Data Models and I/O

ATPESC 2018

Rob Latham
Math and Computer Science Division
Argonne National Laboratory
Q Center, St. Charles, IL (USA)
August 3, 2018
Plan of attack

• Bottom-up tour of I/O interfaces
  – POSIX routines called by MPI-IO implementations
  – Parallel-NetCDF routines build on top of MPI-IO

• Simple toy programs
  – Refining example several times throughout day
  – We can apply these lessons to your own code in evening session

• Demonstrating some tools for understanding what’s going on

• “Game of Life” for your reference
Hands on materials

• Code for this …
  – Simple array I/O

• … and other sections available on our gitlab site:
  – Game of Life I/O
  – Darshan
  – Burst buffers
  – Globus

• https://xgitlab.cels.anl.gov/ATPESC-IO/hands-on

• I’m going to give you a few minutes to try each hands-on. Can continue working in evening session if you need more time.
Operating on Arrays

• Arrays show up in many scientific applications
  – Matrix operations
  – Particle maps
  – Regions of space
  – Time series
  – Images

• Probably your real application more complicated but an array or two (or more) is in there somewhere, I’d wager.
Decomposition

- How do we physically access locally parts of a logically larger distributed array in parallel…
  - Piecewise?
  - Chunks?
  - Rows?

- Largely dictated by application algorithm needs
  - E.g. volume rendering math requires chunks not rows.

- Choice impacts memory and I/O performance
Supporting Checkpoint/Restart

• For long-running applications, the cautious user checkpoints
• Application-level checkpoint involves the application saving its own state
  – With a bit of extra effort, can be portable
• A canonical representation is preferred
  – Independent of number of processes
• Restarting is then possible
  – Canonical representation aids restarting with a different number of processes
• Also eases data analysis (when using same output)
Defining a Checkpoint

• Need enough to restart
  – Header information
    • Size of problem (e.g. matrix dimensions)
    • Description of environment (e.g. input parameters)
  – Program state
    • Should represent the global (canonical) view of the data

• Ideally stored in a convenient container
  – Single “thing” (file, object, keyval store...)

• If all processes checkpoint at once, naturally a parallel, collective operation
HANDS-ON 1: simple data descriptions

• Consider an application that operates on a 2-d array of integers.
  1. Write code declaring a 2-d array of integers
     • Probably want to allocate on heap, not stack
     • Later steps will be easier if you make it a single allocation
  2. Define a data structure describing the experiment
     • E.g. C struct with row, column, iteration

• Use whatever language you like…
  – … but Phil and I can only be helpful if you use C

• Source “setup-env.sh” to load necessary modules
HANDS-ON 1 solutions

C struct holding metadata

```c
typedef struct {
    int row;
    int col;
    int iter;
} science;
```

Do this: index into a single big allocation

```c
int *array;
array = malloc(XDIM*YDIM*sizeof(*array));
```

Don’t do this: N allocations will be slower and harder to describe

/* not MPI-friendly: describing this memory region will require
   * a more complicated data type description */
```c
int **annoying;
annoying = malloc(YDIM*sizeof(*array));
for (int i=0; i<YDIM; i++)
    annoying[i] = malloc(XDIM*sizeof(*array));
```
POSIX I/O

• POSIX is the IEEE Portable Operating System Interface for Computing Environments

• “POSIX defines a standard way for an application program to obtain basic services from the operating system”
  – Mechanism almost all serial applications use to perform I/O

• POSIX was created when a single computer owned its own file system
Deficiencies in serial interfaces

POSIX:

```c
fd = open("some_file", O_WRONLY|O_CREAT,
         S_IRUSR|S_IWUSR);
ret = write(fd, w_data, nbytes);
ret = lseek(fd, 0, SEEK_SET);
ret = read(fd, r_data, nbytes);
ret = close(fd);
```

FORTRAN:

```fortran
OPEN(10, FILE='some_file', &
     STATUS="replace", &
     ACCESS="direct", RECL=16);
WRITE(10, REC=2) 15324
CLOSE(10);
```

- Typical (serial) I/O calls seen in applications
- No notion of other processors
- Primitive (if any) data description methods
- Tuning limited to open flags
- No mechanism for data portability
  - Fortran not even portable between compilers
HANDS-ON 2: simple I/O

• We haven’t talked about MPI-IO or I/O libraries, but we can still checkpoint.
  – Serial I/O, not parallel

• Implement “write_data”
  – Will create file and fill in data
  – Prototype:
    • `int write_data(char *filename)`
  – Use system calls (open(), write(), close()), not “stdio” calls (fopen(), fwrite(), fclose()): will map more closely to MPI-IO later
  – How will you know it worked?
  – We are going to repeatedly revise “write_data” (and later “read_data”) with each exercise
    • Software engineering: hide details
• Submit to the ‘training’ queue

• I’ve provided a ‘submit.sh’ shell script
  – qsub -q training submit.sh <program> [filename]
    • If you don’t give [filename], then ‘testfile’ used.

• Which Theta file system to use?
  – Tried to make scripts do right thing by default
  – Please don’t use the NFS-mounted home directory
  – submit.sh should already point you to the right lustre directory
Solution fragments:

```c
int write_data(char *filename)
{
    science data = {
        .row = YDIM,
        .col = XDIM,
        .iter = 1
    };

    int *array;
    int fd;
    int ret=0;

    array = buffer_create(0, XDIM, YDIM);

    fd = open(filename, O_CREAT|O_WRONLY,
                S_IRUSR|S_IWUSR);
    ret = write(fd, &data, sizeof(data));
    ret = write(fd, array, XDIM*YDIM*sizeof(int));
    ret = close(fd);

    return ret;
}
```

Reading a binary file: “cat” won’t work. Could write a c program to read. Several utilities available. I like ‘od’: historically it only did an “octal dump”. The (t)ype argument can select (d)ecimal

```
% od -td testfile
0000000  1  5  1  0
0000020  1  2  3  4
0000040
```

Reading a binary file: “cat” won’t work. Could write a c program to read. Several utilities available. I like ‘od’: historically it only did an “octal dump”. The (t)ype argument can select (d)ecimal

```
% od -td testfile
0000000  1  5  1  0
0000020  1  2  3  4
0000040
```
HANDS-ON 3: send-to-master

• Parallel program, but serial I/O
  1. Write_data() should take an MPI Communicator
  2. Call MPI_Init() and MPI_Finalize()
  3. Use MPI_Gather to collect all data onto rank 0:

• Only rank 0 does I/O; writes header and all array information
• What’s good about send-to-master? What’s bad?
Solution fragments: MPI_Gather, write larger data from rank 0

MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &nprocs);
/* every process creates its own buffer */
array = buffer_create(rank, XDIM, YDIM);

/* and then sends it to rank 0 */
int *buffer =
    malloc(XDIM*YDIM*nprocs*sizeof(int));

MPI_CHECK(MPI_Gather(
    /* sender (buffer,count,type) tuple */
    array, XDIM*YDIM, MPI_INT,
    /* receiver tuple */
    buffer, XDIM*YDIM, MPI_INT,
    /* who gathers and across which context */
    0, comm));
**Solution fragments: writing from rank 0**

```c
if (rank == 0) {
    /* looks like serial with more data */
    ...
    /* writing a global array, not just our local piece of it */
    data.row = YDIM*nprocs;
    data.col = XDIM;
    data.iter = 1;
    // omitted code
    ret = write(fd, &data, sizeof(data));
    ret = write(fd, buffer, XDIM*YDIM*nprocs*sizeof(int));
    ret = close(fd);
    return ret;
}
```
Other questions:

• Lots of machines (your laptop; Theta) represent integers as 32 bit little endian. What if you ran this code on Mira?

• We wrote row-wise. What if you wanted to write a column of data?

• What impact would a header have on data layout? Are there other options?
Understanding I/O

• **Instrumentation:**
  – What do we measure?
  – How much overhead is acceptable and when?

• **Analysis:**
  – How do we correlate data and extract actionable information?
  – Can we identify the root cause of performance problems?

• **Impact:**
  – Develop best practices and tune applications
  – Improve system software
  – Design and procure better systems
What is Darshan?

**Darshan** is a scalable HPC I/O characterization tool. It captures an accurate but concise picture of *application* I/O behavior with minimum overhead.

- No code changes, easy to use
  - Negligible performance impact: just “leave it on”
  - Enabled by default at ALCF, NERSC, NCSA, and KAUST
  - Installed and available for case by case use at many other sites
- Produces a *summary* of I/O activity for each job, including:
  - Counters for file access operations
  - Time stamps and cumulative timers for key operations
  - Histograms of access, stride, datatype, and extent sizes

Project began in 2008, first public software release and deployment in 2009
Darshan design principles

• The Darshan run time library is inserted at link time (for static executables) or at run time (for dynamic executables)

• Transparent wrappers for I/O functions collect per-file statistics

• Statistics are stored in bounded memory at each rank

• At shutdown time:
  – Collective reduction to merge shared file records
  – Parallel compression
  – Collective write to a single log file

• No communication or storage operations until shutdown

• Command-line tools are used to post-process log files
JOB analysis example

Example: Darshan-job-summary.pl produces a 3-page PDF file summarizing various aspects of I/O performance.

- Estimated performance
- Percentage of runtime in I/O
- Access size histogram
- Access type histograms
- File usage
SYSTEM analysis example

- With a sufficient archive of performance statistics, we can develop heuristics to detect anomalous behavior

  - This example highlights large jobs that spent a disproportionate amount of time managing file metadata rather than performing raw data transfer
  - Worst offender spent 99% of I/O time in open/close/stat/seek
  - This identification process is not yet automated; alerts/triggers are needed in future work for greater impact

Example of heuristics applied to a population of production jobs on the Hopper system in 2013:

<table>
<thead>
<tr>
<th>JOBS IDENTIFIED USING METADATA RATIO METRIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thresholds</td>
</tr>
<tr>
<td>meta_time / nprocs &gt; 30 s</td>
</tr>
<tr>
<td>nprocs ≥ 192</td>
</tr>
<tr>
<td>metadata_ratio ≥ 25%</td>
</tr>
<tr>
<td>Total jobs analyzed</td>
</tr>
<tr>
<td>261,890</td>
</tr>
<tr>
<td>Jobs matching metric</td>
</tr>
<tr>
<td>252</td>
</tr>
<tr>
<td>Unique users matching metric</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>Largest single-job metadata ratio</td>
</tr>
<tr>
<td>&gt; 99%</td>
</tr>
</tbody>
</table>

\[
\sum_{n=1}^{nfiles} \frac{metadata\_time}{metadata\_time + IO\_time}
\]

Typical deployment and usage

• Darshan usage on Mira, Cetus, Vesta, Theta, Cori, or Edison, abridged:
  – Run your job
  – If the job calls MPI_Finalize(), log will be stored in `DARSHAN_LOG_DIR/month/day/`
    • If your job does not call MPI_Finalize, you cannot use Darshan. Check out Tau.
  – Theta: `/lus/theta-fs0/logs/darshan/theta`
  – Use tools (next slides) to interpret log
• On Titan: “module load darshan” first
• More details:
  – https://www.alcf.anl.gov/user-guides/darshan
Generating job summaries

• Run job and find its log file:

```
pca@cori07:/working/other/nersc-darshan-seminar-2017> sbatch tor-shared.sh
Submitted batch job 5598961
pca@cori07:/working/other/nersc-darshan-seminar-2017> ls /global/cscratch1/sd/darshanlogs/2017/6/29 | grep 5598961
pca@cori07:~> ls /global/cscratch1/sd/darshanlogs/2017/6/29/*carns*
```

• Copy log files to save, generate PDF summaries:

```
pca@cori12:/Working/other/nersc-darshan-seminar-2017/logs> ls
pca@cori12:/Working/other/nersc-darshan-seminar-2017/logs> module load latex
pca@cori12:/Working/other/nersc-darshan-seminar-2017/logs> darshan-job-summary.pl pca@cori07:~> ls /global/cscratch1/sd/darshanlogs/2017/6/29/*carns*
pca@cori12:/Working/other/nersc-darshan-seminar-2017/logs> module load latex
pca@cori12:/Working/other/nersc-darshan-seminar-2017/logs> darshan-job-summary.pl
```
HANDS-ON 4: introduction to Darshan

1. Find the darshan log for the last exercise
2. View the raw counters with “darshan-parser”
3. Generate a report
   – You might have to transfer PDF locally to view
4. Find the darshan log for the exercise #2
   – Hint: you can’t!
I/O benchmarking challenges

- **Variability**
  - Storage systems shared, mechanical

- **Caching**
  - Watch out for “speed of light” violations

- **Ganging**
  - Be sure you are timing what you think you are timing
I/O benchmarking: variability

• Silicon (e.g. Read from DRAM, multiply 100 integers) pretty stable
  – E.g. easy to observe register, L1, L2, memory, swap behavior

• Write to disk… less stable
  – How many users are also writing? How full is disk?

• I/O experiments cannot be short, one-offs
  – Ideal: run each experiment cfg a dozen times, sized to run for about a minute
  – Reality: supercomputer time is precious

• Try out the variance example in hands-on repository
I/O benchmarking: caching

- Caching at every layer of storage
  - Disk drive, Raid controller, Server RAM, Compute node SSD
- Storage expensive; vendors don’t give stuff away
  - If spec says “240 GB/sec”, you ain’t getting 250 GB/sec
I/O benchmarking: ganging

• Fast-finisher problem
  – Maybe a caching or aggregation layer resulted in less work for one process

• Staggered-start problem
  – Probably want all processes writing/reading at once

• variance code example
  – MPI_Barrier() before timing
  – MPI_Reduce() to find maximum time
Bonus topic: “Game of Life” I/O

• Next stepping stone between toy array i/o and full application
• More sophisticated use of MPI datatypes, communication
  – “ghost cell” optimization heavily used in nearest-neighbor pattern
• Using “duplicated communicator” to separate library, application communication
• Also demonstrates a way to link different approaches
Rules for Life (you’ve probably seen this before)

• Matrix values $A(i,j)$ initialized to 1 (live) or 0 (dead)
• In each iteration, $A(i,j)$ is set to
  – 1 (live) if either
    • the sum of the values of its 8 neighbors is 3, or
    • the value was already 1 and the sum of its 8 neighbors is 2 or 3
  – 0 (dead) otherwise

All code examples in this tutorial can be found in hands-on repo:
xgitlab.cels.anl.gov/ATPESC-IO/hands-on
Decomposition and Boundary Regions

- Decompose 2d array into rows, shared across processes
- In order to calculate next state of cells in edge rows, need data from adjacent rows
- Need to communicate these regions at each step
Life Checkpoint/Restart API

• Define an interface for checkpoint/restart for the row-block distributed Life code

• Five functions:
  – MLIFEIO_Init
  – MLIFEIO_Finalize
  – MLIFEIO_Checkpoint
  – MLIFEIO_Can_restart
  – MLIFEIO_Restart

• All functions are collective
  – i.e., all processes must make the call

• We can implement API for different back-end formats
  – Insulate main code from I/O details:
  – back-end also makes good spot for tuning
Life Checkpoint

- MLIFEIO_Checkpoint(char *prefix,
  int **matrix,
  int rows,
  int cols,
  int iter,
  MPI_Info info);

- Prefix is used to set filename
- Matrix is a reference to the data to store
- Rows, cols, and iter describe the data (header)
- Info is used for tuning purposes
Life stdout “checkpoint”

- The first implementation is one that simply prints out the “checkpoint” in an easy-to-read format
- MPI standard does not specify that all stdout will be collected in any particular way
  - Pass data back to rank 0 for printing
  - Portable!
  - Not scalable, but ok for the purpose of stdout
stdio Life Checkpoint Code Walkthrough

• Points to observe:
  – All processes call checkpoint routine
    • Collective I/O from the viewpoint of the program
  – Interface describes the *global* array
  – Output is independent of the number of processes

See mlife-io-stdout.c pp. 1-3 for code example.
/* SLIDE: stdio Life Checkpoint Code Walkthrough */
/* -*- Mode: C; c-basic-offset:4 ; -*- */
/* (C) 2004 by University of Chicago. */
/*           See COPYRIGHT in top-level directory. */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#include "mlife.h"
#include "mlife-io.h"

/* stdout implementation of checkpoint (no restart) for MPI Life */
* Data output in matrix order: spaces represent dead cells, *
* ''s represent live ones. */

static int MLIFEIO_Type_create_rowblk (int **matrix, int myrows,
                                 int cols,
                                 MPI_Datatype *newtype);

void MLIFEIO_Row_print (int *data, int cols, int rownr);

void MLIFEIO_msleep (int msec);

static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
/* SLIDE: stdio Life Checkpoint Code Walkthrough */

int MLIFEIO_Init (MPI_Comm comm)
{
    int err;

    err = MPI_Comm_dup(comm, &mlifeio_comm);

    return err;
}

int MLIFEIO_Finalize (void)
{
    int err;

    err = MPI_Comm_free(&mlifeio_comm);

    return err;
}
/* SLIDE: Life stdout "checkpoint" */
/* MLIFEIO_Checkpoint
Parameters:
prefix - prefix of file to hold checkpoint (ignored)
matrix - data values
rows  - number of rows in matrix
cols  - number of columns in matrix
iter  - iteration number of checkpoint
info  - hints for I/O (ignored)

Returns MPI_SUCCESS on success, MPI error code on error.
*/

int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
                        int cols, int iter, MPI_Info info)
{
    int err = MPI_SUCCESS, rank, nprocs, myrows, myoffset;
    MPI_Datatype type;

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
/* SLIDE: Describing Data */
if (rank != 0) {
    /* send all data to rank 0 */
    MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
    MPI_Type_commit(&type);
    err = MPI_Send(MPI_BOTTOM, 1, type, 0, 1, mlifeio_comm);
    MPI_Type_free(&type);
} else {
    int i, procrows, totrows;
    printf("\033[H\033[2J# Iteration %d
", iter);
    /* print rank 0 data first */
    for (i=1; i < myrows+1; i++) {
        MLIFEIO_Row_print(&matrix[i][1], cols, i);
    }
    totrows = myrows;
/* SLIDE: Describing Data */
/* receive and print others’ data */
for (i=1; i < nprocs; i++) {
    int j, *data;
    procrows = MLIFE_myrows(rows, i, nprocs);
data = (int *) malloc(procrows * cols * sizeof(int));
err = MPI_Recv(data, procrows * cols, MPI_INT, i, 1,
               mlifeio_comm, MPI_STATUS_IGNORE);
    for (j=0; j < procrows; j++) {
        MLIFEIO_Row_print(&data[j * cols], cols,
totrows + j + 1);
    }
totrows += procrows;
free(data);
}
MLIFEIO_msleep(250); /* give time to see the results */
return err;
}
Describing Data

- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI_Type_vector
    
    ```c
    MPI_Type_vector(count = myrows, 
                    blklen = cols, stride = cols+2, MPI_INT, &vectype);
    ```
  - Second type gets memory offset right (allowing use of MPI_BOTTOM in 
    MPI_File_write_all)
    
    ```c
    MPI_Type_hindexed(count = 1, len = 1, 
                      disp = &matrix[1][1], vectype, &type);
    ```

See mlife-io-stdout.c pp. 4-6 for code example.
/* SLIDE: Describing Data */

/* MLIFEIO_Type_create_rowblk */

* Creates a MPI_Datatype describing the block of rows of data
* for the local process, not including the surrounding boundary
* cells.

* Note: This implementation assumes that the data for matrix is
* allocated as one large contiguous block!

*/

static int MLIFEIO_Type_create_rowblk (int **matrix, int myrows,
int cols,
MPI_Datatype *newtype)
{
int err, len;
MPI_Datatype vectype;
MPI_Aint disp;

/* since our data is in one block, access is very regular! */
err = MPI_Type_vector(myrows, cols, cols+2, MPI_INT,
&vectype);
if (err != MPI_SUCCESS) return err;

/* wrap the vector in a type starting at the right offset */
len = 1;
MPI_Address(&matrix[1][1], &disp);
err = MPI_Type_hindexed(1, &len, &disp, vectype, newtype);
MPI_Type_free(&vectype); /* decrement reference count */
146:     return err;
147: }
148:
149: static void MLIFEIO_Row_print(int *data, int cols, int rownr)
150: {
151:     int i;
152:     printf("%3d: ", rownr);
153:     for (i=0; i < cols; i++) {
154:         printf("%c", (data[i] == BORN) ? '*' : ' ');
155:     }
156:     printf("\n");
157: }
158:
159: int MLIFEIO_Can_restart(void)
160: {
161:     return 0;
162: }
163:
164: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
165:                     int cols, int iter, MPI_Info info)
166: {
167:     return MPI_ERR_IO;
168: }
Next steps: thinking about I/O interfaces for parallel programming (MPI-IO)