ATPESC 2016

Argonne NATIONAL LABORATORY

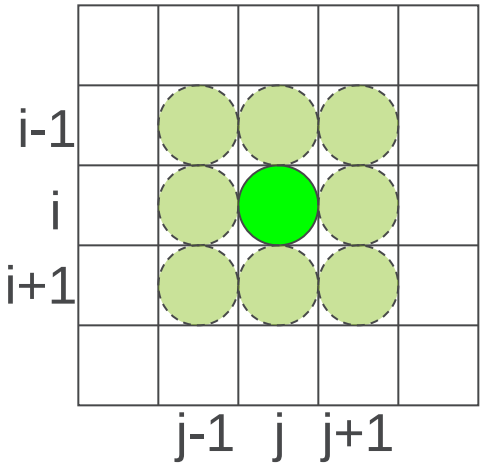
BUILDING AN I/O API: GAME OF LIFE CASE STUDY

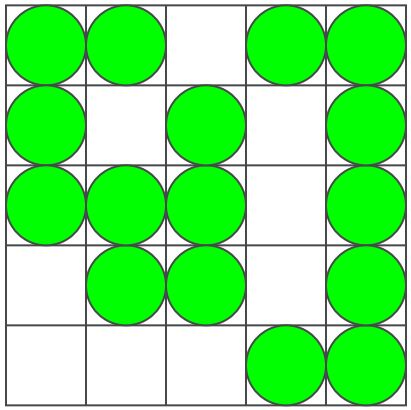
ROB LATHAM
PHILP CARNS
Argonne National Laboratory
robl@mcs.anl.gov
carns@mcs.anl.gov

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



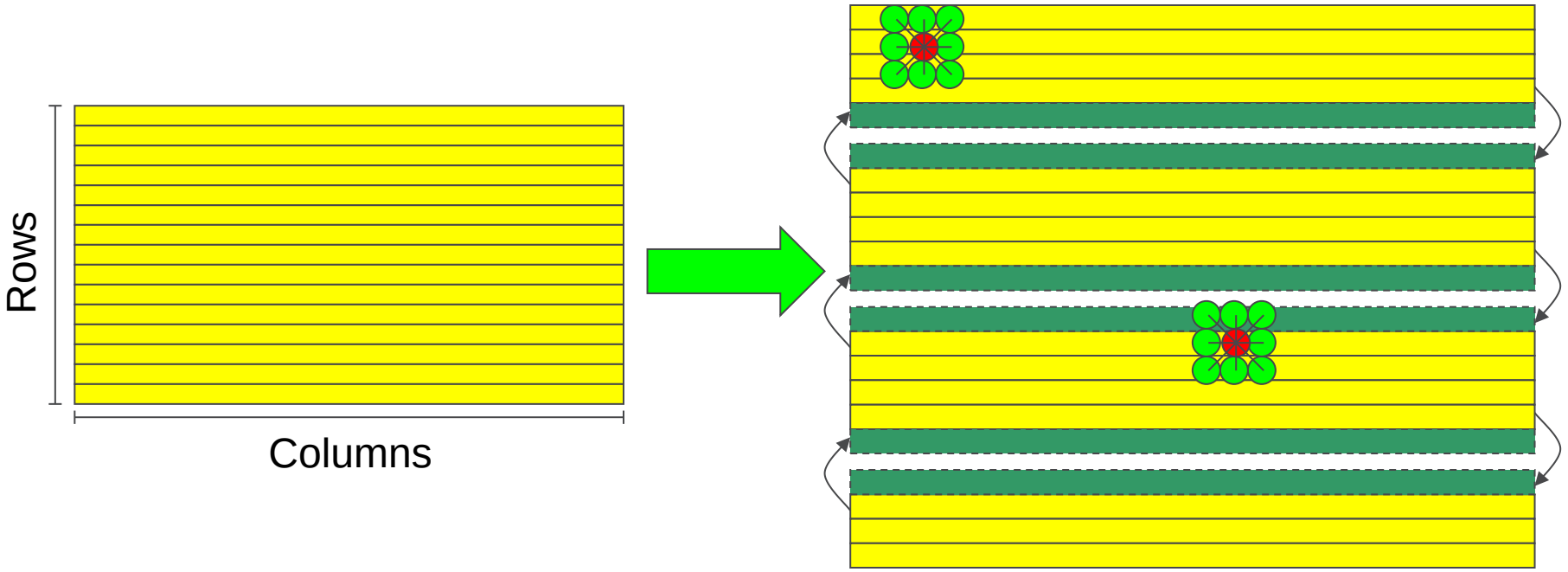


All code examples in this tutorial can be found in hands-on repo:

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

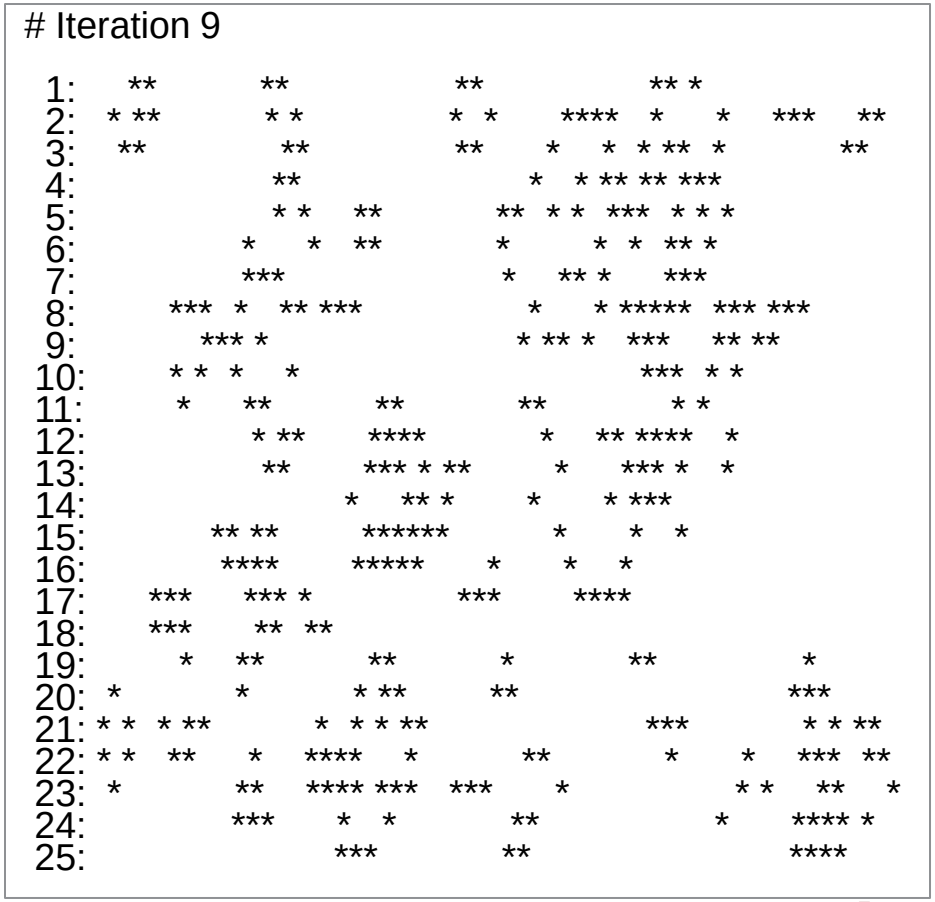
LIFE CHECKPOINT

```
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 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.

```
 1: /* SLIDE: stdio Life Checkpoint Code Walkthrough */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *  (C) 2004 by University of Chicago.
 5:  *      See COPYRIGHT in top-level directory.
 6:  */
 7:
 8: #include <stdio.h>
 9: #include <stdlib.h>
10: #include <unistd.h>
11:
12: #include <mpi.h>
13:
14: #include "mlife.h"
15: #include "mlife-io.h"
16:
17: /* stdout implementation of checkpoint (no restart) for MPI Life
18:  *
19:  * Data output in matrix order: spaces represent dead cells,
20:  * '*'s represent live ones.
21:  */
22: static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
23:                                       int cols,
24:                                       MPI_Datatype *newtype);
25: static void MLIFEIO_Row_print(int *data, int cols, int rownr);
26: static void MLIFEIO_msleep(int msec);
27:
28: static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
```

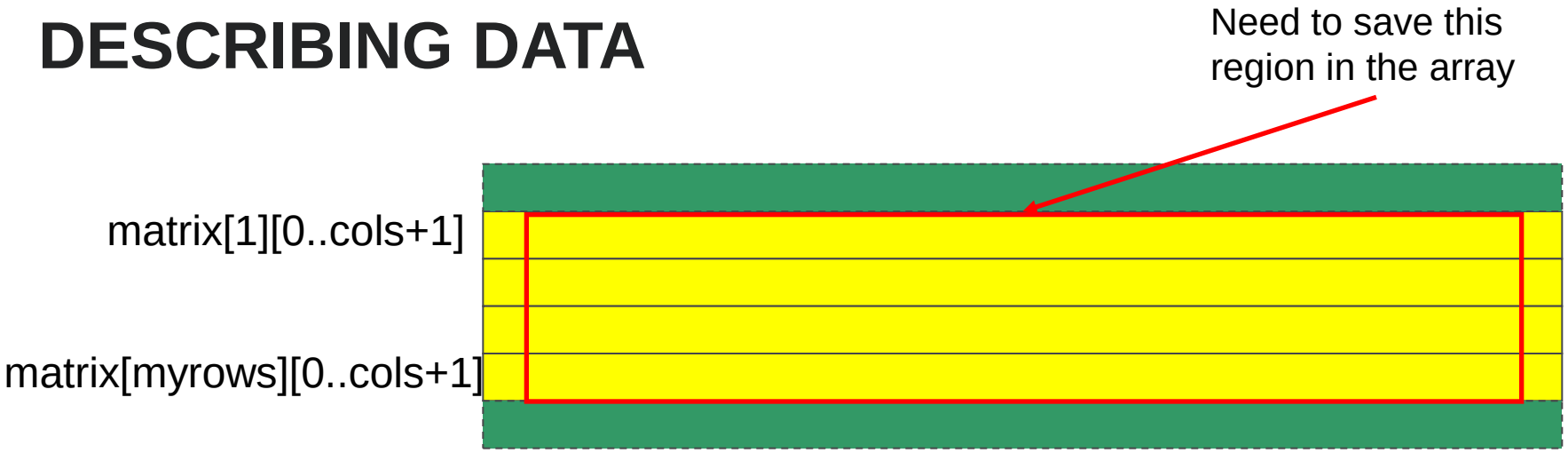
```
29: /* SLIDE: stdio Life Checkpoint Code Walkthrough */
30: int MLIFEIO_Init(MPI_Comm comm)
31: {
32:     int err;
33:
34:     err = MPI_Comm_dup(comm, &mlifeio_comm);
35:
36:     return err;
37: }
38:
39: int MLIFEIO_Finalize(void)
40: {
41:     int err;
42:
43:     err = MPI_Comm_free(&mlifeio_comm);
44:
45:     return err;
46: }
```

```
47: /* SLIDE: Life stdout "checkpoint" */
48: /* MLIFEIO_Checkpoint
49:  *
50:  * Parameters:
51:  * prefix - prefix of file to hold checkpoint (ignored)
52:  * matrix - data values
53:  * rows   - number of rows in matrix
54:  * cols   - number of columns in matrix
55:  * iter   - iteration number of checkpoint
56:  * info   - hints for I/O (ignored)
57:  *
58:  * Returns MPI_SUCCESS on success, MPI error code on error.
59:  */
60: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
61:                        int cols, int iter, MPI_Info info)
62: {
63:     int err = MPI_SUCCESS, rank, nprocs, myrows, myoffset;
64:     MPI_Datatype type;
65:
66:     MPI_Comm_size(mlifeio_comm, &nprocs);
67:     MPI_Comm_rank(mlifeio_comm, &rank);
68:
69:     myrows   = MLIFE_myrows(rows, rank, nprocs);
70:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
71:
```

```
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:
```

```
 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.

```
117: /* SLIDE: Describing Data */
118: /* MLIFEIO_Type_create_rowblk
119:  *
120:  * Creates a MPI_Datatype describing the block of rows of data
121:  * for the local process, not including the surrounding boundary
122:  * cells.
123:  *
124:  * Note: This implementation assumes that the data for matrix is
125:  *       allocated as one large contiguous block!
126:  */
127: static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
128:                                       int cols,
129:                                       MPI_Datatype *newtype)
130: {
131:     int err, len;
132:     MPI_Datatype vectype;
133:     MPI_Aint disp;
134:
135:     /* since our data is in one block, access is very regular! */
136:     err = MPI_Type_vector(myrows, cols, cols+2, MPI_INT,
137:                           &vectype);
138:     if (err != MPI_SUCCESS) return err;
139:
140:     /* wrap the vector in a type starting at the right offset */
141:     len = 1;
142:     MPI_Address(&matrix[1][1], &disp);
143:     err = MPI_Type_hindexed(1, &len, &disp, vectype, newtype);
144:
145:     MPI_Type_free(&vectype); /* decrement reference count */
```

```
146:
147:     return err;
148: }
149:
150: static void MLIFEIO_Row_print(int *data, int cols, int rownr)
151: {
152:     int i;
153:
154:     printf("%3d: ", rownr);
155:     for (i=0; i < cols; i++) {
156:         printf("%c", (data[i] == BORN) ? '*' : ' ');
157:     }
158:     printf("\n");
159: }
160:
161: int MLIFEIO_Can_restart(void)
162: {
163:     return 0;
164: }
165:
166: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
167:                     int cols, int iter, MPI_Info info)
168: {
169:     return MPI_ERR_IO;
170: }
```

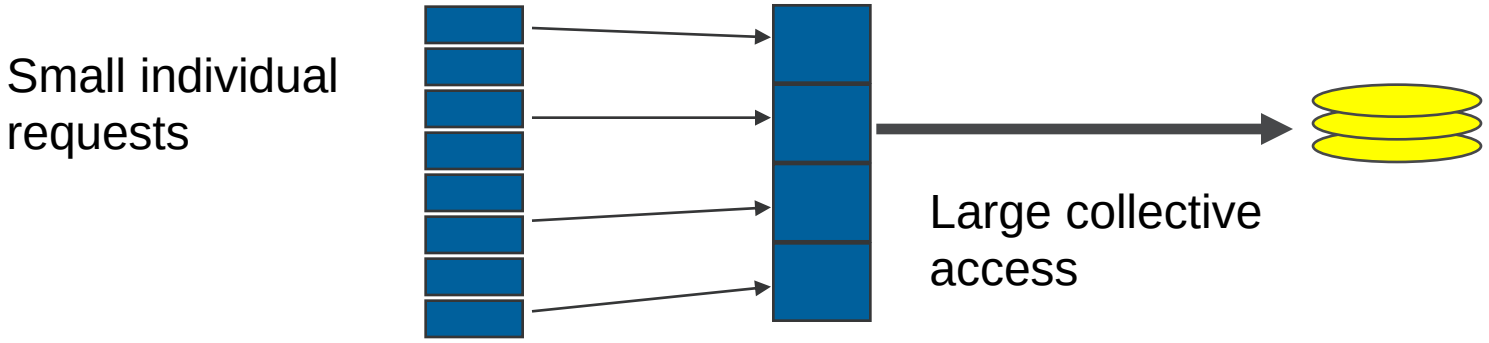
PARALLELIZING OUR I/O API

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 optimizations at the MPI-IO layer
  - e.g., two-phase I/O



COLLECTIVE MPI I/O FUNCTIONS

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

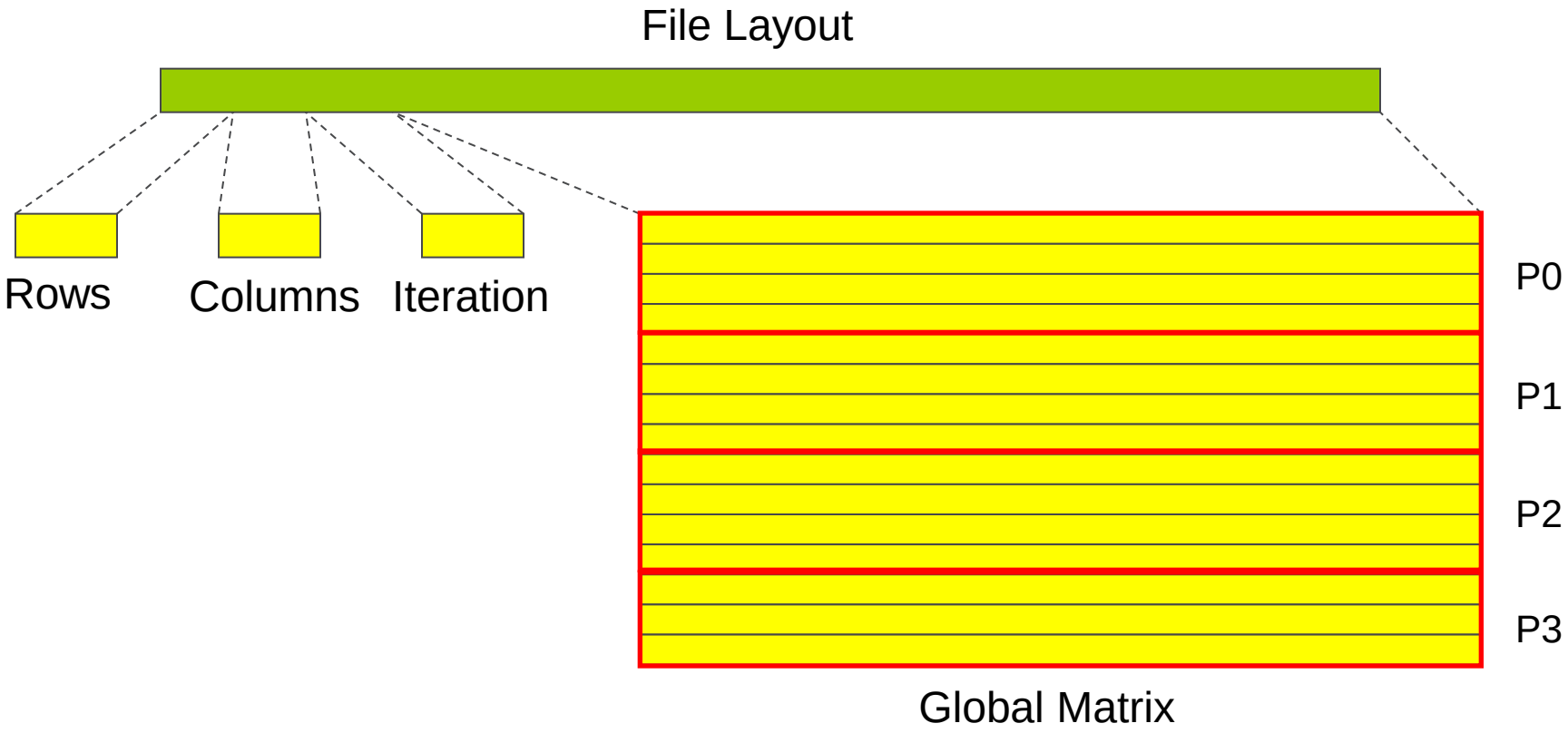
- **MPI_File_write_at_all**, etc.
  - _**all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
  - _**at** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than 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.

```
56: /* SLIDE: Life MPI-IO Checkpoint/Restart */
57: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
58:                         int cols, int iter, MPI_Info info)
59: {
60:     int err;
61:     int amode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
62:                 MPI_MODE_UNIQUE_OPEN;
63:     int rank, nprocs;
64:     int myrows, myoffset;
65:
66:     MPI_File fh;
67:     MPI_Datatype type;
68:     MPI_Offset myfileoffset;
69:     char filename[64];
70:
71:     MPI_Comm_size(mlifeio_comm, &nprocs);
72:     MPI_Comm_rank(mlifeio_comm, &rank);
73:
74:     myrows   = MLIFE_myrows(rows, rank, nprocs);
75:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
76:
77:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
78:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
79:     if (err != MPI_SUCCESS) {
80:         fprintf(stderr, "Error opening %s.\n", filename);
81:         return err;
82:     }
83:
84:
```

```
 85: /* SLIDE: Life MPI-IO Checkpoint/Restart */
 86:     if (rank == 0) {
 87:         MLIFEIO_Type_create_hdr_rowblk(matrix, myrows, &rows,
 88:                                        &cols, &iter, &type);
 89:         myfileoffset = 0;
 90:     }
 91:     else {
 92:         MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
 93:         myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
 94:     }
 95:
 96:     MPI_Type_commit(&type);
 97:     err = MPI_File_write_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
 98:                                 type, MPI_STATUS_IGNORE);
 99:     MPI_Type_free(&type);
100:
101:     err = MPI_File_close(&fh);
102:     return err;
103: }
104:
```

```
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:
115:     MPI_File fh;
116:     MPI_Datatype type;
117:     MPI_Offset myfileoffset;
118:     char filename[64];
119:
120:     MPI_Comm_size(mlifeio_comm, &nprocs);
121:     MPI_Comm_rank(mlifeio_comm, &rank);
122:
123:     myrows   = MLIFE_myrows(rows, rank, nprocs);
124:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
125:
126:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
127:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
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);
133:
```

```
134: /* SLIDE: Life MPI-IO Checkpoint/Restart */
135:     /* Have all process check that nothing went wrong */
136:     MPI_Allreduce(&err, &gErr, 1, MPI_INT, MPI_MAX, mlifeio_comm);
137:     if (gErr || buf[0] != rows || buf[1] != cols) {
138:         if (rank == 0) fprintf(stderr, "restart failed.\n");
139:         return MPI_ERR_OTHER;
140:     }
141:
142:     MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
143:     myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
144:
145:     MPI_Type_commit(&type);
146:     err = MPI_File_read_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
147:                                type, MPI_STATUS_IGNORE);
148:     MPI_Type_free(&type);
149:
150:     err = MPI_File_close(&fh);
151:     return err;
152: }
153:
```

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.

```
154: /* SLIDE: Describing Header and Data */
155: /* MLIFEIO_Type_create_hdr_rowblk
156:  *
157:  * Used by process zero to create a type that describes both
158:  * the header data for a checkpoint and its contribution to
159:  * the stored matrix.
160:  *
161:  * Parameters:
162:  * matrix  - pointer to the matrix, including boundaries
163:  * myrows  - number of rows held locally
164:  * rows_p  - pointer to # of rows in matrix (so we can get its
165:  *           address for use in the type description)
166:  * cols_p  - pointer to # of cols in matrix
167:  * iter_p  - pointer to iteration #
168:  * newtype - pointer to location to store new type ref.
169:  */
170: static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
171:                                           int myrows,
172:                                           int *rows_p,
173:                                           int *cols_p,
174:                                           int *iter_p,
175:                                          MPI_Datatype *newtype)
176: {
177:     int err;
178:     int lens[4] = { 1, 1, 1, 1 };
179:     MPI_Aint disps[4];
180:     MPI_Datatype types[4];
181:     MPI_Datatype rowblk;
182:
```

```
183: /* SLIDE: Describing Header and Data */
184:     MLIFEIO_Type_create_rowblk(matrix, myrows, *cols_p, &rowblk);
185:
186:     MPI_Address(rows_p, &disps[0]);
187:     MPI_Address(cols_p, &disps[1]);
188:     MPI_Address(iter_p, &disps[2]);
189:     disps[3] = (MPI_Aint) MPI_BOTTOM;
190:     types[0] = MPI_INT;
191:     types[1] = MPI_INT;
192:     types[2] = MPI_INT;
193:     types[3] = rowblk;
194:
195: #if defined(MPI_VERSION) && MPI_VERSION >= 2
196:     err = MPI_Type_create_struct(3, lens, disps, types, newtype);
197: #else
198:     err = MPI_Type_struct(3, lens, disps, types, newtype);
199: #endif
200:
201:     MPI_Type_free(&rowblk);
202:
203:     return err;
204: }
205:
```

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