Building an I/O API: game of life case study

Presented to
ATPESC 2017 Participants

Rob Latham, Phil Carns
Math and Computer Science Division
Argonne National Laboratory
Q Center, St. Charles, IL (USA)
Date 08/04/2017
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-2017
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 “thing” (file, object, keyval store...)

• 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
  – 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 stdio
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;

28: 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);
72: /* SLIDE: Describing Data */
73:     if (rank != 0) {
74:         /* send all data to rank 0 */
75:             
76:             MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
77:             MPI_Type_commit(&type);
78:             err = MPI_Send(MPI_BOTTOM, 1, type, 0, 1, mlifeio_comm);
79:             MPI_Type_free(&type);
80:         }
81:     } else {
82:         int i, procrows, totrows;
83:         
84:         printf("\033[H\033[2J# Iteration %d\n", iter);
85:         
86:         /* print rank 0 data first */
87:         for (i=1; i < myrows+1; i++) {
88:             MLIFEIO_Row_print(&matrix[i][1], cols, i);
89:         }
90:         totrows = myrows;
91:         

/* 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`
    ```
    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 */
146:     return err;
148: }
150: static void MLIFEIO_Row_print(int *data, int cols, int rownr)
151: {
152:     int i;
153:     printf("%3d: ", rownr);
154:     for (i=0; i < cols; i++) {
155:         printf("%c", (data[i] == BORN) ? '*' : ' ');
157:     }
158:     printf("\n");
160: }
161: int MLIFEIO_Can_restart(void)
162: {
164:     return 0;
166: }
166: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
167:                     int cols, int iter, MPI_Info info)
168: {
170:     return MPI_ERR_IO;
Parallelizing our I/O API
Parallel I/O and MPI

• 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 transformations/optimizations at the MPI-IO layer
  - Discussed these earlier today
  - e.g., two-phase I/O

Small individual requests | 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 behind the scenes 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.
Note2: this is our own custom layout. Naïve readers will have no idea what to do with this!

See mlife-io-mpiio.c pp. 1-9 for code example.
/* SLIDE: MPI-IO 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 <mpi.h>
#include "mlife-io.h"

/* MPI-IO implementation of checkpoint and restart for MPI Life
* Data stored in matrix order, with a header consisting of three
* integers: matrix size in rows and columns, and iteration no.
* Each checkpoint is stored in its own file.
*/

static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
                                      int cols,
                                      MPI_Datatype *newtype);
static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
                                           int myrows,
                                           int *rows_p,
                                           int *cols_p,
                                           int *iter_p,
                                           MPI_Datatype *newtype);
/* SLIDE: MPI-IO Life Checkpoint Code Walkthrough */
static MPI_Comm mlifeio_comm = MPI_COMM_NULL;

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;
}

int MLIFEIO_Can_restart(void)
{
    return 1;
}
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/Rerstart */

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;
    }

• Precise ‘amode’ might let library optimize
• Collective open with our duped comm
• “large enough” MPI_Offset
/* SLIDE: Life MPI-IO Checkpoint/Rerstart */

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:   /* Learn from me: check your error codes! */

/* SLIDE: Life MPI-IO Checkpoint/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);
erm = 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;
}
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