CS420 – Lecture 22

Marc Snir

Fall 2018

Quiz 4

Q1

```
int i, myid, ntasks;
int size = 100;
int *message = malloc(sizeof(int) * size);
int *receiveBuffer = malloc(sizeof(int) * size);
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
/* Initialize message */
for (i = 0; i < size; i++)    message[i] = myid + i;
MPI_Win_create(message, size*sizeof(int), sizeof(int), MPI_INFO_NULL,
          MPI_COMM_WORLD, &rma_window);
MPI_Win_fence(0, rma_window);
if (myid > 0) {
  MPI_Get(receiveBuffer, size, MPI_INT, myid - 1, 0, size, MPI_INT,
      rma_window);
}
MPI_Win_fence(0, rma_window);
if (myid == 7) {
  printf("Receiver: %d:", myid);
  for (int i=0; i<5; ++i) printf(" %d ", receiveBuffer[i]);
}
```

What is the output in if we run mpirun -np 10 a.out

- Process with rank r writes in its buffer message $r, r+1, \ldots, r+size$
- Process with rank r gets values written by process $r-1$ and prints first 5 values
- Process with rank 7 receives and prints $6, 7, 8, 9, 10$

```
int ntasks, my_id, irank;
int dims[2];       /* Dimensions of the grid */
int coords[2];     /* Coordinates in the grid */
int neighbors[4]; /* Neighbors in 2D grid */
int period[2] = {1, 1};
MPI_Comm comm2d;
MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
/* Determine the process grid (dims[0] x dims[1] = ntasks) */
dims[0] = 2; dims[1] = ntasks / dims[0];
/* Create the 2D Cartesian communicator */
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, period, 1, &comm2d);
/* Find out and store the ranks with which to perform halo exchange */
MPI_Cart_shift(comm2d, 0, 1, &neighbors[0], &neighbors[1]);
MPI_Cart_shift(comm2d, 1, 1, &neighbors[2], &neighbors[3]);
/* Find out and store also the Cartesian coordinates of a rank */
MPI_Cart_coords(comm2d, my_id, 2, coords);
for (irank = 0; irank < ntasks; irank++) {
```

```
  if (my_id == irank) {
   printf("%3i = %2i %2i neighbors=%3i %3i %3i %3i\n", my_id, coords[0],
     coords[1], neighbors[0], neighbors[1], neighbors[2], neighbors[3]);
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

What is the output in the following blank when running mpirun -np 8 a.out

- Program creates 2×4 cyclic Cartesian mesh $\begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{matrix}$
- `MPI_Cart_shift` returns rank of process one away up and down and process one away to the left and right
- Barrier in last loop ensures that each process writes in turn
- rank 0 process will output `0 0 0 neighbors 4 4 3 1`
- rank 1 process will output `1 0 1 neighbors 5 5 0 2`
- ...
- rank 7 process will output `7 1 3 neighbors 3 3 6 4`

Q3

Which MPI collective operation would you use to compute the transpose of a matrix that is distributed in a row-wise manner across P processes?
Answer

- MPI_Allreduce
- MPI_Scan
- MPI_Alltoall
- MPI_Reduce

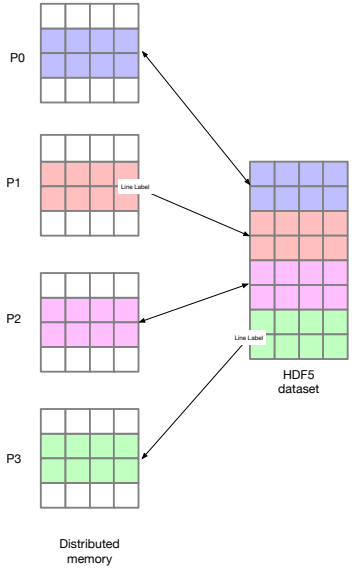
Q3

Which MPI collective operation would you use to compute the transpose of a matrix that is distributed in a row-wise manner across P processes?
Answer

- MPI_Allreduce
- MPI_Scan
- MPI_Alltoall
- MPI_Reduce

PHDF5

Example: scatter and gather



Input and output slices of Jacobi matrix from a dataset

```
#include "hdf5.h"
#include "stdlib.h"
#include "mpi.h"

void Jacobi(int N, int M, double a[M][N], double tolerance)
{...}

int main (int argc, char **argv)
{
```

```
/*
* HDF5 APIs definitions
*/
hid_t        file, jacobi_in,
      jacobi_out;        /* file and dataset identifiers */
hid_t        filespace, memspace;     /* file and memory dataspace identifiers */
hsize_t      dim[2];                  /* dataset dimensions */
int          *data;                   /* pointer to data buffer in  memory */
hsize_t    count[2];              /* hyperslab selection parameters */
hsize_t    offset[2];
hid_t    plist;                   /* property list identifier */
int          i, j;
herr_t   status;
```

```
/*
* MPI variables
*/
int mpi_size, mpi_rank;
/*
* Initialize MPI
*/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
MPI_Comm_rank(MOPI_COMM_WORLD, &mpi_rank);
/*
* Set up file access property list for parallel I/O access
*/
plist = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist, MPI_COMM_WORLD, MPI_INFO_NULL);
```

```
plist = H5Pcreate(H5P_FILE_ACCESS);
hid_t H5Pcreate(hid_t cls_id)
```

Create property list as an instance of the specified property list class (in our case, file access property list)

```
H5Pset_fapl_mpio(plist, comm, MPI_INFO_NULL);
H5Pset_fapl_mpio(hid_t fapl_id, MPI_Comm comm, MPI_Info info)
```

Stores the communicator to be used for parallel I/O and additional Info into the file access property list.

`H5P` prefixes functions that manipulate property lists

```
/*
* Open collectively the file  containing input dataset
*/
file = H5Fopen(Jacobi.h5, H5F_ACC_RDONLY, H5P_DEFAULT, plist);
/*
* Open the dataset and get its dataspace
*/
jacobi_in = H5Dopen(file,"jacobi_in", plist);
filespace = H5Dget_space(jacobi_in);
/*
* Get the dataspace dimensions
*/
H5Sget_simple_extent_dims(filespace, dim, NULL);
```

```
file = H5Fopen(Jacobi.h5, H5F_ACC_RDONLY, plist);
```

```
hid_t H5Fopen( const char *name, unsigned flags, hid_t fapl_id )
```

Open a file with specified name; specifies access mode, and property lists for access (parallel access).

```
filespace = H5Dget_space(jacobi_in);
```

```
hid_t H5Dget_space(hid_t dataset_id )
```

Get the dataspace describing the dataset argument

```
H5Sget_simple_extent_dims(filespace, dim, NULL);
```

```
int H5Sget_simple_extent_dims(hid_t space_id, hsize_t *dims, hsize_t
*maxdims)
```

Get dimensions and maximum dimensions of dataspace

```
/*
* Each process computes tile dimensions, creates dataset
* describing it and allocate space for it
*/
i = dim[0]/mpi_size;
j = dim[0]%mpi_size;
count[0] = mpi_rank<j? i+1 : i;
count[1] = dim[1];
data = (double *) malloc(sizeof(double)*(count[0]+2)*count[1]);
memspace = H5Screate_simple(2, count, NULL);
```

```
H5Screate_simple(2, count, NULL);
```

```
hid_t H5Screate_simple( int rank, const hsize_t * current_dims, const
hsize_t * maximum_dims )
```

Creates datapsace with rank, current dimensions and maximum dimensions as specified

```
/*
* Select hyperslab in the file.
*/
offset[0] = mpi_rank<j? mpi_rank*(i+1): npi_rank*i+j;
offset[1] = 0;
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, count, NULL);
```

```
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, count, NULL)
```

```
herr_t H5Sselect_hyperslab(hid_t space_id, H5S_seloper_t op, const hsize_t
*start, const hsize_t *stride, const hsize_t *count, const hsize_t *block )
```

Modifies specified dataspace using the specified operation(replace, add, OR, AND, ...), using start, stride, count and block arguments for the subarray

```
/*
* Create property list for collective dataset read.
*/
plist = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist, H5FD_MPIO_COLLECTIVE);
/*
* Processes read collectively input array; each process reads a slab.
*/
status = H5Dread(jacobi_in, H5T_NATIVE_DOUBLE, memspace, filespace,
plist, data);
```

```
plist = H5Pcreate(H5P_DATASET_XFER);
```

Create a property list for dataset read/write

```
H5Pset_dxpl_mpio(plist, H5FD_MPIO_COLLECTIVE);
herr_t H5Pset_dxpl_mpio( hid_t dxpl_id, H5FD_mpio_xfer_t xfer_mode)
```

Set the data transfer property list to specified transfer mode (`H5FD_MPIO_INDEPENDENT` or `H5FD_MPIO_COLLECTIVE`)

```
status = H5Dread(jacobi_in, H5T_NATIVE_INT, memspace, filespace, plist,
data)
herr_t H5Dread( hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
hid_t file_space_id, hid_t xfer_plist_id, const void * buf )
```

dataset_id: Identifier of the dataset to read from.

mem_type_id: Identifier of the memory datatype.

mem_space_id: Identifier of the memory dataspace.

file_space_id: Identifier of the dataspace in the file (in our case, slice of the file dataset)

xfer_plist_id: Identifier of a transfer property list for this I/O operation (in our case, specifies collective I/O)

buf: Start of buffer with data to be written to the file.

```
/*
* Collective execution of Jacobi
*/
 Jacobi(count[0], count[1], data[count[0]][count[1]], 0.00001);

/*
* Write back solution and close (free) all HDF5 objects
*/


MPI_Finalize();
return 0;
```

Data Analysis

Big Data – 5V's

Volume: Beyond what is handled by single machine or can be managed by conventional database. E.g., WWW

Velocity: How quickly data is updated (e.g., Internet traffic)

Variety: text, audio, video, sensor data...

Veracity: Data is noisy

Value: good data + good analysis generates value

Large scientific data

- Data generated by large experiments (e.g. LHC – 25 PB/year) – Volume, Velocity
- Data generated by large observations (e.g., DES – 2.5 TB/night; LSST 15 TB/night) – Volume, Velocity, Veracity
- X-Informatics (bioinformatics, cheminformatics, environmental informatics...) – aggregation of many distinct information sources (publications, images, observations...) — Volume, Velocity, Variety, Veracity, Value

Characteristics

- Data (often) does not fit in memory – Need to repeatedly load and store data – Key performance issue is to minimize number of passes over data.
- CPU performance is less critical – interpreted languages such as Python are often used
- Frameworks are used to orchestrate data movement, and to hide underlying architecture
- Most often done in cloud systems – where each node has own disk (better if data is read multiple times)

Big Data + Big Compute

There is an increasing convergence of big data and HPC:

- Deep Learning (training neural networks)
- Analysis of simulation outputs
- Comparison of simulation to observation or experiment

Hadoop

Apache open source framework for storing and analyzing big data. Contains:

Hadoop Distributed File System (HDFS) – distributed file system built atop clusters of commodity machines

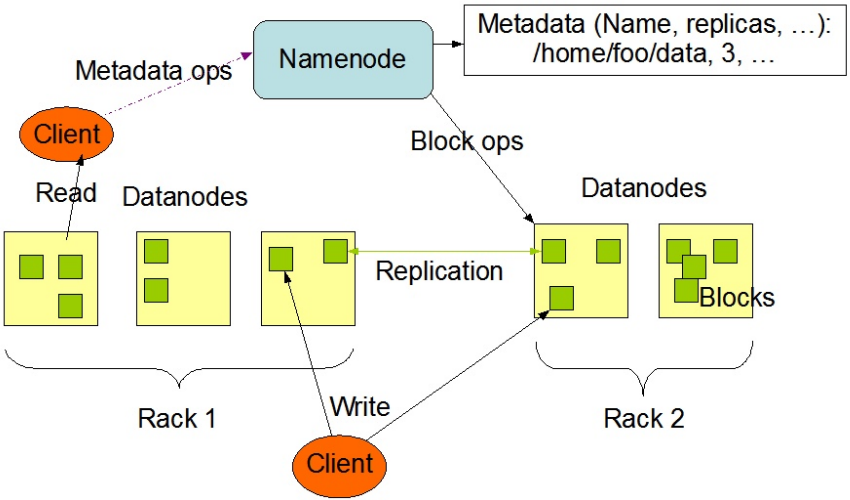
Hadoop MapReduce – an implementation of the MapReduce programming model.

Originally inspired by Google MapReduce and Google File System
Now used as part of larger packages, such as *Apache Spark*

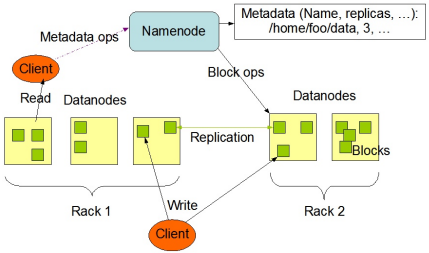
Big Data Storage – HDFS

- Unstructured data – arrays a la HDF5 are not that useful
- Focus on streaming I/O – reading or writing sequentially large amounts of data (as distinct from random I/O)
- Relaxes POSIX semantics – no need for strict coherence
- Build atop large number of local file systems on commodity nodes
- Use replication to provide fault recovery
- Move computation to data, whenever possible

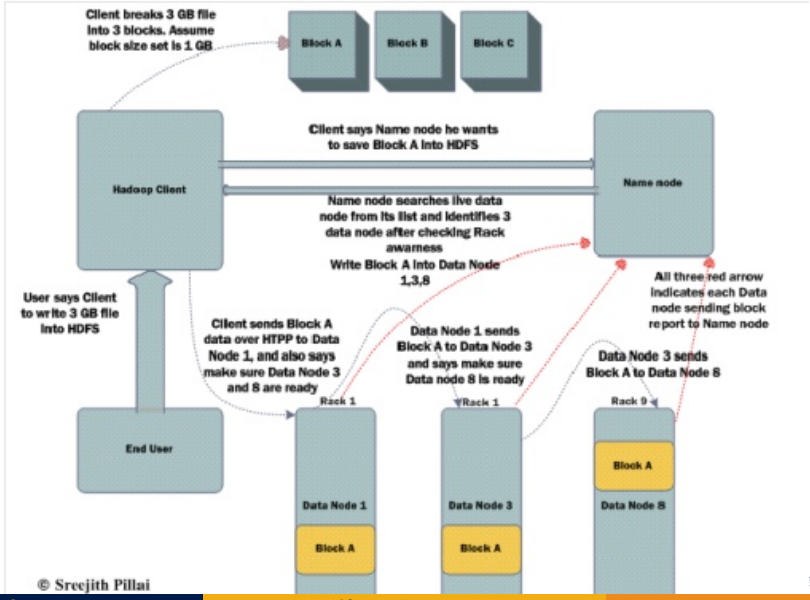
HDFS Architecture



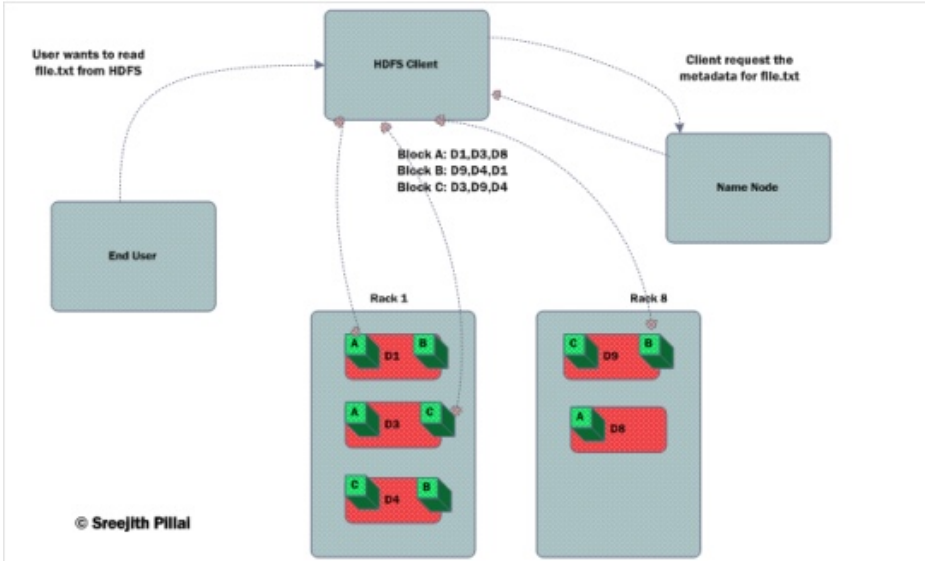
- *namenode* (aka master node) stores metadata and provides external interface
  - May have a backup
- *datanodes* store data blocks (64 MB typical)
  - blocks may be replicated (2 or 3 times – on different nodes/racks)
- Data moves directly from client to datanode and vice-versa



HDFS write

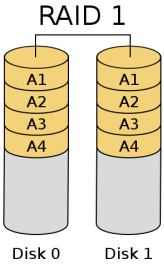

Client breaks 3 GB file into 3 blocks. Assume block size set is 1 GB
Block A
Block B
Block C
Hadoop Client
Client says Name node he wants to save Block A into HDFS
Name node
Name node searches live data node from its list and identifies 3 data node after checking Rack awarness
Write Block A into Data Node 1,3,8
All three red arrow indicates each Data node sending block report to Name node
User says Client to write 3 GB file into HDFS
Client sends Block A data over HTPP to Data Node 1, and also says make sure Data Node 3 and 8 are ready
Data Node 1 sends Block A to Data Node 3 and says make sure Data node 8 is ready
Data Node 3 sends Block A to Data Node 8
End User
Rack 1
Rack 1
Rack 9
Data Node 1
Block A
Data Node 3
Block A
Block A
Data Node 8
© Sreejith Pillai

HDFS read

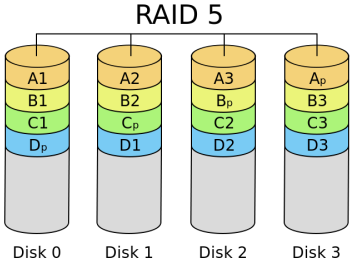

User wants to read file.txt from HDFS
HDFS Client
Client request the metadata for file.txt
Block A: D1,D3,D8
Block B: D9,D4,D1
Block C: D3,D9,D4
Name Node
End User
Rack 1
Rack 8
A
B
D1
A
C
D3
C
B
D4
C
B
D9
A
D8
© Sreejith Pillai

Comparison to PFS

- Relaxed Posix I/O semantics (no coherence)
- Replication (RAID 1) vs. RAID 5 and higher



- 2/3 times as much storage
- Read: 1 disk access
- Write: 2-3 disk accesses (need not be simultaneous)



- 1/3 (or less) as much storage
- Read: 1 disk access
- Write: 3 disk accesses (in one transaction)
- Requires more coordination (typically in firmware)