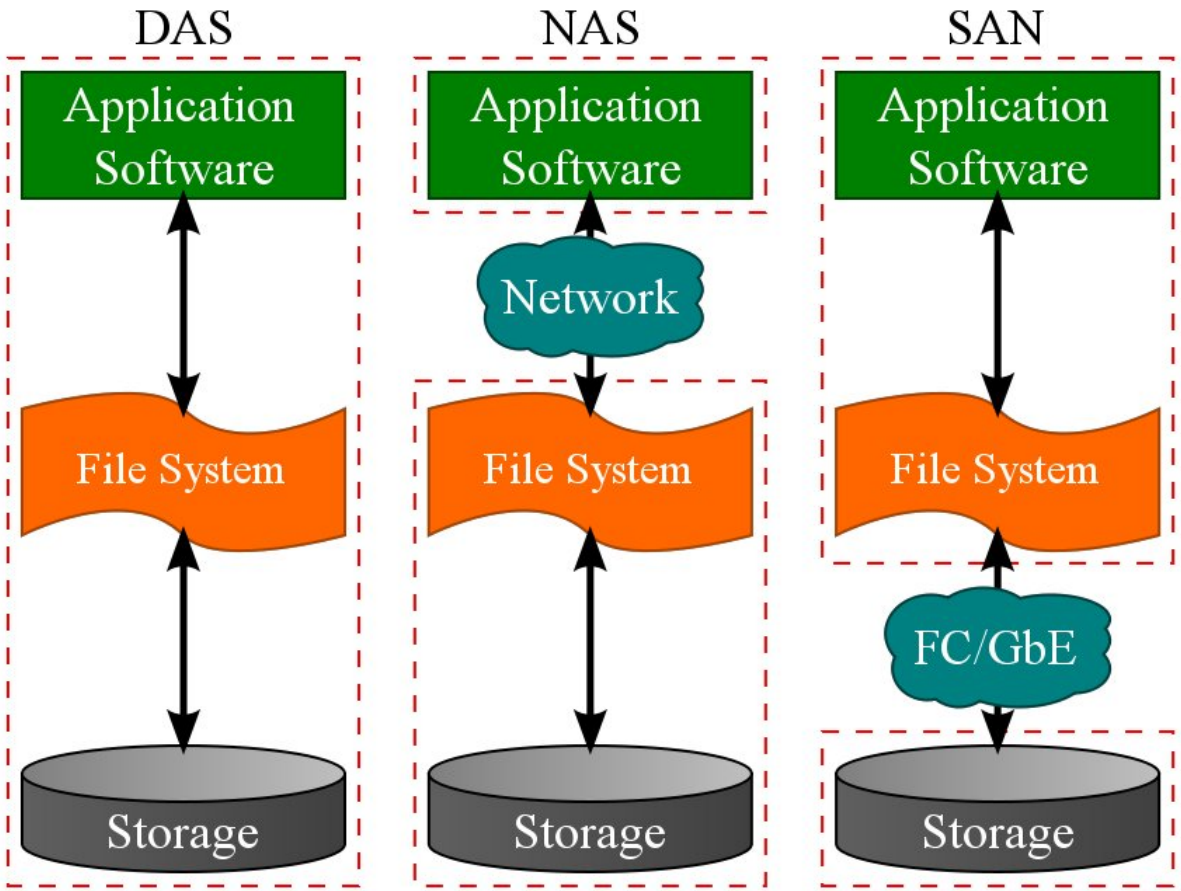
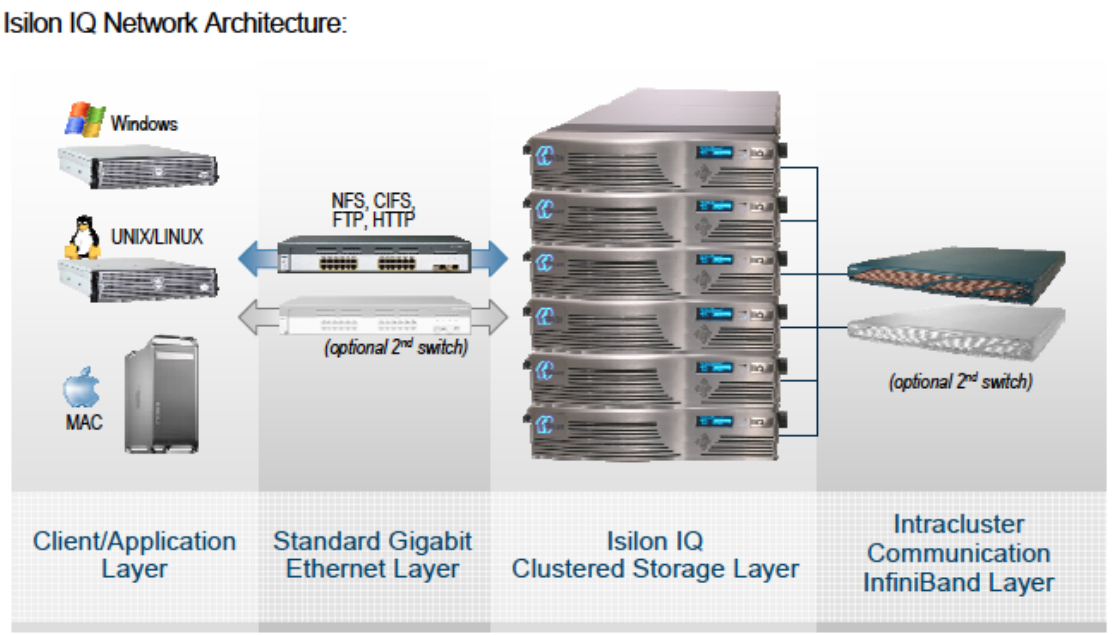


Image courtesy Wikipedia commons.

CCR NAS

CCR's NAS is from Isilon/EMC, and is a clustered filesystem:



Note that it is **not** a parallel filesystem, but is designed to be a robust NFS/NAS system.

Parallel File System Zoo

There are quite a number of extant parallel file systems. They are not really our focus as we are most concerned about the application point-of-view, but we should at least briefly cover the current players. Even though the choice of parallel file systems is not up to us as end users, they will greatly impact our performance (and sometimes the available options):

- **Lustre**
- **GPFS**
- **PVFS**
- **pNFS**

Lustre File System

Lustre (blend of Linux + cluster) is an open-source (GPL) file system that has a sizable install base in HPC:

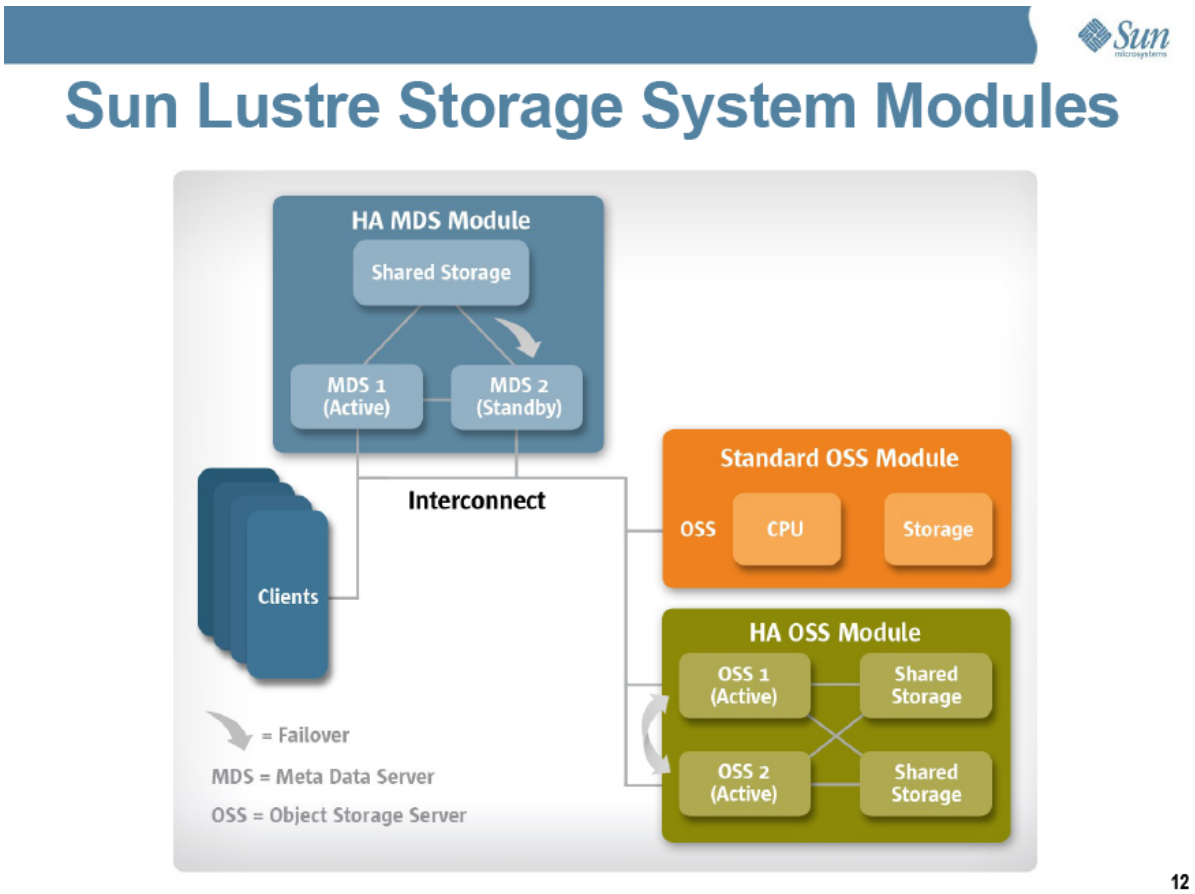
- www.lustre.org
- Originated at Carnegie Mellon University (Peter Braam) as a research project
- Developed, maintained, and supported by Sun Microsystems (acquired Cluster File Systems Inc. in 2007, itself acquired by Oracle, which has since dropped support for Lustre on non-Oracle systems)
- As of 2009-06 lustre was used on 15 of the top 30 (top500) HPC systems
- Designed for scalability, PBs of storage, tens of thousands of clients and hundreds of GB/s in aggregate throughput
- The commercial Panasas filesystem (PanFS) is similar in design to Lustre (object based, also arose from early work at CMU), and a precursor to pNFS.

Lustre Design

Lustre architecture has three main units:

1. A metadata target (MDT) for each file system that maps directories and files (and their attributes), stored on a metadata server (MDS)
2. Object storage servers (OSSs) that store data on object storage targets (OSTs); typically 2-8 OSTs/OSS, several TB in size
3. Clients that access data (can be concurrent); provides a POSIX interface, and can used RDMA over supporting networks (e.g., Infiniband, Myrinet, etc.)

Lustre Schematic



(Torben Kling-Petersen, Sun HPC Software Workshop, Regersburg, Germany 2009).

Lustre Performance

Lustre at TACC Performance

- TACC ranger system – has observed 46 GB/sec throughput
- They use 50 Sun Fire X4500 servers as OSS
- A single app achieved 35 GB/sec throughput

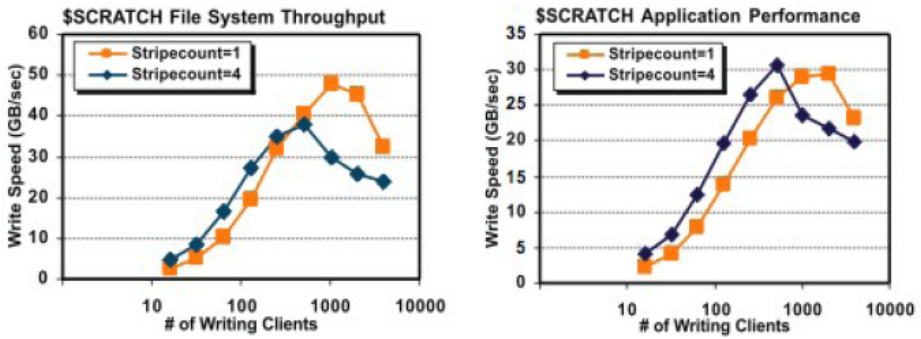


Figure 12. Lustre file system performance at TACC.

Open Storage Track at Sun Regensburg HPC Workshop 2009

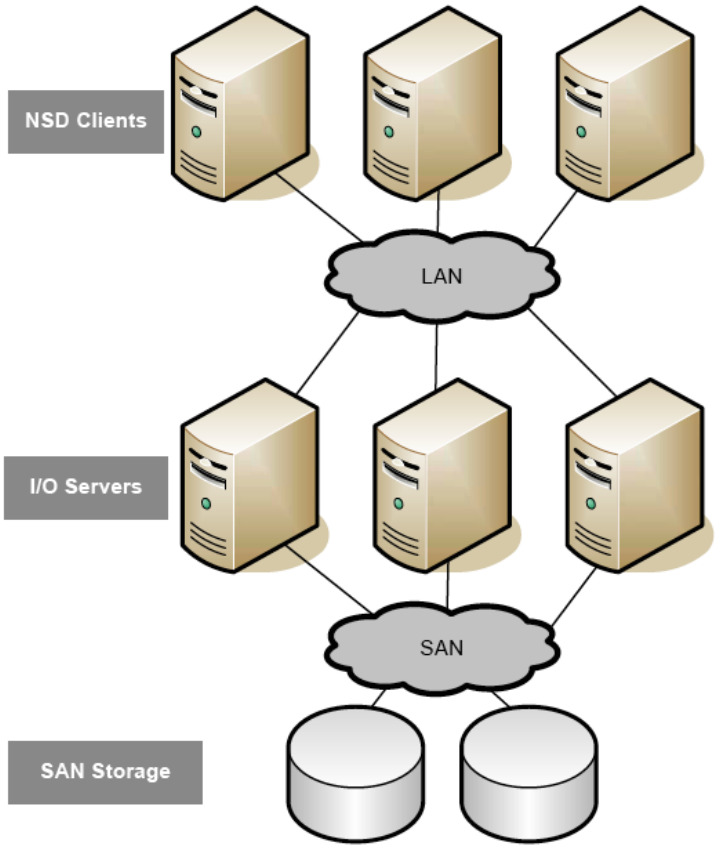
(Dan Ferber, Sun HPC Software Workshop, Regersburg, Germany 2009).

GPFS

GPFS (General Parallel File System) is IBM's clustered parallel file system product:

- Began as Tiger Shark file system at IBM Almaden, successor to PIOFS (circa 1998)
- Block decomposition (tunable size) of files across data storage resources (disks, servers, etc.), can be redundant within or across RAID
- Distributed metadata and file locking
- Provides POSIX interface and high-speed MPI-I/O
- Clients supported under AIX, Linux, and Windows (2008), support RDMA on high performance interconnects (Infiniband, Myrinet, 10 gigE, etc.)

GPFS Schematic

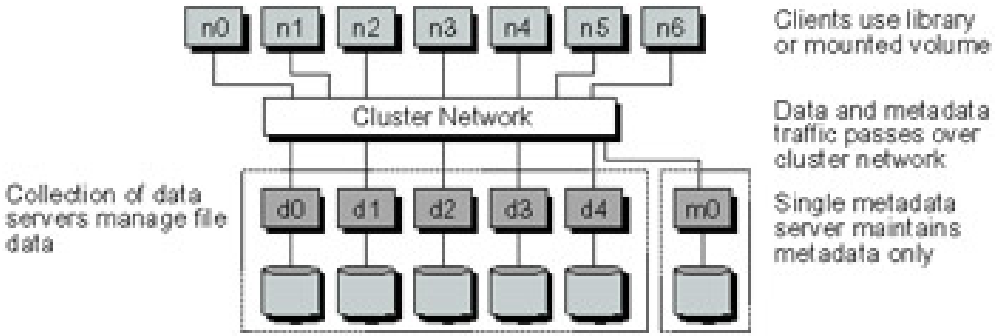


PVFS

PVFS (Parallel Virtual File System) was born as a Clemson research project around 1999:

- Open-source, freely distributed, popular among smaller clusters
- Focused on high performance for large data sets
- Provides servers and client library, including optimized MPI-IO
- Linux kernel module provides clients with POSIX interface (although not necessarily very optimized)

PVFS Illustrated



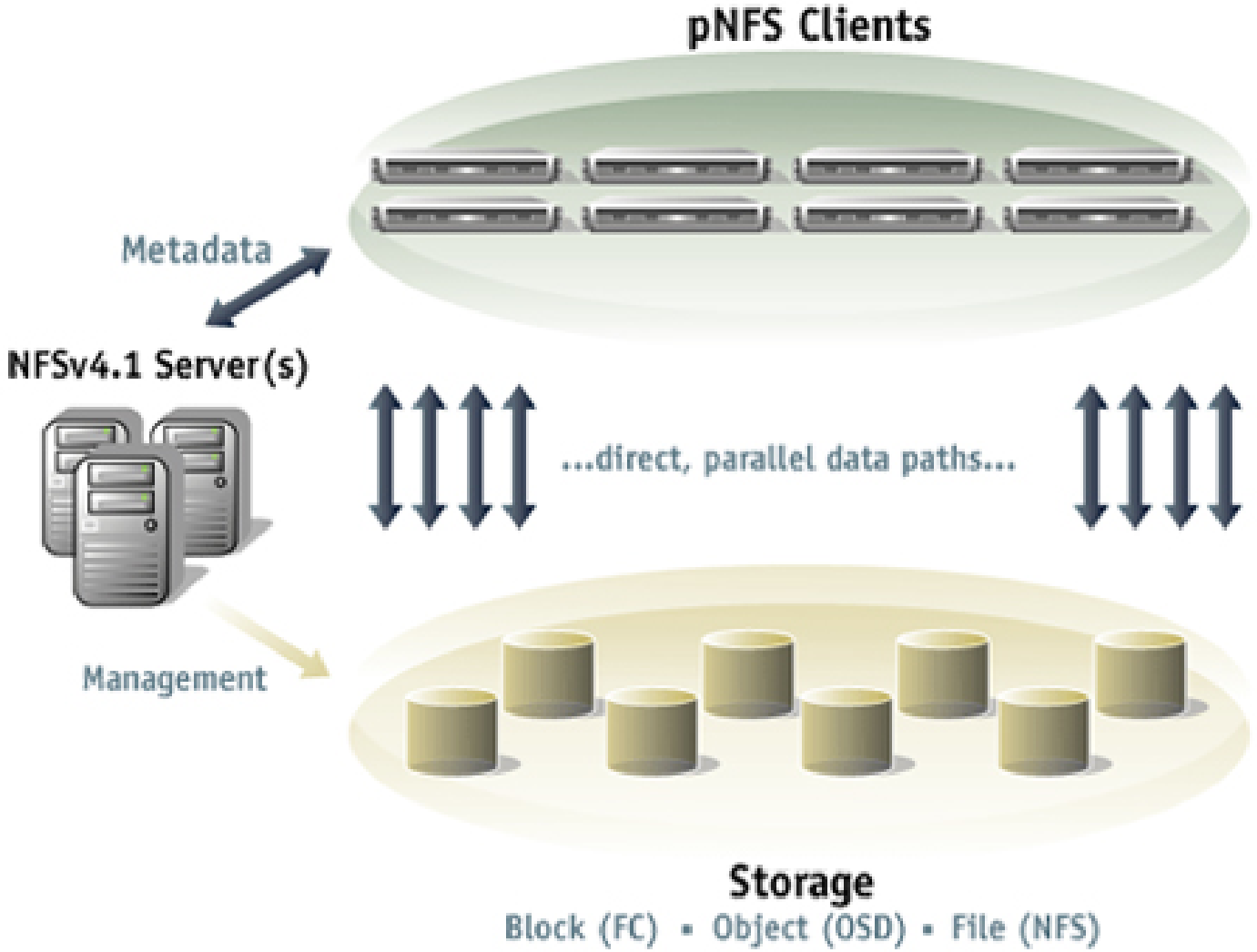
PVFS has forked into another Clemson project called OrangeFS.

pNFS

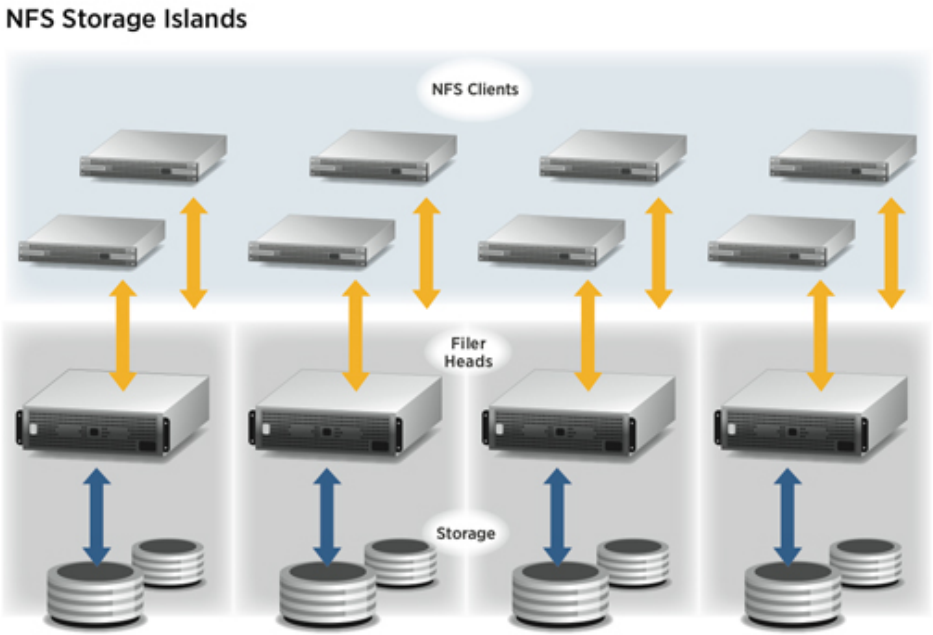
Unlike standard NFS, pNFS (a shorthand to distinguish NFS version 4.1 and later) will support scalable parallel access to clustered file servers (which currently exist as proprietary solutions of varying degrees of scalability):

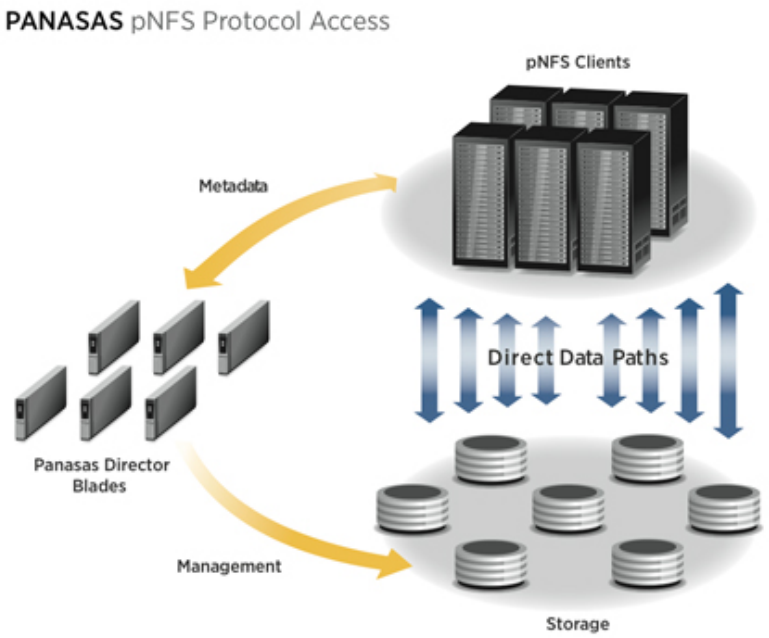
- www.pnfs.com
- Specification has been released (2009), first RFCs in 2010 (part of the Internet Engineering Task Force, IETF).
- Support starting to appear in Linux distributions (commercial products coming in 2012/2013)
- Server support anticipated from all the major storage vendors
- Preserve interoperability among different storage products while allowing high performance and scalability

pNFS Illustrated



pNFS vs. NFS

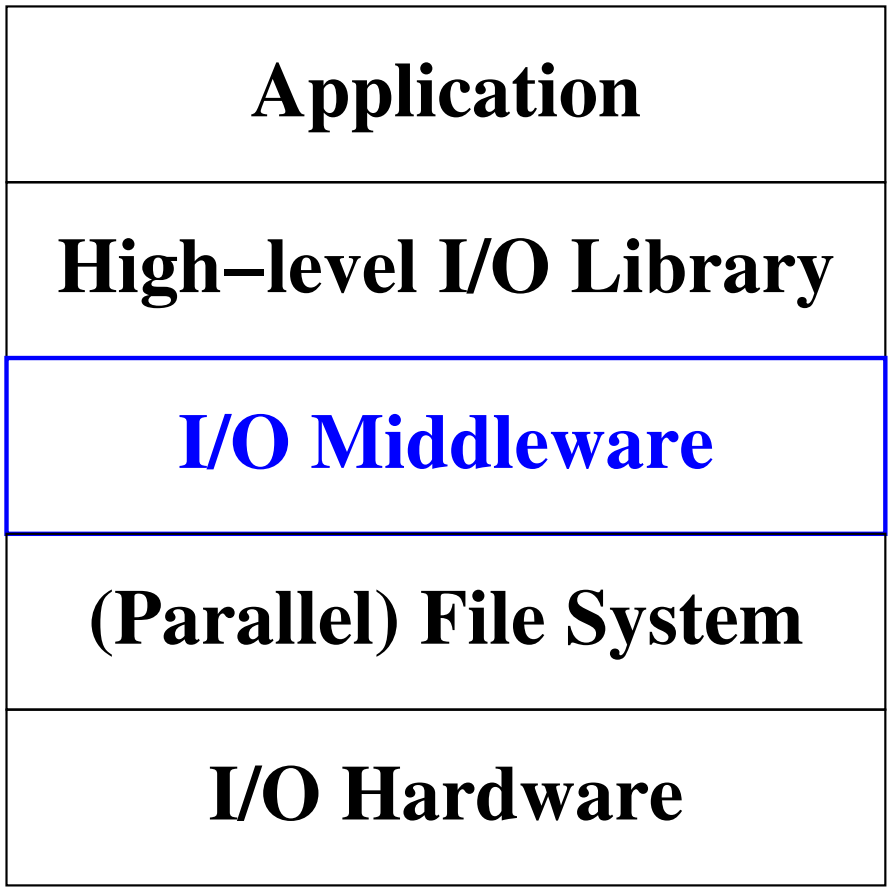




(Images courtesy Panasas Inc.) Note that pNFS client/server support is starting to become available (circa 2012) in mainstream Linux distributions. See, e.g., notes from the USENIX LISA 2011 BOF:

www.pnfs.com/docs/LISA-11-pNFS-BoF-final.pdf

Role of Parallel I/O Middleware



Generally the idea of parallel I/O middleware:

- Provide a match to the programming model (e.g., MPI)
- Provide concurrent access to groups of processes, atomically and collectively
- Leverage parallel file system attributes (scalable features, atomicity, etc.)
- Create a portable generic foundation for high-level libraries

POSIX I/O

So what is POSIX I/O again?

- POSIX == IEEE Portable Operating System Interface for Computing Environments (for Unix - hence the X)
- IEEE 1002 standard (ISO/IEC 9945), original adoption IEEE Std 1002.1-1989
- POSIX is much more than just I/O (also command line interface, shell behavior, C library, etc.)
- POSIX provides the interface between applications (`fopen, fread, fwrite, ...`) and the operating system
- Windows requires `cygwin` or an optional Windows subsystem for POSIX compliance
- POSIX I/O created for and best suited to a single local file system
- POSIX is a separate topic in its own right, and significantly off topic ...

POSIX Parallel Issues

Basically POSIX is entirely unsuited to parallel I/O:

- POSIX I/O inherently serialized
- No ability to perform concurrent operations on files
- Very expensive to support full POSIX features for heavily shared files (e.g., NFS will cheat to gain performance - ever do a big "make" on an NFS file system and see it complain about time stamps in the future?)
- Strict locking to guarantee atomicity of file operations (substantial overhead)

POSIX HECEWG

There is a working group proposing extensions to the existing POSIX standard to provide an API that would better suite large cluster/HPC environments:

- HECEWG - High End Computing Extensions Working Group
- Relax POSIX sematics that are most expensive in large shared environments, or provide information about know data access patterns
- http://www.opengroup.org/platform/hecewg

MapReduce

What is this MapReduce of which you speak?

- Patented framework design of Google, designed from the outset for distributed computation on very large data sets (justification obvious?)
- More than just I/O, really, but it is closely tied to same ideas
- Large number of processors ("nodes") in a cluster collectively process huge data set - can be structured (database) data, or unstructured (stored in a filesystem)
- "Map" - master node(s) decompose input, and distribute to workers
- "Reduce" - master node(s) take sub-answers and combine them

7650331

U.S. Patent Jan. 19, 2010 Sheet 3 of 7 US 7,650,331 B1

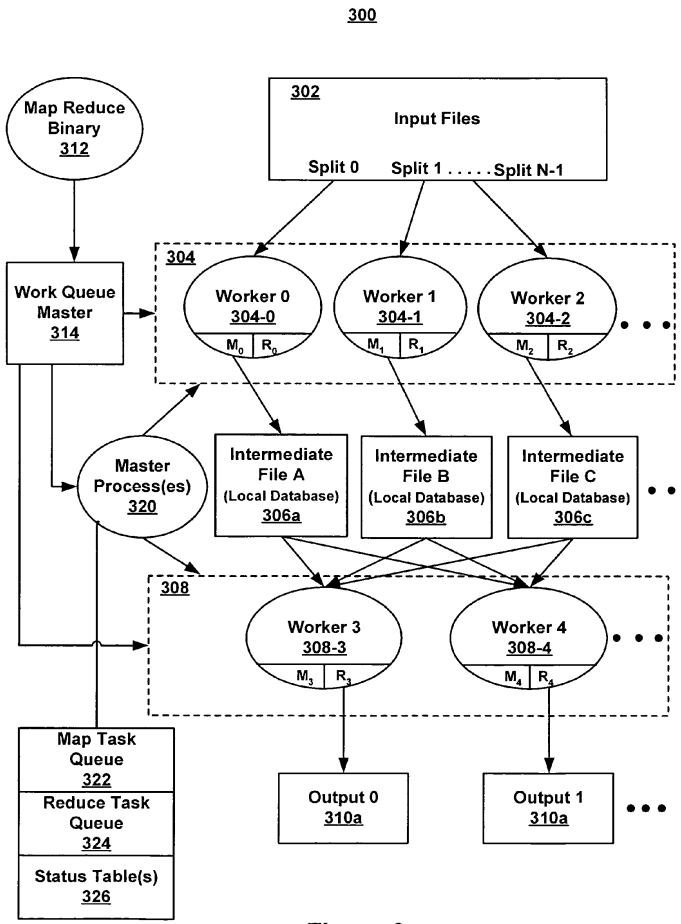


Figure 3

Advantages

Advantages of MapReduce:

- For independent sub-problems, similar to Monte Carlo methodology, very high degree of parallelism
- Data kept close to computing elements working on that subset of data (recall that Google has their own parallel filesystem as well)
- Large cluster can process PB of data in relatively short time (hours)
- High degree of reliability/availability - error recovery can be pretty easy as long as the data persists
- Connection between Map and Reduce processes often in the filesystem itself (not the only possibility)

MapReduce Implementations

- Google - proprietary (C++ with python and Java interfaces)
- Apache Hadoop - free, Java-based (Yahoo! has been a big booster, reported to use it extensively), has its own filesystem (more on this later)
- Mars, interesting project for MapReduce on Nvidia GPUs: `http://www.cse.ust.hk/gpuqp/Mars.html`
- Many more proprietary implmentations (e.g., Oracle)

MapReduce at CCR

Notes for running Hadoop on the UB/CCR cluster:

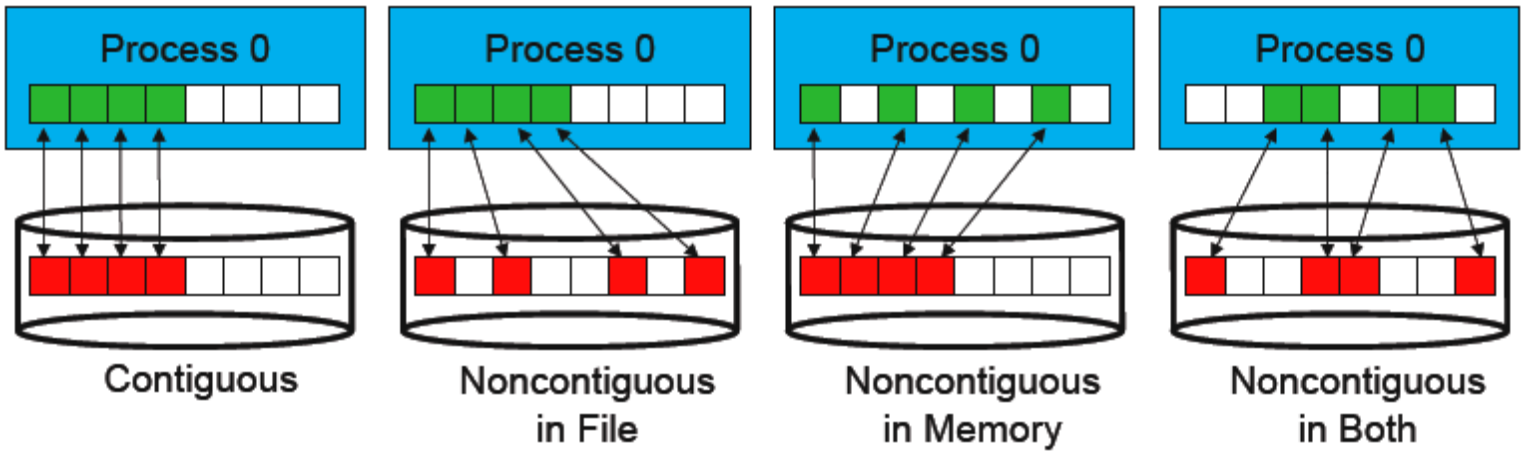
http://ccr.buffalo.edu/support/software-resources/mpi-multi-threaded/hadoop.html

MPI-IO

MPI-IO is an I/O interface specification for use in MPI applications:

- Data model is the same as POSIX - stream of bytes in a file
- Provides:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - System for encoding files in a portable format (`external32`)
  - Implementations widely available (officially part of MPI-2, but ROMIO from ANL widely available)

Contiguous & Noncontiguous I/O



- **Contiguous** - move single block of memory to single file region
- **Noncontiguous** - can be irregular in memory, file, or both
- Structured data (e.g., decomposition in multiple dimensions) leads naturally to noncontiguous I/O
- Using noncontiguous operations tells the I/O subsystem more about the data

Nonblocking & Asynchronous I/O

Nonblocking and asynchronous I/O:

- Blocking (or synchronous) I/O operations return when the buffer can be safely reused (data could be written to disk or stored in system buffers)
- Can hide I/O latencies using nonblocking I/O - overlap computation with I/O (submit I/O operations, later test for completion)
- Asynchronous I/O (on platforms that support it) can allow I/O operations to be completed in the background

MPI-IO Basics

MPI-IO will look pretty familiar to someone already versed in the non-I/O abilities of MPI:

- MPI-IO was designed to be separately implemented from remainder of MPI (i.e. MPI-IO does not require a full MPI library)
- MPI-IO consists of about 60 functions - we will just look at some of the highlights
- Basic building blocks for MPI-IO are communicators and MPI datatypes
- File access and message passing are actually fairly analogous

Basic File Access

```
1  MPI_File fh;
2  MPI_Status status;
3
4  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
6
7  bufsize = FILESIZE/nprocs;
8  nints = bufsize/sizeof(int);
9
10 MPI_File_open(MPI_COMM_WORLD, "/my/parallelfs/datafile",
11               MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
12 MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
13 MPI_File_read(fh, buf, nints, MPI_INT, &status);
14 MPI_File_close(&fh);
```

File Access with Offsets

```
1   include "mpif.h"
2
3     integer status(MPI_STATUS_SIZE)
4     integer (kind=MPI_OFFSET_KIND) offset
5   ! in F77, see implementation notes (might be integer*8)
6
7     call MPI_FILE_OPEN(MPI_COMM_WORLD, "/my/parallelfs/datafile", &
8               MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
9     nints = FILESIZE / (nprocs*INTSIZE)
10    offset = rank * nints * INTSIZE
11    call MPI_FILE_READ_AT(fh, offset, buf, nints,
12                          MPI_INTEGER, status, ierr)
13    call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
14    print *, "process ", rank, "read ", count, " integers"
15
16    call MPI_FILE_CLOSE(fh, ierr)
```

Basic Writes

- Use MPI_File_write or MPI_File_write_at
- Use MPI_MODE_WRONLY or MPI_MODE_RDWR as the flags to MPI_File_open
- If the file does not exist previously, the flag MPI_MODE_CREATE must also be passed to MPI_File_open
- We can pass multiple flags by using bitwise-or "|" in C, or addition "+" in Fortran

File Views

In MPI-IO file views allow processes to be assigned separate regions of the shared file:

MPI_File_set_view

```
MPI_File_set_view(fh,disp,etype,filetype,datarep,info)
```

MPI_File fh (IN), file handle
MPI_Offset disp , byte displacement
MPI_Datatype etype ,
MPI_Datatype filetype ,
char *datarep , data representation
MPI_Info info ,

set by a triplet:

displacement : number of bytes offset from beginning of file
etype : any MPI datatype (predefined or derived)
filetype : which portion of the file is visible to the process

File View Example

```
1  MPI_File thefile;
2
3  for (i=0; i<BUFSIZE; i++)
4      buf[i] = myrank * BUFSIZE + i;
5      MPI_File_open(MPI_COMM_WORLD, "testfile", MPI_MODE_CREATE | MPI_MODE_WRONLY,
6                    MPI_INFO_NULL, &thefile);
7      MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int), MPI_INT,
8                    MPI_INT, "native", MPI_INFO_NULL);
9  MPI_File_write(thefile, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
10 MPI_File_close(&thefile);
```

Collective Operations

Collective blocking MPI-IO operations look just like their non-collective counterparts:

- Collectives have same argument list, just get `_all` suffix to names
- *May* be implicit barrier among participating processes
- Collectives allow MPI-IO software to optimize file accesses

Nonblocking Operations

Unsurprisingly, nonblocking MPI-IO calls pick up an "i" prefix to the names:

```
1  MPI_Request request;
2  MPI_Status status;
3
4  MPI_File_iwrite_at(fh, offset, buf, count, datatype, &request);
5
6  for (i=0; i<1000; i++) {
7      /* perform some computations, overlap with I/O */
8  }
9
10 MPI_Wait(&request, &status); /* wait for buf to clear */
```

Nonblocking Collectives

Nonblocking MPI-IO collectives are special, in that the usual MPI rule of not allowing multiple simultaneous collective operations on a communicator would be violated (here communicator is analogous to a file).

- Compromise is called *split collective data access*, collectives are used to start and finish nonblocking data access
- No request object needed (only one outstanding request at a time)

```
1  MPI_File_write_all_begin(fh, buf, count, datatype);
2
3  for (i=0; i<1000; i++) {
4      /* perform some computations */
5  }
6
7  MPI_File_write_all_end(fh, buf, &status);
```

Shared versus Local Pointers

Shared file pointers are another method for calculating offsets in MPI-IO:

- Third method for offsets - other two are local file pointers and explicit offsets
- Processes having shared pointers must have identical views
- Shared pointers allow coordinated access (therefore generally much lower performing)
- Suffix `_shared` for non-collectives, `_ordered` for collectives

Data Representation

Remember the string to indicate the data representation in setting file views? There are three predefined values:

"native" Data is stored on a file the same way it is stored in memory. fast but non-portable (use it when writing scratch files).

"internal" portable data format, which is supported across various platforms by a given MPI implementation, e.g., MPICH. You may not be able, in principle, to write data in this format with MPICH and then read it with LAM MPI or with IBM MPI. But you should be able to write data in this format with MPICH on Solaris and then read it, say, on DEC Alpha.

"external32" All data is converted to and from "external32," should work from MPI to MPI and from vendor to vendor. But data precision can be lost (32-bits only), and I/O should be expected to suffer.

MPI-IO Hints

Hints can be used in combination with the MPI-IO calls:

MPI_File_open

MPI_File_set_info

MPI_File_set_view

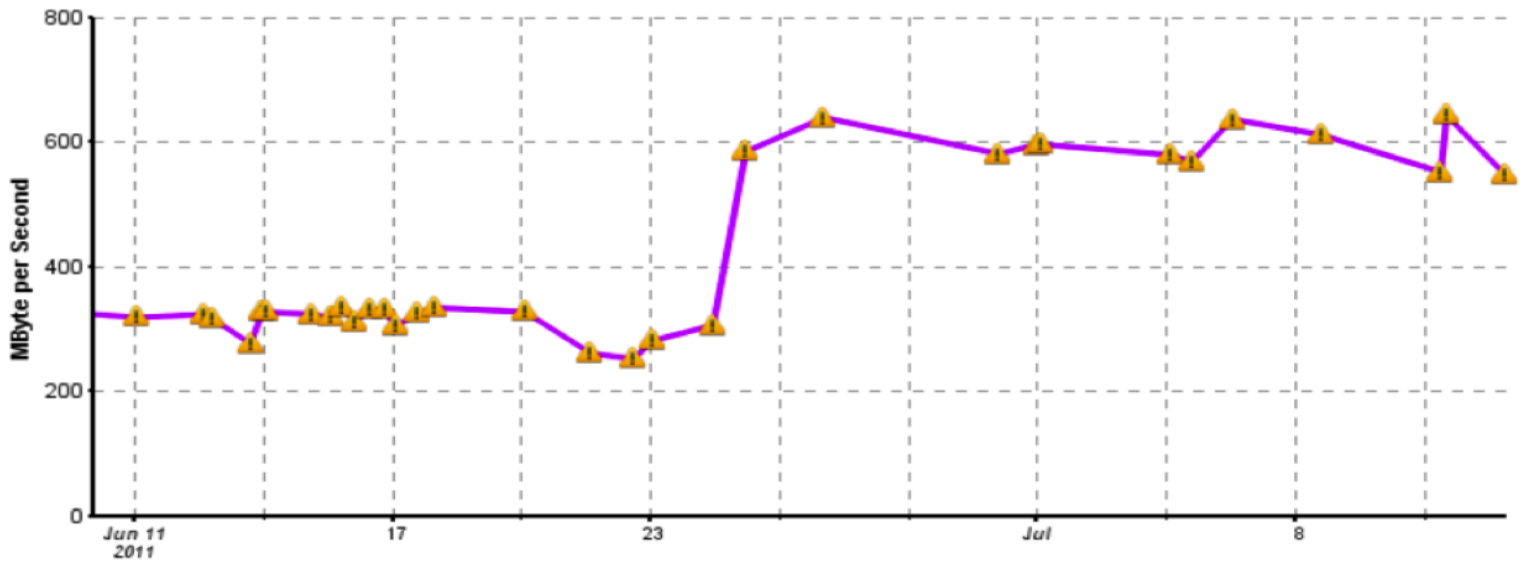
- Hints are entirely optional to the implementation (should be ignored by those not supporting those hints)
- `MPI_File_get_info` can be used to get list of hints

Hint Example (ROMIO)

Simple example for ROMIO (ANL MPI-IO implementation):

```
1  MPI_Info info;
2
3  MPI_Info_create(&info);
4
5  /* number of I/O devices used for file striping */
6  MPI_Info_set(info, "striping_factor", "4");
7
8  /* striping unit in bytes */
9  MPI_Info_set(info, "striping_unit", "65536");
10
11 MPI_File_open(MPI_COMM_WORLD, "/my/parallelfs/datafile",
12         MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);
13
14 MPI_Info_free(&info);
```

Another Hint Example



MPI Tile IO benchmark (http://www.mcs.anl.gov/research/projects/pio-benchmark/), run on 64 processes of U2. Transition shows adoption of Intel MPI 4.0.1 which supported the `panfs_concurrent_write` file hint.

More MPI-IO File Hints

Most MPI-IO implementations are based on `ROMIO`, the ANL project to support MPI-IO, so that is a good spot to start looking for documentation on file hints (and other issues):

- http://www.mcs.anl.gov/research/projects/romio
- http://www.mcs.anl.gov/research/projects/romio/doc/users-guide/index.html

Tips for MPI-IO

Some good general guidelines for achieving good performance with MPI-IO:

- Use sufficient I/O hardware (of course that may not be up to you)
- Use fastest available file systems (**not** NFS-mounted directories)
- Do not perform I/O from only one process
- Make large requests when possible (latency in I/O is a killer)
- For noncontiguous requests, use derived datatypes and a single collective I/O call
- Use appropriate MPI-IO file hints (which depend a lot on the underlying filesystem, so you may have to do some research to find them)

NPB BT Benchmark

As an example, let us consider the NPB (NASA Parallel Benchmarks), a suite of application kernels and simulated applications based on applications used by NASA (primarily computational fluid dynamics, or CFD).

http://www.nas.nasa.gov/Resources/Software/npb.html

One of the simulated applications, a block-tridiagonal solver (BT), mimics codes like NASA's *ARC3D*, for solving multiple independent systems of non-diagonally dominant block-tridiagonal equations. This benchmark also will utilize I/O in several flavors when available.

BT I/O Decomposition

The NPB BT solver uses 5x5 blocks and a 3D Cartesian grid:

- N^3 sub-cubes, or cells, the decomposition for which is that the code must run on N^2 processing elements
- So each parallel process gets N (disjoint) cells, in any slice of the x, y, or z directions, any process owns only one of the N^2 cells in that slice
- Solution field has (you guessed it) 5 components, the solution at each time step sweeps back and forth in each spatial dimension (corresponds to forward elimination and back-substitution)
- Periodically the solution is written out to disk (i.e. later visualization or analysis)

BT I/O Subtypes

THE NPB BT I/O benchmark has four different I/O types:

- **full:** MPI-I/O with collective buffering; data scattered in memory among processes is collected on a subset and rearranged before writing to increase granularity
- **simple:** MPI-I/O without collective buffering - lots of seek operations necessary to write scattered data
- **fortran:** Similar to **simple**, but uses Fortran direct-access I/O
- **epio:** "Poor man's" parallel I/O, each process writes its own data to a separate file. Does not conform to benchmark guidelines, but would require patching the data back together again to make use of it. Upper limit to performance.

Various size settings:

- **C** 162 grid points in each direction, about 6GB total for 40 time steps worth of data
- **D** 408 grid points in each dimension, about 101GB

NPB BT I/O Summary

Basic I/O steps associated with the NPB BT benchmark:

- Setup - open/close the output file
- Time step - output solution at various time steps
- Norms - read solution, accumulate solution norm over time steps

Another feature that NPB BT I/O shares with many applications is that you can switch from MPI-I/O to "regular" Posix I/O (in this case the usual Fortran read/write statements).

NPB BT Fortran-I/O

Pieces of NPB I/O with Fortran/Posix I/O:

```
1       open (unit=99, file=filenm,
2      $      form='unformatted', access='direct',
3      $      recl=record_length)
```

```
1       do cio=1,ncells
2           do kio=0, cell_size(3,cio)-1
3               do jio=0, cell_size(2,cio)-1
4                   iseek=(cell_low(1,cio) +
5      $                    PROBLEM_SIZE*((cell_low(2,cio)+jio) +
6      $                    PROBLEM_SIZE*((cell_low(3,cio)+kio) +
7      $                    PROBLEM_SIZE*idump)))
8
9                   do ix=0,cell_size(1,cio)-1
10                      write(99, rec=iseek+ix+1)
11     $                      u(1,ix, jio,kio,cio),
12     $                      u(2,ix, jio,kio,cio),
13     $                      u(3,ix, jio,kio,cio),
14     $                      u(4,ix, jio,kio,cio),
15     $                      u(5,ix, jio,kio,cio)
16                  enddo
17              enddo
18          enddo
19      enddo
```

and accumulating the time steps is just a loop over the above with reads replacing the writes.

NPB BT MPI-I/O

Pieces of NPB I/O with MPI-I/O:
Setup also includes a custom datatype for the element storage,

```
1       call MPI_File_open(comm_solve,
2      $           filenm,
3      $           MPI_MODE_WRONLY+MPI_MODE_CREATE,
4      $           MPI_INFO_NULL, fp, ierr)
5
6       if (ierr .ne. MPI_SUCCESS) then
7                 print *, 'Error opening file'
8                 stop
9       endif
10
11       call MPI_File_set_view(fp, iseek, element,
12     $           combined_ftype, 'native', info, ierr)
13
14      if (ierr .ne. MPI_SUCCESS) then
15                print *, 'Error setting file view'
16                stop
17      endif
```

Time step output (`eltext` is the outcome of the `MPI_Type_extent` on the `element` datatype):

```
1       call MPI_File_write_at_all(fp, iseek, u,
2      $                           1, combined_btype, mstatus, ierr)
3       if (ierr .ne. MPI_SUCCESS) then
4           print *, 'Error writing to file'
5           stop
6       endif
7
8       call MPI_Type_size(combined_btype, iosize, ierr)
9       iseek = iseek + iosize/eltext
```

Note the use of the collective write with an explicit offset (iseek).

Accumulate norms:

```
1       call MPI_File_open(comm_solve,filenm,MPI_MODE_RDONLY,MPI_INFO_NULL,fp,ierr)
2       iseek = 0
3       call MPI_File_set_view(fp, iseek, element, combined_ftype,
4      $           'native', MPI_INFO_NULL, ierr)
5  c    clear the last time step
6       call clear_timestep
7  c    read back the time steps and accumulate norms
8       do m = 1, 5
9          xce_acc(m) = 0.d0
10      end do
11      do ii=0, idump-1
12
13        call MPI_File_read_at_all(fp,iseek,u,1,combined_btype,mstatus,ierr)
14        if (ierr .ne. MPI_SUCCESS) then
15           print *, 'Error reading back file'
16           call MPI_File_close(fp, ierr)
17           stop
18        endif
19
20        if (node .eq. root) print *, 'Reading data set ', ii+1
21        call error_norm(xce_single)
22        do m = 1, 5
23           xce_acc(m) = xce_acc(m) + xce_single(m)
24        end do
25
26        call MPI_Type_size(combined_btype, iosize, ierr)
27        iseek = iseek + iosize/eltext
28 ...
29      call MPI_File_close(fp, ierr)
```

Note the use of the collective read with an explicit offset (iseek).

High-level Libraries

Purpose of high-level (parallel) I/O libraries:

- Match storage to computational domain (many dimensional datasets, variables, and attributes)
- (hopefully) self-describing and structured format
- Map to middleware (e.g., MPI-IO) interface and its characteristics (e.g., collective I/O)
- Optimization potential beyond middleware (knowledge of datasets may suggest optimal chunk sizes, attribute caching)
- Portability - ability to move application and data between platforms

HDF5

HDF (Hierarchical Data Format) was originally developed in 1988 at the National Center for Supercomputing Applications (NCSA) at the U. of Illinois:

- Comprehensive and complex (parallel) I/O library
- Hierarchical data organization in single file
- Typed, multidimensional array storage
- C, C++, and Fortran interfaces
- Portable data format
- Optional compression (not in parallel I/O mode)
- Data reordering (chunking)
- Noncontiguous I/O (memory and file) with hyperslabs

HDF5 Files

HDF5 files consist of groups, datasets, and attributes:

- Groups are like directories, holding other groups and datasets
- Datasets hold an array of typed data
  - datatype describes the type (not an MPI datatype)
  - dataspace gives the dimensions of the array
- Attributes are small datasets associated with the file, a group, or another dataset:
  - Also have a datatype and dataspace
  - May only be accessed as a unit

HDF5 Tutorials and Documentation

We can not hope to cover HDF5 in any great detail here (not our focus anyway), but it is worth spending some time looking at the primary source of HDF5 documentation:

www.hdfgroup.org/HDF5

Note in particular the availability of tools to view and parse HDF5 formatted files (as well as the third party tools that support and use HDF5).

Writing HDF5 Files

Basic steps in writing HDF5 files:

1. Create file
2. Create group (if desired)
3. Define (>=1) dataspaces
4. Define datatypes
5. Create datasets
6. Write attributes
7. Write data
8. Close all objects

The actual API routines to utilize vary depending on whether you choose to use the low-level or one of the high-level APIs available in HDF5.

Simple Example

```
1  /* Creating and closing a dataset with HDF5 */
2  #include "hdf5.h"
3  #define FILE "dset.h5"
4
5  int main() {
6    hid_t       file_id, dataset_id, dataspace_id;  /* identifiers */
7    hsize_t     dims[2];
8    herr_t      status;
9
10 /* Create a new file using default properties. */
11    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
12
13 /* Create the data space for the dataset. */
14   dims[0] = 4;
15   dims[1] = 6;
16   dataspace_id = H5Screate_simple(2, dims, NULL);
17
18 /* Create the dataset. */
19   dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id,
20      H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
21
22 /* End access to the dataset and release resources used by it. */
23   status = H5Dclose(dataset_id);
24
25 /* Terminate access to the data space. */
26   status = H5Sclose(dataspace_id);
27
28 /* Close the file. */
29   status = H5Fclose(file_id);
30 }
```

Simple Example in Parallel HDF5

```
1  /*
2  *  This example creates an HDF5 file.
3  */
4  #include "hdf5.h"
5  #define H5FILE_NAME     "SDS_row.h5"
6
7  int main (int argc, char **argv)
8  {
9  /*
10 * HDF5 APIs definitions
11 */
12 hid_t       file_id;          /* file and dataset identifiers */
13 hid_t plist_id;         /* property list identifier( access template) */
14 herr_t status;
15
16 /*
17  * MPI variables
18 */
19
20 int mpi_size, mpi_rank;
21 MPI_Comm comm  = MPI_COMM_WORLD;
22 MPI_Info info  = MPI_INFO_NULL;
23
24  /*
25 * Initialize MPI
26 */
27
28 MPI_Init(&argc, &argv);
29 MPI_Comm_size(comm, &mpi_size);
30 MPI_Comm_rank(comm, &mpi_rank);
```

```
31  /*
32  * Set up file access property list with parallel I/O access
33  */
34
35  plist_id = H5Pcreate(H5P_FILE_ACCESS);
36  H5Pset_fapl_mpio(plist_id, comm, info);
37
38  /*
39   * Create a new file collectively.
40  */
41
42  file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
43
44  /*
45   * Close property list.
46   */
47
48  H5Pclose(plist_id);
49
50  /*
51   * Close the file.
52  */
53
54  H5Fclose(file_id);
55  MPI_Finalize();
56
57  return 0;
58  }
```

netCDF

netCDF is a worthy data format in its own right, and is worth mentioning in this context since it has recently (2008) been extended to support HDF5 (as of netCDF-4), including parallel support through MPI-IO. Unfortunately there is some confusion due to another project that independently added parallel support to netCDF. The two are not compatible either in format or API:

- http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial/Parallel.html
- http://trac.mcs.anl.gov/projects/parallel-netcdf

Guidelines

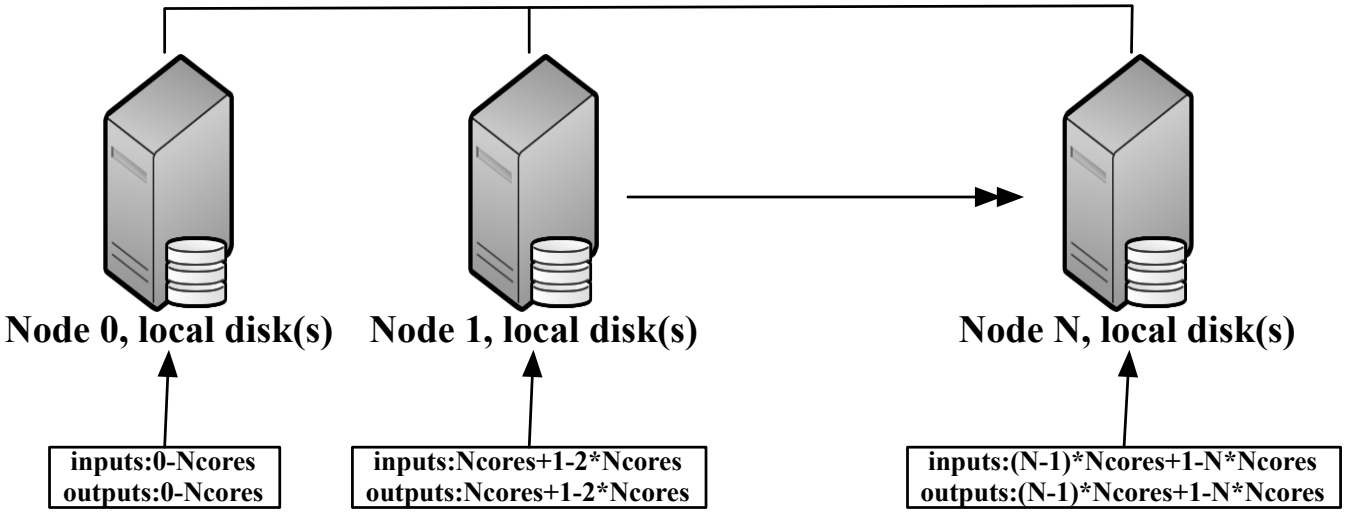
What to do for parallel I/O support in a new application? That really depends on what you need:

- Large-scale I/O requirements (and that can be in file count as well as file size, the equivalent of communication latency versus bandwidth) force you to use parallel I/O - go for a portable high-level library (HDF5, netCDF). Most large-scale simulation codes fall into this category, various visualization packages have the ability to digest these file formats.
- Smaller scale requirements do not force you to use truly parallel I/O, but you will still have to decide for a parallel application if you will serialize the I/O, or leverage an available library.
- The hardware is mostly out of our control (unless we are part of the decision-making process leading to its purchase/architecture).

How to Decide, Part II

You can use “embarrassingly parallel” I/O to measure the impact that the shared storage system is having on a given application.

- Each node typically has a local I/O subsystem (disk(s), perhaps with RAID) for a scratch space - most quantum chemistry codes make very use of such space for temporary integral files that are evaluated once locally and then used repeatedly. Why?
- The local I/O solution is really the **only scalable** choice - it can be sustained over any number of nodes as long as there is no need for shared files (or metadata) between them (see following Figure).
- If you can do so, using this **local** surrogate decouples you from any bottleneck in the shared filesystems, and can give you a useful upper bound on I/O performance. In fact, if you can then use a different process/procedure for gathering and integrating the resulting files, you can stick with this approach and use regular POSIX I/O.



Scaling local storage out in an “embarrassingly parallel” fashion; aggregated we have N times the I/O bandwidth of each node (with roughly the same I/O latency).

Embarrassingly Parallel I/O

Frequently you get better performance out of this local approach to I/O even on a smaller scale:

- Shared filesystems are shared amongst tens to hundreds of users, and the contention for them can be significant.
- The network connection to the shared storage is often slower than the local I/O subsystem, so even on small parallel runs local I/O can be faster (that ignores time needed to accumulate and integrate the scattered files if needed).