I/O -(Parallel and Otherwise)
on a
Scyld Beowulf Cluster
Outline

- What is Parallel I/O?
- Do I need it?
- Parallel Filesystems
- MPI I/O and ROMIO
- Other interfaces to PVFS
- Example striping schemes
- Out-of-core programming
Parallel I/O in Data Parallel Programs

- Each task reads a distinct partition of the input data and writes a distinct partition of the output data.
- Each task reads its partition in parallel
- Data is distributed to the slave nodes
- Each task computes output data from input data
- Each task writes its partition in parallel
Since each node in a Beowulf has its own disk, making the same files available on each node can be problematic.

Three filesystem options:

- Local
- Remote (eg. NFS)
- Parallel (eg. PVFS)
Filesystems (cont.)

- Local - Use storage on each node's disk
  - Relatively high performance
  - Each node has different filesystem
  - Shared datafiles must be copied to each node
  - No synchronization
  - Most useful for temporary/scratch files accessed only by copy of program running on single node
Filesystems (cont.)

• Remote - Share a single disk among all nodes
  - Every node sees same filesystem
  - Synchronization mechanisms manage changes
  - "Traditional" UNIX approach
  - Relatively low performance
  - Doesn't scale well; server becomes bottleneck in large systems
  - Simplest solution for small clusters, reading/writing small files
Filesystems (cont.)

- Parallel - Stripe files across multiple disks on multiple nodes
  - PVFS - the Parallel Virtual File System
    - Developed at Clemson University
    - Distributed in the new release of Scyld OS
  - Relatively high performance
  - Each node sees same filesystem
  - Works best for I/O intensive applications
  - Not a good solution for small files
  - Certain slave nodes are designated I/O nodes, local disks used to store pieces of filesystem
Using File Systems

- Local File Systems
  - EXT2
- Network File Systems
  - NFS, AFS
- Parallel File Systems
  - PVFS, ROMIO
- I/O Libraries
  - HDF, Panda
Accessing Local File Systems

• I/O system calls on compute nodes are executed on the compute node

• File systems on the slave can be made available to tasks running there and accessed as on any Linux system

• Recommended programming model does not assume that a task will run on a specific node
  – Best used for temporary storage
  – Access permissions may be a problem
Accessing Network File Systems

- Network file systems such as NFS and AFS can be mounted by slave nodes
- Provides a shared storage space for home directories, parameter files, smaller data files
- Can be a performance problem when many slaves access a shared file system at once
- Performance problems can be severe for a very large number of nodes (100+)
- Otherwise, works like local file systems
Accesing Parallel File Systems

- Distribute file data among many I/O nodes (servers), potentially every node in the system.
- Typically not so good for small files, but very good for large data files.
- Should provide good performance even for a very large degree of sharing.
- Critical for scalability in applications with large I/O demands.
- Particularly good for data parallel model.
Issues in Parallel I/O

• Physical distribution of data to I/O nodes interacts with logical distribution of the I/O requests to affect performance
  – Logical record sizes should be considered in physical distribution
  – I/O buffer sizes depend on physical distribution and number of tasks

• Performance is best with rather large requests
  – Buffering should be used to get requests of 1MB or more, depending on the size of the system
I/O Libraries

• May make I/O simpler for certain applications
  – Multidimensional data sets
  – Special data formats
  – Consistent access to shared data
  – "Out-of-core" computation

• May hide some details of parallel file systems
  – Partitioning

• May provide access to special features
  – Caching, buffering, asynchronous I/O, performance
MPI-IO

- Common file operations
  - MPI_File_open();
  - MPI_File_close();
  - MPI_File_read();
  - MPI_File_write();
  - MPI_File_read_at();
  - MPI_File_write_at();
  - MPI_File_read_shared();
  - MPI_File_write_shared();

- Open, close are collective. The rest have collective counterparts; add _all
**MPI_File_open**

MPI_File_open(
    MPI_Comm comm,
    char *filename,
    int amode,
    MPI_Info info,
    MPI_File *fh);

- Collective operation on comm
- **amode** similar to UNIX file mode; a few extra MPI possibilities
MPI_File_close

MPI_File_close(
    MPI_File *fh
);

C A E F F
File Views

• File views supported
  - MPI_File_set_view();

• Essentially, a file view allows you to change your program's treatment of a file as simply a stream of bytes, to viewing the file as a set of MPI_Datatype and displacements.

• Arguments to set view are similar to the arguments for creating derived datatypes
MPI_File_read

MPI_File_read(
    MPI_File fh,
    void *buf,
    int count,
    MPI_Datatype datatype,
    MPI_Status *status
);
MPI_File_read_at

MPI_File_read_at(
    MPI_File fh,
    MPI_Offset offset,
    void *buf,
    int count,
    MPI_Datatype datatype,
    MPI_Status *status
);

- **MPI_File_read_at_all()** is the collective version
Non-Blocking I/O

MPI_File_iread();
MPI_File_iwrite();
MPI_File_iread_at();
MPI_File_iwrite_at();
MPI_File_iread_shared();
MPI_File_iwrite_shared();
MPI_File_iread

MPI_File_iread(
    MPI_File fh,
    void *buf,
    int count,
    MPI_Datatype datatype,
    MPI_Request *request
);

- Request structure can be queried to determine if the operation is complete
Collective access

- The “shared” routines use a collective file pointer
- Collective routines also provided to allow each task to read/write a specific chunk of the file:
  - MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype type, MPI_Status *st)
  - MPI_File_write_ordered()
  - MPI_File_seek_shared()
File Functions

- MPI_File_delete();
- MPI_File_set_size();
- MPI_File_preallocate();
- MPI_File_get_size();
- MPI_File_get_group();
- MPI_File_get_amode();
- MPI_File_set_info();
- MPI_File_get_info();
Example Application for Parallel I/O
# Parallel I/O Example

```c
#include <fcntl.h>
#define INSIZE 1024
#define OUTSIZE 1024
#define CHUNKS_PER_TASK 128
unsigned char inbuff[INSIZE]; // input buffer
unsigned char outbuff[OUTSIZE]; // output buffer
int size; // number of tasks
int rank; // unique task index number

main(int argc, char **argv)
{
    int if, of; // file descriptors
    int chunk; // chunk counter
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if = open("input_file.dat", O_RDONLY);
    of = open("output_file.dat", O_RDWR);
    for (chunk = 0; chunk < CHUNKS_PER_TASK; chunk++)
    {
        lseek(if, (rank + (chunk * size)) * INSIZE, SEEK_SET);
        read(if, buffer, INSIZE);
        process(inbuff, outbuff);
        lseek(fd, (rank + (chunk * size)) * OUTSIZE, SEEK_SET);
        write(fd, buffer, OUTSIZE);
    }
    close(if);
    close(of);
    MPI_Finalize();
}
```
What Are All These Names?

- **MPI** - Message Passing Interface Standard
  - Also known as MPI-1
- **MPI-2** - Extensions to MPI standard
  - I/O, RDMA, dynamic processes
- **MPI-IO** - I/O part of MPI-2 extensions
- **ROMIO** - Implementation of MPI-IO
  - Handles mapping MPI-IO calls into communication (MPI) and file I/O
- **PVFS** - Parallel Virtual File System
  - An implementation of a file system for Linux clusters
Parallel Virtual File System

- Parallel file system for Linux clusters
  - Global name space
  - Distributed file data
  - Builds on TCP, local file systems
- Mountable like NFS file systems
- User-level interface library (used by ROMIO)
- Tuned for high performance concurrent access
PVFS characteristics

**Capabilities**
- Consistent file name space across cluster
- User-defined striping of data
- Access via existing programs
- > 2Gbyte files (if kernel supports them)
- > 2Tbyte file systems
- 2.2. and 2.4 Linux kernel support

**Shortcomings**
- No redundancy
- No really, none.
- No locking
- Not production level (i.e. it crashes some times)
- Slow metadata operations
PVFS Components

- Two server types:
  - `mgr` - file manager, handles metadata for files
  - `iods` - I/O servers, store and retrieve file data
- Client-side library (libpvfs)
  - UNIX-like API
  - hides details of data transfer from application
- Kernel VFS module hooks into Linux
Data flow in PVFS operations

Atomically handled by mgr (eliminating need for locks)
- e.g. rename, chmod, unlink

Clients communicate directly with iods
- libpvfs reconstructs file region from received pieces
ROMIO MPI-IO Implementation

- Implementation of MPI-2 I/O specification
  - Operates on wide variety of platforms
  - Abstract Device Interface for I/O (ADIO) aids in porting to new file systems
  - Fortran and C bindings

- Successes
  - Adopted by industry (e.g. Compaq, HP, SGI)
  - Used at ASCI sites (e.g. LANL Blue Mountain)
Data Staging for Tiled Display

- Commodity components
  - projectors, PCs
- Provide very high resolution visualization
- Staging application splits frames into a tile stream for each visualization node
  - Uses MPI-IO to access data from PVFS file system
  - Streams of tiles are merged into movie files on visualization node
Splitting Movie Frames into Tiles

- Hundreds of frames make up a single movie
- Each frame is stored in its own file in PVFS
- Frame size is 2532x1408 pixels
- 3x2 display
- Tile size is 1024x768 pixels (overlapped)
Obtaining Highest Performance

- To make best use of PVFS:
  - Use MPI-IO (ROMIO) for data access
  - Use file views and datatypes
  - Take advantage of collectives
  - Use hints to optimize for your platform

- Simple, right :)?
Trivial MPI-IO Example

- Reading contiguous pieces with MPI-IO calls
  - Simplest, least powerful way to use MPI-IO
  - Easy to port from POSIX calls
  - Lots of I/O operations to get desired data

```c
MPI_File_open(comm, fname, MPI_MODE_RDONLY, MPI_INFO_NULL, &handle);

/* read tile data from one frame */
for (row = 0; row < 768; row++) {
    offset = row*row_size + tile_offset + header_size;
    MPI_File_read_at(handle, offset, buffer, 1024*3, MPI_BYTE, &status);
}

MPI_File_close(&handle);
```

(We’ll leave out the open and close calls from here on.)
Avoiding the VFS Layer

- UNIX calls go through VFS layer
- MPI-IO calls use PVFS library directly
- Significant performance gain

![Read Performance Graph](image-url)
Why Use File Views?

• Concisely describe noncontiguous regions in a file
  – Create datatype describing region
  – Assign “view” to open file handle

• Separate description of region from I/O operation
  – Datatype can be reused on subsequent calls

• Access these regions with a single operation
  – Single MPI read call requests all data
  – Provides opportunity for optimization of access in MPI-IO implementation…
Setting a File View

- Use MPI_Type_create_subarray() to define a datatype describing the data in the file

- Example for tile access (24-bit data):

  ```c
  MPI_Type_contiguous(3, MPI_BYTE, &rgbtype);
  frame_size[1] = 2532; /* frame width */
  frame_size[0] = 1408; /* frame height */
  tile_size[1] = 1024; /* tile width */
  tile_size[0] = 768; /* tile height */

  /* create datatype describing tile */
  MPI_Type_create_subarray(2, frame_size, tile_size,
                           tile_offset, MPI_ORDER_C, rgbtype, &tiletype);
  MPI_Type_commit(&tiletype);

  MPI_File_set_view(handle, header_size, rgbtype,
                    tiletype, "native", MPI_INFO_NULL);

  MPI_File_read(handle, buffer, buffer_size,
                rgbtype, &status);
  ```
Noncontiguous Access in ROMIO

- ROMIO performs “data sieving” to cut down number of I/O operations
- Uses large reads which grab multiple noncontiguous pieces
- Example, reading tile 1:
Data Sieving Performance

- Reduces I/O operations from 4600+ to 6
- 87% effective throughput improvement
- Reading 3 times as much data as necessary…
Collective I/O

- MPI-IO supports “collective” I/O calls (_all suffix)
- All processes call the same function at once
  - May vary parameters (to access different regions)
- More fully describe the access pattern as a whole
  - Explicitly define relationship between accesses
- Allow use of ROMIO aggregation optimizations
  - Flexibility in what processes interact with I/O servers
  - Fewer, larger I/O requests
Collective I/O Example

- Single line change:

```c
/* create datatype describing tile */
MPI_Type_create_subarray(2, frame_size, tile_size,
    tile_offset, MPI_ORDER_C, rgbtype, &tiletype);
MPI_Type_commit(&tiletype);

MPI_File_set_view(handle, header_size, rgbtype,
    tiletype, "native", MPI_INFO_NULL);

#if 0
MPI_File_read(handle, buffer, buffer_size,
    rgbtype, &status);
#endif

/* collective read */
MPI_File_read_all(handle, buffer, buffer_size,
    rgbtype, &status);```
Two-Phase Access

• ROMIO implements two-phase collective I/O
  – Data is read by clients in contiguous pieces (phase 1)
  – Data is redistributed to the correct client (phase 2)
• ROMIO applies two-phase when collective accesses overlap between processes
• More efficient I/O access than data sieving alone
Two-Phase Performance

Often a big win:

For us, 1/3 the speed of independent operations!!!

Not enough clients to stress I/O servers
Hints

• Controlling PVFS
  – striping_factor - size of “strips” on I/O servers
  – striping_unit - number of I/O servers to stripe across
  – start_iodevice - which I/O server to start with

• Controlling aggregation
  – cb_config_list - list of aggregators
  – cb_nodes - number of aggregators (upper bound)

• Tuning ROMIO optimizations
  – romio_cb_read, romio_cb_write - aggregation on/off
  – romio_ds_read, romio_ds_write - data sieving on/off
The Proof is in the Performance

- Final performance is almost 3 times VFS access!
- Hints allowed us to turn off two-phase, modify striping of data
Summary: Why Use MPI-IO?

- Better concurrent access model than POSIX one
  - Explicit list of processes accessing concurrently
  - More lax (but still very usable) consistency model
- More descriptive power in interface
  - Derived datatypes for concise, noncontiguous file and/or memory regions
  - Collective I/O functions
- Optimizations built into MPI-IO implementation
  - Noncontiguous access
  - Collective I/O (aggregation)
- Performance portability
Redundancy

• Two applications
  – Metadata - ensuring that the storage system maintains a consistent state
  – Data – ensuring that stored files remain available in the event of component failure

• Traditional RAID approaches are often applied to solve both problems at a block level

• Performance implications are quite serious
  – Back to locking again
  – Lots of extra I/O
Redundancy Requirements

• Metadata really should be redundantly stored at all times
• Users often don't need their data files to be redundantly stored at every moment in time
  – Want completed checkpoints to be redundantly stored
  – Want output from completed runs to be redundantly stored
• There is room for a separate policy for data redundancy
Lazy Data Redundancy

• What if we allowed users to specify when to enforce redundancy?
  – MPI_File_replicate()
  – Could pass a flag at open to force "always redundantly stored" mode

• Advantages
  – Provides underlying storage with an opportunity to optimize replica creation
  – Allows users to trade performance for reliability

• Questions
  – How does this propagate down in storage layers?
Out-of-Core Computation

• What to do when dataset is larger than physical memory? (Serial or parallel?)
  – Let OS deal with the problem
    • Virtual Memory
    • Lots of small (~4k) I/O accesses...
    • Very Inefficient
  – Programmer deals with problem
    • Can do fewer but larger accesses (match memory size)
    • A little more work...
    • This is “Out-of-core” (OOC) computation
OOC Example - Matrix Multiply

- Routines to put and get block of matrix:

```c
struct dblock {
    MPI_File fh;
    void *buffer;
    int gsize[2];
    int distr[2];
    int dargs[2];
    int psize[2];
};

get_dblock(struct dblock *blk, int r, int c){
    MPI_Status status;
    MPI_Datatype blocktype;
    MPI_Type_create_darray(blk->psize[0]*blk->psize[1],
                           (r*blk->psize[0])+c, 2, blk->gsize, blk->distr, blk->dargs,
                           blk->psize, MPI_ORDER_C, MPI_FLOAT, &blocktype);
    MPI_File_read(blk->fh, blk->buffer, 1, blocktype, &status);
}

put_dblock(struct dblock *blk, int r, int c){
    MPI_Status status;
```
OOC Example - Matrix Multiply

- Routines to put and get block of matrix:

```c
struct dblock {
    MPI_File fh;
    void *buffer;
    int gsize[2];
    int distr[2];
    int dargs[2];
    int psize[2];
};

get_dblock(struct dblock *blk, int r, int c){
    MPI_Status status;
    MPI_Datatype blocktype;
    MPI_Type_create_darray(blk->psize[0]*blk->psize[1],
                            (r*blk->psize[0])+c,2, blk->gsiz,
                            blk->distr, blk->dargs,
                            blk->psize, MPI_ORDER_C, MPI_FLOAT, &blocktype);
    MPI_File_read(blk->fh, blk->buffer, 1, blocktype, &status);
}
```
OOC Example - Matrix Multiply

- Routines to put and get block of matrix:

```
put_dblock(struct dblock *blk, int r, int c)
{
    MPI_Status status;
    MPI_Datatype blocktype;
    MPI_Type_create_darray(blk->psize[0]*blk->psize[1],
    (r*blk->psize[0])+c, 2, blk->gsize, blk->distr, blk->dargs, blk->psize,
    MPI_ORDER_C, MPI_FLOAT, &blocktype);
    MPI_File_write(blk->fh, blk->buffer, 1, blocktype, &status);
}
```
OOC Example - Matrix Multiply

- Routines to put and get block of matrix:

```c
put_dblock(struct dblock *blk, int r, int c)
{
    MPI_Status status;
    MPI_Datatype blocktype;
    MPI_Type_create_darray(blk->psize[0]*blk->psize[1],
                           (r*blk->psize[0])+c, 2, blk->gsizes, blk->dists,
                           blk->dargs, blk->psizes,
                           MPI_ORDER_C, MPI_FLOAT, &blocktype);
    MPI_File_write(blk->fh, blk->buffer, 1, blocktype,
                   &status);
}
```
OOC Example - Matrix Multiply

• Main routine to do multiply

// now do matrix multiply
for (i = rank*(szN/size); i < (rank+1)*(szN/size); i++)
{
    for (j = 0; j < szL; j++)
    {
        memset(c, 0, csz);
        for (k = 0; k < szM; k++)
        {
            get_dblock(&ablk,i,k);
            get_dblock(&bblk,k,j);
            mmulf(a, b, c_part, N/szN, M/szM, L/szL);
            maddf(c, c_part, csz);
        }
        put_dblock(&cblk,i,j);
    }
}
OOC Example - Matrix Multiply

- Multiply subroutines

```c
maddf(float *x, float *y, int size)
{
    while(size > 0){
        *x++ += *y++;
        size -= sizeof(float);
    }
}
mulf(float *x, float *y, float *z, int r, int s, int t)
{
    int i, j;
    for (i = 0; i < r; i++) {
        for (j = 0; j < t; j++) {
            rtcf(x, y+j, z++, s, t);
        }
        x += s;
    }
}
rtcf(float *row, float *col, float *res, int s, int t)
{
    *res = 0.0;
    while (s--){
        *res += *row * *col;
        row += 1;
        col += 1;
    }
}
```
OOC Example - Matrix Multiply

- Distribution code

```c
// Multiply matrix A times B into C
// A is an N x M matrix broken into szN x szM submatrices
// B is an M x L matrix broken into szM x szL submatrices
// C is an N x L matrix broken into szN x szL submatrices
// We assume szN is evenly divisible by size (number of tasks)
// We divide matrix C into size row-blocks of submatrices
ablk.gsize[0] = N;
ablk.gsize[1] = M;
ablk.distr[0] = MPI_DISTRIBUTE_BLOCK;
ablk.distr[1] = MPI_DISTRIBUTE_BLOCK;
ablk.dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
ablk.dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
ablk.psize[0] = szN;
ablk.psize[1] = szM;
```
How to get access to the grid

- [http://www.parl.clemson.edu/minigrid/](http://www.parl.clemson.edu/minigrid/)
- dstanzi@parl.clemson.edu
  Contact Dan Stanzione at the address above with any requests or problems
- Remote access to the system is through the secure shell utility "ssh" (no telnet, rsh, ftp, etc)
- ssh is freely available for all platforms (windows clients available on the web page).
  - ssh thymine.parl.clemson.edu
  - ssh -l <username> thymine.parl.clemson.edu