WE START WITH YES.

BUILDING AN I/O API: GAME OF LIFE CASE STUDY

ROB LATHAM
PHILP CARNS
Argonne National Laboratory
robl@mcs.anl.gov
carns@mcs.anl.gov

August 11, 2016
St. Charles IL
RULES FOR LIFE (YOU SAW THIS LAST WEEK)

- 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-2016
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
SUPPORTING CHECKPOINT/RESTART

- For long-running applications, the cautious user checkpoints
- Application-level checkpoint involves the application saving its own state
  - 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 file!

- If all processes checkpoint at once, naturally a parallel, collective operation
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
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);
static void MLIFEIO_Row_print(int *data, int cols, int rownr);
static 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\n", iter);
    /* print rank 0 data first */
    for (i=1; i < myrows+1; i++) {
        MLIFEIO_Row_print(&matrix[i][1], cols, i);
    }
    totrows = myrows;
92: /* SLIDE: Describing Data */
93:     /* receive and print others' data */
94:     for (i=1; i < nprocs; i++) {
95:         int j, *data;
96:         
97:         procrows = MLIFE_myrows(rows, i, nprocs);
98:         data = (int *) malloc(procrows * cols * sizeof(int));
99:         
100:         err = MPI_Recv(data, procrows * cols, MPI_INT, i, 1,
101:                          mlifeio_comm, MPI_STATUS_IGNORE);
102:         
103:         for (j=0; j < procrows; j++) {
104:             MLIFEIO_Row_print(&data[j * cols], cols,
105:                                totrows + j + 1);
106:         }
107:         totrows += procrows;
108:         
109:         free(data);
110:     }
111: }
112: 
113: MLIFEIO_msleep(250); /* give time to see the results */
114: 
115: return err;
116: }
DESCRIBING DATA

- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI_Type_vector
    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)
    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 */
return err;
}

static void MLIFEIO_Row_print(int *data, int cols, int rownr)
{
    int i;
    printf("%3d: ", rownr);
    for (i=0; i < cols; i++) {
        printf("%c", (data[i] == BORN) ? '*' : ' ');
    }
    printf("\n");
}

int MLIFEIO_Can_restart(void)
{
    return 0;
}

int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
                     int cols, int iter, MPI_Info info)
{
    return MPI_ERR_IO;
}
PARALLELING OUR I/O API
The stdio checkpoint routine works but is not parallel
- One process is responsible for all I/O
- Wouldn’t want to use this approach for real

How can we get the full benefit of a parallel file system?
- We first look at how parallel I/O works in MPI
- We then implement a fully parallel checkpoint routine

MPI is a good setting for parallel I/O
- Writing is like sending and reading is like receiving
- Any parallel I/O system will need:
  • collective operations
  • user-defined datatypes to describe both memory and file layout
  • communicators to separate application-level message passing from I/O-related message passing
  • non-blocking operations
- i.e., lots of MPI-like machinery
COLLECTIVE I/O

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - Framework for I/O optimizations at the MPI-IO layer
  - e.g., two-phase I/O

Small individual requests → Large collective access

Diagram showing small individual requests and large collective access.
COLLECTIVE MPI I/O FUNCTIONS

- Not going to go through the MPI-IO API in excruciating detail
  - Can talk during hands-on

- **MPI_File_write_at_all**, etc.
  - `_all` indicates that all processes in the group specified by the communicator passed to `MPI_File_open` will call this function
  - `_at` indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate “seek” call

- Each process specifies only its own access information
  - the argument list is the same as for the non-collective functions
  - OK to participate with zero data
    - All processes must call a collective
    - Process providing zero data might participate anyway
MPI-IO LIFE CHECKPOINT CODE WALKTHROUGH

- Points to observe:
  - Use of a user-defined MPI datatype to handle the local array
  - Use of MPI_Offset for the offset into the file
    - “Automatically” supports files larger than 2GB if the underlying file system supports large files
  - Collective I/O calls
    - Extra data on process 0

See mlife-io-mpiio.c pp. 1-2 for code example.
DATA LAYOUT IN MPI-IO CHECKPOINT FILE

Note: We store the matrix in global, canonical order with no ghost cells.

See mlife-io-mpiio.c pp. 1-9 for code example.
LIFE MPI-IO CHECKPOINT/RESTART

- We can map our collective checkpoint directly to a single collective MPI-IO file write: `MPI_File_write_at_all`
  - Process 0 writes a little extra (the header)

- On restart, two steps are performed:
  - Everyone reads the number of rows and columns from the header in the file with `MPI_File_read_at_all`
    - Sometimes faster to read individually and bcast (see later example)
  - If they match those in current run, a second collective call used to read the actual data
    - Number of processors can be different

See `mlife-io-mpiio.c` pp. 3-6 for code example.
/* SLIDE: Life MPI-IO Checkpoint/Restart */

int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
                        int cols, int iter, MPI_Info info)
{
    int err;
    int amode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
                MPI_MODE_UNIQUE_OPEN;
    int rank, nprocs;
    int myrows, myoffset;
    MPI_File fh;
    MPI_Datatype type;
    MPI_Offset myfileoffset;
    char filename[64];

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

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

    snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
    err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
    if (err != MPI_SUCCESS) {
        fprintf(stderr, "Error opening %s.
", filename);
        return err;
    }

    /* Additional code... */
/* SLIDE: Life MPI-IO Checkpoint/Restart */

if (rank == 0) {
    MLIFEIO_Type_create_hdr_rowblk(matrix, myrows, &rows,
        &cols, &iter, &type);
    myfileoffset = 0;
}

else {
    MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
    myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
}

MPI_Type_commit(&type);

err = MPI_File_write_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
    type, MPI_STATUS_IGNORE);

MPI_Type_free(&type);

err = MPI_File_close(&fh);

return err;
105: /* SLIDE: Life MPI-IO Checkpoint/Restart */
106: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
107:                  int cols, int iter, MPI_Info info)
108: {
109:     int err, gErr;
110:     int amode = MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN;
111:     int rank, nprocs;
112:     int myrows, myoffset;
113:     int buf[3]; /* rows, cols, iteration */
114:     MPI_File fh;
115:     MPI_Datatype type;
116:     MPI_Offset myfileoffset;
117:     char filename[64];
118:     MPI_Comm_size(mlifeio_comm, &nprocs);
119:     MPI_Comm_rank(mlifeio_comm, &rank);
120:     myrows   = MLIFE_myrows(rows, rank, nprocs);
121:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
122:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
123:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
124:     if (err != MPI_SUCCESS) return err;
125:     /* check that rows and cols match */
126:     err = MPI_File_read_at_all(fh, 0, buf, 3, MPI_INT,
127:                                 MPI_STATUS_IGNORE);
128:     if (err != MPI_SUCCESS) return err;
129: ...
130:     /* check that rows and cols match */
131:     err = MPI_File_read_at_all(fh, 0, buf, 3, MPI_INT,
132:                                 MPI_STATUS_IGNORE);
/* SLIDE: Life MPI-IO Checkpoint/R restart */
/* Have all process check that nothing went wrong */
MPI_Allreduce(&err, &gErr, 1, MPI_INT, MPI_MAX, mlifeio_comm);
if (gErr || buf[0] != rows || buf[1] != cols) {
    if (rank == 0) fprintf(stderr, "restart failed.\n");
    return MPI_ERR_OTHER;
}
MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
MPI_Type_commit(&type);
err = MPI_File_read_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
                          type, MPI_STATUS_IGNORE);
MPI_Type_free(&type);
err = MPI_File_close(&fh);
return err;
}
DESCRIBING HEADER AND DATA

- Data is described just as before
- Create a struct wrapped around this to describe the header as well:
  - no. of rows
  - no. of columns
  - Iteration no.
  - data (using previous type)

See mlife-io-mpiio.c pp. 7 for code example.
/* SLIDE: Describing Header and Data */

/* MLIFEIO_Type_create_hdr_rowblk */

* Used by process zero to create a type that describes both
* the header data for a checkpoint and its contribution to
* the stored matrix.

* Parameters:
* matrix - pointer to the matrix, including boundaries
* myrows - number of rows held locally
* rows_p - pointer to # of rows in matrix (so we can get its
*         address for use in the type description)
* cols_p - pointer to # of cols in matrix
* iter_p - pointer to iteration #
* newtype - pointer to location to store new type ref.

*/

static int MLIFEIO_Type_create_hdr_rowblk (int **matrix,
                                          int myrows,
                                          int *rows_p,
                                          int *cols_p,
                                          int *iter_p,
                                          MPI_Datatype *newtype)
{
    int err;
    int lens[4] = { 1, 1, 1, 1 };
    MPI_Aint disps[4];
    MPI_Datatype types[4];
    MPI_Datatype rowblk;
/* SLIDE: Describing Header and Data */

MLIFEIO_Type_create_rowblk(matrix, myrows, *cols_p, &rowblk);

MPI_Address(rows_p, &disps[0]);
MPI_Address(cols_p, &disps[1]);
MPI_Address(iter_p, &disps[2]);
disps[3] = (MPI_Aint) MPI_BOTTOM;
types[0] = MPI_INT;
types[1] = MPI_INT;
types[2] = MPI_INT;
types[3] = rowblk;

#if defined(MPI_VERSION) && MPI_VERSION >= 2
err = MPI_Type_create_struct(3, lens, disps, types, newtype);
#else
err = MPI_Type_struct(3, lens, disps, types, newtype);
#endif

MPI_Type_free(&rowblk);

return err;
}

return err;
MPI-IO TAKEAWAY

- Sometimes it makes sense to build a custom library that uses MPI-IO (or maybe even MPI + POSIX) to write a custom format
  - e.g., a data format for your domain already exists, need parallel API

- We’ve only touched on the API here
  - There is support for data that is noncontiguous in file and memory
  - There are independent calls that allow processes to operate without coordination

- In general we suggest using data model libraries
  - They do more for you
  - Performance can be competitive