```
 1: /* SLIDE: Life Point-to-Point 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 <string.h>
11:
12: #include <mpi.h>
13:
14: #include "mlife.h"
15: #include "mlife-io.h"
16:
17: extern double drand48(void);
18: static int MLIFE_nextstate(int **matrix, int y, int x);
19: static int MLIFE_parse_args(int argc, char **argv);
20:
21: /* options */
22: static int opt_rows = 25, opt_cols = 70, opt_iter = 10;
23: static int opt_restart_iter = -1;
24: static char opt_prefix[64] = "mlife";
25:
```

```
26: /* SLIDE: Life Point-to-Point Code Walkthrough */
27: int main(int argc, char *argv[])
28: {
29:     int rank;
30:     double time;
31:
32:     MPI_Init(&argc, &argv);
33:     MLIFE_parse_args(argc, argv);
34:     MLIFEIO_Init(MPI_COMM_WORLD);
35:
36:     time = life(opt_rows, opt_cols, opt_iter, MPI_COMM_WORLD);
37:
38:     /* print the total time taken */
39:     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
40:     if (rank == 0)
41:         printf("[%d] Life finished in %lf secs of calculation\n",
42:                rank, time);
43:
44:     MLIFEIO_Finalize();
45:     MPI_Finalize();
46:
47:     return 0;
48: }
49:
```

```
50: /* SLIDE: Life Point-to-Point Code Walkthrough */
51: double life(int rows, int cols, int ntimes, MPI_Comm comm)
52: {
53:     int      err, i, j, k, rank, nprocs, next, prev;
54:     int      myrows, myoffset;
55:     int    **matrix, **temp, **addr;
56:     int     *mdata, *tdata;
57:     double   mytime, totaltime=0.0, starttime;
58:
59:     MPI_Comm_size(comm, &nprocs);
60:     MPI_Comm_rank(comm, &rank);
61:
62:     /* set neighbors */
63:     if (rank == 0)
64:         prev = MPI_PROC_NULL;
65:     else
66:         prev = rank-1;
67:     if (rank == nprocs - 1)
68:         next = MPI_PROC_NULL;
69:     else
70:         next = rank+1;
71:
72:     /* determine my part of the matrix, row-block distribution */
73:     myrows   = MLIFE_myrows(rows, rank, nprocs);
74:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
75:
76:     /* allocate memory for the matrix using single blocks */
77:     matrix = (int **) malloc((myrows+2) * sizeof(int *));
78:     temp   = (int **) malloc((myrows+2) * sizeof(int *));
79:     mdata  = (int *) malloc((myrows+2) * (cols+2) * sizeof(int));
80:     tdata  = (int *) malloc((myrows+2) * (cols+2) * sizeof(int));
```

```
 81: /* SLIDE: Life Point-to-Point Code Walkthrough */
 82:     /* set up pointers for convenience */
 83:     matrix[0] = mdata;
 84:     temp[0]   = tdata;
 85:     for (i = 1; i < myrows+2; i++) {
 86:         matrix[i] = matrix[i-1] + cols + 2;
 87:         temp[i]   = temp[i-1] + cols + 2;
 88:     }
 89:
 90:     /* initialize the boundaries of the life matrix */
 91:     for (j = 0; j < cols+2; j++) {
 92:         matrix[0][j] = matrix[myrows+1][j] = temp[0][j]
 93:                      = temp[myrows+1][j] = DIES;
 94:     }
 95:     for (i = 0; i < myrows+2; i++) {
 96:         matrix[i][0] = matrix[i][cols+1] = temp[i][0]
 97:                      = temp[i][cols+1] = DIES;
 98:     }
 99:
100:     if (opt_restart_iter == -1) {
101:         /* initialize the life matrix */
102:         for (i = 1; i <= myrows; i++)  {
103:             srand48((long)(1000^(i + myoffset)));
104:
105:             for (j = 1; j<= cols; j++) {
106:                 if (drand48() > 0.5) matrix[i][j] = BORN;
107:                 else                 matrix[i][j] = DIES;
108:             }
109:         }
110:     }
```

```
111: /* SLIDE: Life Point-to-Point Code Walkthrough */
112:     else if (MLIFEIO_Can_restart()) {
113:         /* read state from checkpoint file */
114:         err = MLIFEIO_Restart(opt_prefix, matrix, rows, cols,
115:                               opt_restart_iter, MPI_INFO_NULL);
116:         if (err != MPI_SUCCESS) {
117:             MPI_Abort(MPI_COMM_WORLD, MPI_ERR_IO);
118:         }
119:     }
120:     else {
121:         fprintf(stderr, "Restart unsupported by I/O backend.\n");
122:         MPI_Abort(MPI_COMM_WORLD, MPI_ERR_IO);
123:     }
124:
125:     MLIFE_exchange_init(comm, &matrix[0][0], &temp[0][0],
126:                         myrows, rows, cols, prev, next);
127:
128:     /* use portable MPI function for timing */
129:     starttime = MPI_Wtime();
130:
131:     for (k = 0; k < ntimes; k++)
132:     {
133:         MLIFE_exchange(matrix, myrows, cols);
134:
135:         /* calculate new state for all non-boundary elements */
136:         for (i = 1; i <= myrows; i++) {
137:             for (j = 1; j < cols+1; j++) {
138:                 temp[i][j] = MLIFE_nextstate(matrix, i, j);
139:             }
140:         }
141:
```

```
142: /* SLIDE: Life Point-to-Point Code Walkthrough */
143:         /* swap the matrices */
144:         addr   = matrix;
145:         matrix = temp;
146:         temp   = addr;
147:
148:         err = MLIFEIO_Checkpoint(opt_prefix, matrix, rows, cols,
149:                                  k, MPI_INFO_NULL);
150:     }
151:
152:     /* return the average time taken/processor */
153:     mytime = MPI_Wtime() - starttime;
154:     MPI_Reduce(&mytime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0,
155:                comm);
156:
157:     MLIFE_exchange_finalize();
158:     free(matrix);
159:     free(temp);
160:     free(mdata);
161:     free(tdata);
162:
163:     return(totaltime/(double) nprocs);
164: }
165:
```

```
/* SLIDE: Life Point-to-Point Code Walkthrough */
static int MLIFE_nextstate(int **matrix, int row, int col)
{
    int sum;

    /* add values of all eight neighbors */
    sum = matrix[row-1][col-1] + matrix[row-1][col] +
          matrix[row-1][col+1] + matrix[row][col-1] +
          matrix[row][col+1] + matrix[row+1][col-1] +
          matrix[row+1][col] + matrix[row+1][col+1];

    if (sum < 2 || sum > 3) return DIES;
    else if (sum == 3)      return BORN;
    else                    return matrix[row][col];
}

int MLIFE_myrows(int rows, int rank, int nprocs)
{
    int myrows;

    myrows = rows / nprocs;

    if (rank < (rows % nprocs)) myrows += 1;

    return myrows;
}
```

```
/* SLIDE: Life Point-to-Point Code Walkthrough */
int MLIFE_myrowoffset(int rows, int rank, int nprocs)
{
    int myoffset;

    myoffset = rank * (rows / nprocs);

    if (rank > (rows % nprocs)) myoffset += (rows % nprocs);
    else                        myoffset += rank;

    return myoffset;
}

```

```
/* SLIDE: Note: Passing Arguments */
/* MLIFE_parse_args
 *
 * Note: Command line arguments are not guaranteed in the MPI
 *       environment to be passed to all processes.  To be
 *       portable, we must process on rank 0 and distribute
 *       results.
 */
static int MLIFE_parse_args(int argc, char **argv)
{
    int ret;
    int rank;
    int myargs[5]; /* array for simple sending of arguments */

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        while ((ret = getopt(argc, argv, "x:y:i:p:r:")) >= 0)
        {
            switch(ret) {
                case 'x':
                    opt_cols = atoi(optarg);
                    break;
                case 'y':
                    opt_rows = atoi(optarg);
                    break;
                case 'i':
                    opt_iter = atoi(optarg);
                    break;
                case 'r':
                    opt_restart_iter = atoi(optarg);
```

```
/* SLIDE: Note: Passing Arguments */
                case 'p':
                    strncpy(opt_prefix, optarg, 63);
                    break;
                default:
                    break;
            }
        }

        myargs[0] = opt_rows;
        myargs[1] = opt_cols;
        myargs[2] = opt_iter;
        myargs[3] = opt_restart_iter;
        myargs[4] = strlen(opt_prefix) + 1;
    }

    MPI_Bcast(myargs, 5, MPI_INT, 0, MPI_COMM_WORLD);
    opt_rows = myargs[0];
    opt_cols = myargs[1];
    opt_iter = myargs[2];

    MPI_Bcast(opt_prefix, myargs[4], MPI_CHAR, 0, MPI_COMM_WORLD);

    return 0;
}
```

```
 1: /* SLIDE: Point-to-Point Exchange */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *  (C) 2004 by University of Chicago.
 5:  *      See COPYRIGHT in top-level directory.
 6:  */
 7: #include <mpi.h>
 8:
 9: #include "mlife.h"
10:
11: static MPI_Comm exch_comm = MPI_COMM_NULL;
12: static int exch_prev, exch_next;
13:
14: /* MLIFE_exchange_init
15:  *
16:  * Parameters:
17:  * comm   - communicator describing group of processes
18:  * matrix - pointer to original matrix data
19:  * temp   - pointer to second region used during calculations
20:  * myrows - number of rows held locally
21:  * rows   - number of rows in global matrix
22:  * cols   - number of columns in global matrix
23:  * prev   - rank of processor "above" this one
24:  * next   - rank of processor "below" this one
25:  *
26:  * Note: Some of these parameters are not used by this exchange
27:  *       implementation.
28:  */
29: int MLIFE_exchange_init(MPI_Comm comm, void *matrix, void *temp,
30:                        int myrows, int rows, int cols,
31:                        int prev, int next)
32: {
```

```
/* SLIDE: Point-to-Point Exchange */
    MPI_Comm_dup(comm, &exch_comm);
    exch_prev = prev;
    exch_next = next;

    return MPI_SUCCESS;
}

void MLIFE_exchange_finalize(void)
{
    MPI_Comm_free(&exch_comm);
}

```

```
/* SLIDE: Point-to-Point Exchange */
int MLIFE_exchange(int **matrix, int myrows, int cols)
{
    MPI_Request reqs[4];

    /* exchange edges */
    MPI_Isend(&matrix[1][0], cols+2, MPI_INT,
              exch_prev, 0, exch_comm, reqs);
    MPI_Irecv(&matrix[0][0], cols+2, MPI_INT,
              exch_prev, 0, exch_comm, reqs+1);
    MPI_Isend(&matrix[myrows][0], cols+2, MPI_INT,
              exch_next, 0, exch_comm, reqs+2);
    MPI_Irecv(&matrix[myrows+1][0], cols+2, MPI_INT,
              exch_next, 0, exch_comm, reqs+3);

    MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);

    return MPI_SUCCESS;
}

/* END OF EXAMPLE */
```

```
 1: /* SLIDE: MPI-IO Life Checkpoint Code Walkthrough */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *
 5:  *  (C) 2004 by University of Chicago.
 6:  *      See COPYRIGHT in top-level directory.
 7:  */
 8:
 9: #include <stdio.h>
10: #include <stdlib.h>
11:
12: #include <mpi.h>
13:
14: #include "mlife-io.h"
15:
16: /* MPI-IO implementation of checkpoint and restart for MPI Life
17:  *
18:  * Data stored in matrix order, with a header consisting of three
19:  * integers: matrix size in rows and columns, and iteration no.
20:  *
21:  * Each checkpoint is stored in its own file.
22:  */
23:
24: static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
25:                                        int cols,
26:                                        MPI_Datatype *newtype);
27: static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
28:                                          int myrows,
29:                                          int *rows_p,
30:                                          int *cols_p,
31:                                          int *iter_p,
32:                                          MPI_Datatype *newtype);
```

```
33: /* SLIDE: MPI-IO Life Checkpoint Code Walkthrough */
34: static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
35:
36: int MLIFEIO_Init(MPI_Comm comm)
37: {
38:     int err;
39:
40:     err = MPI_Comm_dup(comm, &mlifeio_comm);
41:
42:     return err;
43: }
44:
45: int MLIFEIO_Finalize(void)
46: {
47:     int err;
48:
49:     err = MPI_Comm_free(&mlifeio_comm);
50:
51:     return err;
52: }
53:
54: int MLIFEIO_Can_restart(void)
55: {
56:     return 1;
57: }
58:
```

```
59: /* SLIDE: Life MPI-IO Checkpoint/Restart */
60: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
61:                        int cols, int iter, MPI_Info info)
62: {
63:     int err;
64:     int amode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
65:                 MPI_MODE_UNIQUE_OPEN;
66:     int rank, nprocs;
67:     int myrows, myoffset;
68:
69:     MPI_File fh;
70:
71:     MPI_Datatype type;
72:     MPI_Offset myfileoffset;
73:
74:     char filename[64];
75:
76:     MPI_Comm_size(mlifeio_comm, &nprocs);
77:     MPI_Comm_rank(mlifeio_comm, &rank);
78:
79:     myrows   = MLIFE_myrows(rows, rank, nprocs);
80:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
81:
82:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
83:
84:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
85:     if (err != MPI_SUCCESS) {
86:         fprintf(stderr, "Error opening %s.\n", filename);
87:         return err;
88:     }
89:
90:     if (rank == 0) {
```

```
 91: /* SLIDE: Life MPI-IO Checkpoint/Restart */
 92:         MLIFEIO_Type_create_hdr_rowblk(matrix, myrows, &rows,
 93:                                        &cols, &iter, &type);
 94:         myfileoffset = 0;
 95:     }
 96:     else {
 97:         MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
 98:         myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
 99:     }
100:
101:     MPI_Type_commit(&type);
102:     err = MPI_File_write_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
103:                                 type, MPI_STATUS_IGNORE);
104:     MPI_Type_free(&type);
105:
106:     err = MPI_File_close(&fh);
107:     return err;
108: }
109:
```

```
110: /* SLIDE: Life MPI-IO Checkpoint/Restart */
111: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
112:                     int cols, int iter, MPI_Info info)
113: {
114:     int err, gErr;
115:     int amode = MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN;
116:     int rank, nprocs;
117:     int myrows, myoffset;
118:     int buf[3]; /* rows, cols, iteration */
119:
120:     MPI_File fh;
121:
122:     MPI_Datatype type;
123:     MPI_Offset myfileoffset;
124:
125:     char filename[64];
126:
127:     MPI_Comm_size(mlifeio_comm, &nprocs);
128:     MPI_Comm_rank(mlifeio_comm, &rank);
129:
130:     myrows   = MLIFE_myrows(rows, rank, nprocs);
131:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
132:
133:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
134:
135:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
136:     if (err != MPI_SUCCESS) return err;
137:
138:     /* check that rows and cols match */
139:     err = MPI_File_read_at_all(fh, 0, buf, 3, MPI_INT,
140:                                MPI_STATUS_IGNORE);
141:
```

```
/* 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);

    err = MPI_File_close(&fh);
    return err;
}
```

```
162: /* SLIDE: Describing Header and Data */
163: /* MLIFEIO_Type_create_hdr_rowblk
164:  *
165:  * Used by process zero to create a type that describes both
166:  * the header data for a checkpoint and its contribution to
167:  * the stored matrix.
168:  *
169:  * Parameters:
170:  * matrix  - pointer to the matrix, including boundaries
171:  * myrows  - number of rows held locally
172:  * rows_p  - pointer to # of rows in matrix (so we can get its
173:  *           address for use in the type description)
174:  * cols_p  - pointer to # of cols in matrix
175:  * iter_p  - pointer to iteration #
176:  * newtype - pointer to location to store new type ref.
177:  */
178: static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
179:                                           int myrows,
180:                                           int *rows_p,
181:                                           int *cols_p,
182:                                           int *iter_p,
183:                                           MPI_Datatype *newtype)
184: {
185:     int err;
186:     int lens[4] = { 1, 1, 1, 1 };
187:     MPI_Aint disps[4];
188:     MPI_Datatype types[4];
189:     MPI_Datatype rowblk;
190:
191:     MLIFEIO_Type_create_rowblk(matrix, myrows, *cols_p, &rowblk);
192:
193:     MPI_Address(rows_p, &disps[0]);
```

```
/* SLIDE: Describing Header and Data */
    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;
}
```

```
/* SLIDE: Placing Data in Checkpoint */
/* MLIFEIO_Type_create_rowblk
 *
 * See stdio version for details (this is a copy).
 */
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_INTEGER,
                          &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;
}
```

```
 1: /* SLIDE: Fence Life Exchange Code Walkthrough */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *
 5:  *  (C) 2004 by University of Chicago.
 6:  *      See COPYRIGHT in top-level directory.
 7:  */
 8:
 9: #include <mpi.h>
10:
11: #include "mlife.h"
12:
13: static int exch_prev, exch_next, nrows_prev;
14: static MPI_Win matrix_win, temp_win;
15:
16: typedef struct mem_win{
17:     void *mem;
18:     MPI_Win win;
19: } mem_win;
20:
21: static mem_win mem_win_map[2];
22:
23: int MLIFE_exchange_init(MPI_Comm comm, void *matrix, void *temp,
24:                        int myrows, int rows, int cols, int prev,
25:                        int next)
26: {
27:     int err=MPI_SUCCESS, nprocs;
28:
29:     exch_prev = prev;
30:     exch_next = next;
31:
32:     /* create windows */
```

```
33: /* SLIDE: Fence Life Exchange Code Walkthrough */
34:     MPI_Win_create(matrix, (myrows+2)*(cols+2)*sizeof(int),
35:                    sizeof(int), MPI_INFO_NULL, comm, &matrix_win);
36:
37:     MPI_Win_create(temp, (myrows+2)*(cols+2)*sizeof(int),
38:                    sizeof(int), MPI_INFO_NULL, comm, &temp_win);
39:
40:     /* store mapping from memory address to associated window */
41:
42:     mem_win_map[0].mem = matrix;
43:     mem_win_map[0].win = matrix_win;
44:     mem_win_map[1].mem = temp;
45:     mem_win_map[1].win = temp_win;
46:
47:     /* calculate no. of local rows in prev */
48:
49:     if (exch_prev == MPI_PROC_NULL)
50:         nrows_prev = 0;
51:     else {
52:         MPI_Comm_size(comm, &nprocs);
53:         nrows_prev = MLIFE_myrows(rows, exch_prev, nprocs);
54:     }
55:
56:     return err;
57: }
58:
59: void MLIFE_exchange_finalize(void)
60: {
61:     MPI_Win_free(&matrix_win);
62:     MPI_Win_free(&temp_win);
63: }
```

```
/* SLIDE: Fence Life Exchange Code Walkthrough */
int MLIFE_exchange(int **matrix, int myrows, int cols)
{
    int err=MPI_SUCCESS;
    MPI_Win win;

    /* Find the right window object */
    if (mem_win_map[0].mem == &matrix[0][0])
        win = mem_win_map[0].win;
    else
        win = mem_win_map[1].win;

    /* Send and receive boundary information */
    MPI_Win_fence(MPI_MODE_NOPRECEDE, win);

    MPI_Put(&matrix[myrows][0], cols+2, MPI_INT, exch_next, 0,
            cols+2, MPI_INT, win);

    MPI_Put(&matrix[1][0], cols+2, MPI_INT, exch_prev,
      (nrows_prev+1)*(cols+2), cols+2, MPI_INT, win);

    MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
                  MPI_MODE_NOSUCCEED,
                  win);

    return err;
}
```

```
 1: /* SLIDE: P/S/C/W Life Exchange Code Walkthrough */
 2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
 3: /*
 4:  *
 5:  *  (C) 2004 by University of Chicago.
 6:  *      See COPYRIGHT in top-level directory.
 7:  */
 8:
 9: #include <mpi.h>
10:
11: #include "mlife.h"
12:
13: static int exch_prev, exch_next, nrows_prev;
14: static MPI_Win matrix_win, temp_win;
15: static MPI_Group rma_grp;
16:
17: typedef struct mem_win{
18:     void *mem;
19:     MPI_Win win;
20: } mem_win;
21:
22: static mem_win mem_win_map[2];
23:
24: int MLIFE_exchange_init(MPI_Comm comm, void *matrix, void *temp,
25:                        int myrows, int rows, int cols, int prev,
26:                        int next)
27: {
28:     int err=MPI_SUCCESS, nprocs, ranks[2], i;
29:     MPI_Group comm_grp;
30:
31:     exch_prev = prev;
32:     exch_next = next;
```

```
33: /* SLIDE: P/S/C/W Life Exchange Code Walkthrough */
34:     /* create windows */
35:     MPI_Win_create(matrix, (myrows+2)*(cols+2)*sizeof(int),
36:                    sizeof(int), MPI_INFO_NULL, comm, &matrix_win);
37:
38:     MPI_Win_create(temp, (myrows+2)*(cols+2)*sizeof(int),
39:                    sizeof(int), MPI_INFO_NULL, comm, &temp_win);
40:
41:     MPI_Comm_group(comm, &comm_grp);
42:
43:     i = 0;
44:     if (prev != MPI_PROC_NULL) {
45:         ranks[i] = prev;
46:         i++;
47:     }
48:     if (next != MPI_PROC_NULL) {
49:         ranks[i] = next;
50:         i++;
51:     }
52:
53:     MPI_Group_incl(comm_grp, i, ranks, &rma_grp);
54:
55:     MPI_Group_free(&comm_grp);
56:
57:     /* store mapping from memory address to associated window */
58:
59:     mem_win_map[0].mem = matrix;
60:     mem_win_map[0].win = matrix_win;
61:     mem_win_map[1].mem = temp;
62:     mem_win_map[1].win = temp_win;
63:
64:     /* calculate no. of local rows in prev */
```

```
/* SLIDE: P/S/C/W Life Exchange Code Walkthrough */
    if (exch_prev == MPI_PROC_NULL)
        nrows_prev = 0;
    else {
        MPI_Comm_size(comm, &nprocs);
        nrows_prev = MLIFE_myrows(rows, exch_prev, nprocs);
    }

    return err;
}

void MLIFE_exchange_finalize(void)
{
    MPI_Win_free(&matrix_win);
    MPI_Win_free(&temp_win);

    MPI_Group_free(&rma_grp);
}
```

```
 83: /* SLIDE: P/S/C/W Life Exchange Code Walkthrough */
 84: int MLIFE_exchange(int **matrix,
 85:                    int   myrows,
 86:                    int   cols)
 87: {
 88:     int err=MPI_SUCCESS;
 89:     MPI_Win win;
 90:
 91:     /* Find the right window object */
 92:     if (mem_win_map[0].mem == &matrix[0][0])
 93:         win = mem_win_map[0].win;
 94:     else
 95:         win = mem_win_map[1].win;
 96:
 97:     MPI_Win_post(rma_grp, 0, win);
 98:     MPI_Win_start(rma_grp, 0, win);
 99:
100:     /* Send and receive boundary information */
101:     MPI_Put(&matrix[myrows][0], cols+2, MPI_INT, exch_next, 0,
102:             cols+2, MPI_INT, win);
103:
104:     MPI_Put(&matrix[1][0], cols+2, MPI_INT, exch_prev,
105:       (nrows_prev+1)*(cols+2), cols+2, MPI_INT, win);
106:
107:     MPI_Win_complete(win);
108:     MPI_Win_wait(win);
109:
110:     return err;
111: }
```

```
 1: /* SLIDE: pNeo Code Walkthrough */
 2: /*  This is an abstraction of the pNeo brain simulation program,
 3:     written to illustrate the MPI one-sided operations.
 4:
 5:     Run with
 6:
 7:       pneo <filename>
 8:
 9:     where <filename> is a file of connections, one on each line
10:     consisting of a blank-separated pair of integers in character
11:     format.  Only one cell per proc is modelled in this version.
12:
13:     This is the fence version.
14:
15: */
16:
17: #include <stdio.h>
18: #include <stdlib.h>
19: #include "mpi.h"
20:
21: void init_state( char * );
22: void reset_state( void );
23: int  ready_to_fire( void );
24: void setup_groups( void );
25: void compute_state( void );
26: void output_spikes( void );
27: void dump_local_arrays( void );
28:
29:
30:
31:
32:
```

```
33: /* SLIDE: pNeo Code Walkthrough */
34: /* Connections to other cells are prepresented by an array of
35:  * inputs (inconnections) and an array of outputs (outconnections)
36:  * A connection array entry contains the rank of the other process
37:  * and the values of the incoming or outgoing spikes.  The input
38:  * connection arrays are the windows. The outconnections also
39:  * contain the displacement into the destination window of the
40:  * connection.
41:  */
42:
43: typedef struct {
44:     int source;
45:     int inspike;
46: } inconnection;
47:
48: inconnection *inarray;          /* array of inputs */
49: int inarray_count;
50: MPI_Win win;                    /* the window for this process,
51:                                  * which will be identified with
52:                                  * the inarray */
53: typedef struct {
54:     int dest;
55:     int disp;
56: } outconnection;
57:
58: outconnection *outarray;        /* array of outputs */
59: int outarray_count;
60:
61: int state;                      /* state of the cell */
62:
63: int numprocs, myrank;
64: int itercount;
```

```
65: /* SLIDE: pNeo Code Walkthrough */
66: int max_steps = 100;            /* number of steps to run */
67:
68: int main(int argc, char *argv[])
69: {
70:     MPI_Init(NULL, NULL);
71:     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
72:     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
73:
74:     if (argc < 2) {
75:         printf("usage: %s <connection file>\n", argv[0]);
76:         MPI_Abort(MPI_COMM_WORLD, -1);
77:     }
78:
79:     /*initialize connection arrays and cell state */
80:     init_state(argv[1]);
81:
82:     /* make input arrays the windows */
83:     MPI_Win_create(inarray, inarray_count * sizeof(int),
84:                    sizeof(int), MPI_INFO_NULL,
85:                    MPI_COMM_WORLD, &win);
86:
87:     for (itercount = 0; itercount < max_steps; itercount++) {
88:         compute_state();
89:         MPI_Win_fence(0, win);
90:         if (ready_to_fire()) {
91:             output_spikes();
92:             reset_state();
93:         }
94:         MPI_Win_fence(0, win);
95:     }
96:
```

```
 97: /* SLIDE: pNeo Code Walkthrough */
 98:     MPI_Win_free(&win);
 99:     MPI_Finalize();
100:     return 0;
101: }
102:
103: void init_state(char *filename)
104: {
105: #   define MAX_CONNECTIONS 10000
106: #   define MAX(x,y) ((x) > (y) ? x : y)
107:
108:     /* array of connections described in connection file */
109:     typedef struct {
110:         int source;
111:         int dest;
112:     } connection;
113:
114:     connection connarray[MAX_CONNECTIONS];
115:
116:     FILE *confile;
117:     int i, j, k, n, connarray_count, dispcnt;
118:     int maxpair, maxcell;
119:
120:     if ((confile = fopen(filename, "r")) == NULL) {
121:         printf("could not open connection file %s\n", filename);
122:         MPI_Abort(MPI_COMM_WORLD, -1);
123:     }
124:
125:     /* The following code should be executed only by process 0,
126:      * which then should broadcast the connarray */
127:     maxcell = connarray_count = i = 0;
128:     inarray_count = outarray_count = 0;
```

```
129: /* SLIDE: pNeo Code Walkthrough */
130:     while ((fscanf(confile, "%d %d", &connarray[i].source,
131:                    &connarray[i].dest)) == 2) {
132:         connarray_count++;
133:         maxpair = MAX(connarray[i].source, connarray[i].dest);
134:         if (maxpair > maxcell)
135:             maxcell = maxpair;
136:         if (connarray[i].dest == myrank)
137:             inarray_count++;
138:         if (connarray[i].source == myrank)
139:             outarray_count++;
140:         i++;
141:     }
142:     if (maxcell > numprocs-1) {
143:         printf("%d processes needed for file %s\n",
144:                maxcell+1, filename);
145:         MPI_Abort(MPI_COMM_WORLD, -1);
146:     }
147:
148:     /* dump connarray, for debugging */
149:     for (i = 0; i < connarray_count; i++) {
150:         printf("%d %d\n", connarray[i].source, connarray[i].dest);
151:     }
152:
153:     inarray  = (inconnection *)
154:                    malloc(inarray_count * sizeof(inconnection));
155:     outarray = (outconnection *)
156:                    malloc(outarray_count * sizeof(outconnection));
157:
158:     for (i = j = k = 0 ; i < connarray_count; i++) {
159:         if (connarray[i].dest == myrank) {
160:             inarray[j].source = connarray[i].source;
```

```
161: /* SLIDE: pNeo Code Walkthrough */
162:                 inarray[j].inspike = 0;
163:                 j++;
164:             }
165:             inarray_count = j;
166:
167:             if (connarray[i].source == myrank) {
168:                 dispcnt = 0;
169:                 for (n = 0; n < connarray_count; n++) {
170:                     if (connarray[n].dest == connarray[i].dest) {
171:                         if (connarray[n].source == myrank)
172:                             break;
173:                         else
174:                             dispcnt++; /* increment counter on
175:                                           destination process i */
176:                     }
177:                 }
178:                 outarray[k].dest = connarray[i].dest;
179:                 outarray[k].disp = dispcnt;
180:                 k++;
181:             }
182:             outarray_count = k;
183:         }
184:
185:         dump_local_arrays();
186:
187:         state = (myrank + 5) % numprocs; /* essentially random
188:                                             for this example */
189: }
190:
191: void compute_state()
192: {
```

```
193: /* SLIDE: pNeo Code Walkthrough */
194:     int i;
195:     int num_incoming = 0;
196:
197:     for (i = 0; i < inarray_count; i++) {
198:         num_incoming += inarray[i].inspike;
199:     }
200:
201:     state = state + num_incoming + 1;
202: }
203:
204: void reset_state()
205: {
206:     state = 0;
207: }
208:
209: int ready_to_fire()
210: {
211:     if (state > 4)
212:         return 1;
213:     else
214:         return 0;
215: }
216:
217: void output_spikes()
218: {
219:     int i;
220:     static int spike = 1;
221:
222:     for (i = 0; i < outarray_count; i++) {
223:         printf("putting spike from %d to %d in interation %d\n",
224:                myrank, outarray[i].dest, itercount);
```

```
225: /* SLIDE: pNeo Code Walkthrough */
226:         MPI_Put(&spike, 1, MPI_INT, outarray[i].dest,
227:                 outarray[i].disp, 1, MPI_INT, win);
228:     }
229: }
230:
231: void dump_local_arrays()
232: {
233:     int i;
234:
235:     printf("inarray for process %d:\n", myrank);
236:     for (i = 0; i < inarray_count; i++)
237:         printf("%d %d\n", inarray[i].source, inarray[i].inspike);
238:     printf("outarray for process %d:\n", myrank);
239:     for (i = 0; i < outarray_count; i++)
240:         printf("%d %d\n", outarray[i].dest, outarray[i].disp);
241: }
242:
```

```
 1: /* SLIDE: pNeo Code Walkthrough */
 2: /*  This is an abstraction of the pNeo brain simulation program,
 3:     written to illustrate the MPI one-sided operations.
 4:
 5:     Run with
 6:
 7:       pneo <filename>
 8:
 9:     where <filename> is a file of connections, one on each line
10:     consisting of a blank-separated pair of integers in character
11:     format. Only one cell per process is modelled in this version.
12:
13:     This is the post-start-complete-wait version.  (pscw)
14:
15: */
16:
17: #include <stdio.h>
18: #include <stdlib.h>
19: #include "mpi.h"
20:
21: void init_state( char * );
22: void reset_state( void );
23: int  ready_to_fire( void );
24: void setup_groups( void );
25: void compute_state( void );
26: void output_spikes( void );
27: void dump_local_arrays( void );
28: void dump_window( void );
29:
```

```
30: /* SLIDE: pNeo Code Walkthrough */
31: /* Connections to other cells are prepresented by an array of
32:  * inputs (inconnections) and an array of outputs (outconnections)
33:  * A connection array entry contains the rank of the other process
34:  * and the values of the incoming or outgoing spikes.  The input
35:  * connection arrays are the windows. The outconnections also
36:  * contain the displacement into the destination window of the
37:  * connection.
38:  */
39:
40: int *inarray;                /* array of possible input spikes */
41: int *inranks;                /* array of possible sources */
42: int inarray_count;
43:
44: MPI_Win win;                     /* the window for this process,
45:                                  * which will be identified with
46:                                  * the inarray */
47: typedef struct {
48:     int dest;
49:     int disp;
50: } outconnection;
51:
52: outconnection *outarray;         /* array of outputs */
53: int outarray_count;
54:
55: int state;                       /* state of the cell */
56:
57: /* For post-start-complete-wait version only */
58: int *outranks;
59: MPI_Group ingroup, outgroup;
60:
61: int numprocs, myrank;
```

```
62: /* SLIDE: pNeo Code Walkthrough */
63: int itercount;
64: int max_steps = 100;            /* number of steps to run */
65: int main(int argc, char *argv[])
66: {
67:     MPI_Init(NULL, NULL);
68:     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
69:     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
70:
71:     if (argc < 2) {
72:         printf("usage: %s <connection file>\n", argv[0]);
73:         MPI_Abort(MPI_COMM_WORLD, -1);
74:     }
75:
76:     init_state(argv[1]);
77:
78:     /* make input arrays the windows */
79:     MPI_Win_create(inarray, inarray_count*sizeof(int),sizeof(int),
80:                    MPI_INFO_NULL, MPI_COMM_WORLD, &win);
81:     setup_groups();             /* only needed for pscw */
82:
83:     for (itercount = 0; itercount < max_steps; itercount++) {
84:         /* dump_window(); */
85:         compute_state();
86:         printf("state for rank %d at iteration %d is %d\n",
87:                myrank, itercount, state);
88:         MPI_Win_post(ingroup, 0, win);
89:         MPI_Win_start(outgroup, 0, win);
90:         if (ready_to_fire()) {
```

```
 91: /* SLIDE: pNeo Code Walkthrough */
 92:             output_spikes();
 93:             reset_state();
 94:         }
 95:         MPI_Win_complete(win);
 96:         MPI_Win_wait(win);
 97:     }
 98:
 99:     MPI_Win_free(&win);
100:     MPI_Finalize();
101:     return 0;
102: }
103:
104: void init_state(char *filename)
105: {
106: #   define MAX_CONNECTIONS 10000
107: #   define MAX(x,y) ((x) > (y) ? x : y)
108:
109:     /* array of connections described in connection file */
110:     typedef struct {
111:         int source;
112:         int dest;
113:     } connection;
114:
115:     connection connarray[MAX_CONNECTIONS];
116:
117:     FILE *confile;
118:     int i, j, k, n, connarray_count, dispcnt;
119:     int maxpair, maxcell;
120:
121:     if ((confile = fopen(filename, "r")) == NULL) {
122:         printf("could not open connection file %s\n", filename);
```

```
123: /* SLIDE: pNeo Code Walkthrough */
124:         MPI_Abort(MPI_COMM_WORLD, -1);
125:     }
126:
127:     /* The following code should be executed only by process 0,
128:      * which then should broadcast the connarray */
129:     maxcell = connarray_count = i = 0;
130:     inarray_count = outarray_count = 0;
131:
132:     while ((fscanf(confile, "%d %d", &connarray[i].source,
133:                    &connarray[i].dest)) == 2) {
134:         connarray_count++;
135:         maxpair = MAX(connarray[i].source, connarray[i].dest);
136:         if (maxpair > maxcell)
137:             maxcell = maxpair;
138:         if (connarray[i].dest == myrank)
139:             inarray_count++;
140:         if (connarray[i].source == myrank)
141:             outarray_count++;
142:         i++;
143:     }
144:     if (maxcell > numprocs-1) {
145:         printf("%d processes needed for file %s\n",
146:                maxcell+1, filename);
147:         MPI_Abort(MPI_COMM_WORLD, -1);
148:     }
149:
150:     /* dump connarray, for debugging */
151:     if (myrank == 0) {
152:         printf("global connections:\n");
153:         for (i = 0; i < connarray_count; i++)
154:             printf("%d %d\n", connarray[i].source,
```

```
155: /* SLIDE: pNeo Code Walkthrough */
156:                     connarray[i].dest);
157:     }
158:
159:     inarray  = (int *)
160:                     malloc(inarray_count * sizeof(int));
161:     outarray = (outconnection *)
162:                     malloc(outarray_count * sizeof(outconnection));
163:     /* the foll. two arrays are only used in the pscw version */
164:     inranks  = (int *) malloc(inarray_count * sizeof(int));
165:     outranks = (int *) malloc(outarray_count * sizeof(int));
166:
167:     for (i = j = k = 0 ; i < connarray_count; i++) {
168:         if (connarray[i].dest == myrank) {
169:             inranks[j] = connarray[i].source;
170:             j++;
171:         }
172:         inarray_count = j;
173:
174:         if (connarray[i].source == myrank) {
175:             dispcnt = 0;
176:             for (n = 0; n < connarray_count; n++) {
177:                 if (connarray[n].dest == connarray[i].dest) {
178:                     if (connarray[n].source == myrank)
179:                         break;
180:                     else
181:                         dispcnt++; /* increment counter on
182:                                       destination process i */
183:                 }
184:             }
185:             outarray[k].dest = connarray[i].dest;
186:             outarray[k].disp = dispcnt;
```

```
/* SLIDE: pNeo Code Walkthrough */
            outranks[k] = connarray[i].dest; /* pscw only */
            k++;
        }
        outarray_count = k;
    }

    /* dump_local_arrays(); */

    state = (myrank + 5) % numprocs; /* essentially random
                                        for this example */
    for (j = 0; j < inarray_count; j++)
        inarray[j] = 0;          /* no incoming spikes to start */

}

void setup_groups()
{
    MPI_Group worldgroup;

    MPI_Comm_group(MPI_COMM_WORLD, &worldgroup);
    MPI_Group_incl(worldgroup, inarray_count, inranks, &ingroup);
    MPI_Group_incl(worldgroup, outarray_count, outranks,
                   &outgroup);
}
```

```c
/* SLIDE: pNeo Code Walkthrough */
void compute_state()
{
    int i;
    int num_incoming = 0;

    for (i = 0; i < inarray_count; i++)
        num_incoming += inarray[i];

    state = state + num_incoming + 1;

    for (i = 0; i < inarray_count; i++)
        inarray[i] = 0;          /* reset spikes */
}

void reset_state()
{
    state = 0;
}

int ready_to_fire()
{
    if (state > 4)
        return 1;
    else
        return 0;
}
```

```
241: /* SLIDE: pNeo Code Walkthrough */
242: void output_spikes()
243: {
244:     int i;
245:     /* Note: the static declaration here is important. */
246:     static int spike = 1;       /* constant values for spikes */
247:
248:     for (i = 0; i < outarray_count; i++) {
249:         printf("putting spike from %d to %d in iteration %d\n",
250:                myrank, outarray[i].dest, itercount);
251:
252:         MPI_Put(&spike, 1, MPI_INT, outarray[i].dest,
253:                 outarray[i].disp, 1, MPI_INT, win);
254:     }
255: }
256:
```

```
257: /* SLIDE: pNeo Code Walkthrough */
258: void dump_local_arrays()
259: {
260:     int i;
261:
262:     printf("inarray for process %d: ", myrank);
263:     for (i = 0; i < inarray_count; i++)
264:         printf("%d \n", inarray[i]);
265:     printf("\n");
266:     printf("inranks for process %d: ", myrank);
267:     for (i = 0; i < inarray_count; i++)
268:         printf("%d ", inranks[i]);
269:     printf("\n");
270:     printf("outarray for process %d:\n", myrank);
271:     for (i = 0; i < outarray_count; i++)
272:         printf("%d %d\n", outarray[i].dest,
273:                 outarray[i].disp);
274:     printf("outranks for process %d: ", myrank);
275:     for (i = 0; i < outarray_count; i++)
276:         printf("%d ", outranks[i]);
277:     printf("\n");
278: }
279:
280: void dump_window()
281: {
282:     int i;
283:     printf("inarray for rank %d: ", myrank);
284:     for (i = 0; i < inarray_count; i++)
285:         printf("%d ", inarray[i]);
286:     printf("\n");
287: }
```

```
 1: /* SLIDE: Mutex Code Walkthrough */
 2: /*
 3:  * Copyright (C) 2004 University of Chicago.
 4:  * See COPYRIGHT notice in top-level directory.
 5:  */
 6:
 7: /* mpimutex implementation using hybrid algorithm */
 8:
 9: #include <mpi.h>
10: #include <stdio.h>
11: #include <stdlib.h>
12: #include <string.h>
13: #include <assert.h>
14:
15: #include "mpimutex.h"
16:
17: #define WAKEUP 13
18:
19: int MPIMUTEX_Create(int homerank, MPI_Comm comm,
20:                      mpimutex_t *mutex_p)
21: {
22:     int nprocs, myrank, mpi_err;
23:     mpimutex_t mutex = NULL;
24:
25:     int blklens[2];
26:     MPI_Aint disps[2];
27:
```

```
28: /* SLIDE: Mutex Code Walkthrough */
29:     /* allocate mpimutex structure, fill in */
30:     mutex = malloc(sizeof(struct mpimutex));
31:     if (!mutex) goto err_return;
32:     mutex->homerank = homerank;
33:     mutex->waitlistwin = MPI_WIN_NULL;
34:     mutex->waitlisttype = MPI_DATATYPE_NULL;
35:     mutex->waitlist = NULL;
36:     mutex->comm = MPI_COMM_NULL;
37:
38:     MPI_Comm_rank(comm, &myrank);
39:     MPI_Comm_size(comm, &nprocs);
40:     mutex->myrank = myrank;
41:     mutex->nprocs = nprocs;
42:
43:     blklens[0] = mutex->myrank;
44:     disps[0]   = 0;
45:     blklens[1] = mutex->nprocs - mutex->myrank - 1;
46:     disps[1]   = mutex->myrank + 1;
47:
48:     /* build type once */
49:     mpi_err = MPI_Type_indexed(2, blklens, disps, MPI_BYTE,
50:                                &mutex->waitlisttype);
51:     if (mpi_err != MPI_SUCCESS) goto err_return;
52:     mpi_err = MPI_Type_commit(&mutex->waitlisttype);
53:     if (mpi_err != MPI_SUCCESS) goto err_return;
54:
55:     MPI_Comm_dup(comm, &mutex->comm);
56:     if (mpi_err != MPI_SUCCESS) goto err_return;
```

```
/* SLIDE: Mutex Code Walkthrough */
    if (myrank == homerank) {
        /* allocate waitlist on this proc */
        mpi_err = MPI_Alloc_mem(nprocs, MPI_INFO_NULL,
                                &mutex->waitlist);
        if (mpi_err != MPI_SUCCESS) goto err_return;
        memset(mutex->waitlist, 0, nprocs);

        mpi_err = MPI_Win_create(mutex->waitlist, nprocs, 1,
                                 MPI_INFO_NULL, mutex->comm,
                                 &mutex->waitlistwin);
    }
    else {
        /* this process contributes nothing */
        mpi_err = MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL,
                                 mutex->comm,
                                 &mutex->waitlistwin);
    }
    if (mpi_err != MPI_SUCCESS) goto err_return;

    *mutex_p = mutex;
    return MPI_SUCCESS;
```

```
/* SLIDE: Mutex Code Walkthrough */
 err_return:
     if (mutex) {
         /* clean up all the structures we allocated */
         if (mutex->waitlistwin != MPI_WIN_NULL)
             MPI_Win_free(&mutex->waitlistwin);
         if (mutex->waitlist)
             MPI_Free_mem(mutex->waitlist);
         if (mutex->waitlisttype != MPI_DATATYPE_NULL)
             MPI_Type_free(&mutex->waitlisttype);
         if (mutex->comm != MPI_COMM_NULL)
             MPI_Comm_free(&mutex->comm);
         free(mutex);
     }
     return MPI_ERR_UNKNOWN;
}
```

```
 96: /* SLIDE: Mutex Code Walkthrough */
 97: int MPIMUTEX_Lock(mpimutex_t mutex)
 98: {
 99:     int mpi_err, i;
100:     unsigned char val = 1;
101:     unsigned char *waitlistcopy = NULL;
102:
103:     waitlistcopy = malloc(mutex->nprocs-1);
104:     if (!waitlistcopy) goto err_return;
105:
106:     /* add self to waitlist */
107:     mpi_err = MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->homerank, 0,
108:                            mutex->waitlistwin);
109:     if (mpi_err != MPI_SUCCESS) goto err_return;
110:
111:     mpi_err = MPI_Get(waitlistcopy, mutex->nprocs-1, MPI_BYTE,
112:                       mutex->homerank, 0, 1, mutex->waitlisttype,
113:                       mutex->waitlistwin);
114:     if (mpi_err != MPI_SUCCESS) goto err_return;
115:     mpi_err = MPI_Put(&val, 1, MPI_BYTE,
116:                       mutex->homerank, mutex->myrank, 1, MPI_BYTE,
117:                       mutex->waitlistwin);
118:     if (mpi_err != MPI_SUCCESS) goto err_return;
119:
120:     mpi_err = MPI_Win_unlock(mutex->homerank, mutex->waitlistwin);
121:     if (mpi_err != MPI_SUCCESS) goto err_return;
122:
```

```
123: /* SLIDE: Mutex Code Walkthrough */
124:     /* check to see if lock is already held */
125:     for (i=0; i < (mutex->nprocs-1) && waitlistcopy[i] == 0; i++);
126:
127:     if (i < mutex->nprocs - 1) {
128:         /* wait for notification from some other process */
129:         mpi_err = MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,
130:                            WAKEUP, mutex->comm,
131:                            MPI_STATUS_IGNORE);
132:         if (mpi_err != MPI_SUCCESS) goto err_return;
133:     }
134:
135:     free(waitlistcopy);
136:
137:     return MPI_SUCCESS;
138:
139:  err_return:
140:     printf("error!\n");
141:     if (waitlistcopy) free(waitlistcopy);
142:
143:     return MPI_ERR_UNKNOWN;
144: }
145:
```

```c
/* SLIDE: Mutex Code Walkthrough */
int MPIMUTEX_Unlock(mpimutex_t mutex)
{
    int mpi_err, i;
    unsigned char val = 0;
    unsigned char *waitlistcopy;

    /* TODO: allocate waitlistcopy only once */
    waitlistcopy = malloc(mutex->nprocs-1);
    if (!waitlistcopy) goto err_return;

    /* remove self from waitlist */
    mpi_err = MPI_Win_lock(MPI_LOCK_EXCLUSIVE, mutex->homerank, 0,
                           mutex->waitlistwin);
    if (mpi_err != MPI_SUCCESS) goto err_return;

    mpi_err = MPI_Get(waitlistcopy, mutex->nprocs-1, MPI_BYTE,
                      mutex->homerank, 0, 1, mutex->waitlisttype,
                      mutex->waitlistwin);
    if (mpi_err != MPI_SUCCESS) goto err_return;
    mpi_err = MPI_Put(&val, 1, MPI_BYTE,
                      mutex->homerank, mutex->myrank, 1, MPI_BYTE,
                      mutex->waitlistwin);
    if (mpi_err != MPI_SUCCESS) goto err_return;

    mpi_err = MPI_Win_unlock(mutex->homerank, mutex->waitlistwin);
    if (mpi_err != MPI_SUCCESS) goto err_return;
```

```
173: /* SLIDE: Mutex Code Walkthrough */
174:     /* check to see if someone is waiting for the lock */
175:     for (i=0; i < (mutex->nprocs-1) && waitlistcopy[i] == 0; i++);
176:
177:     if (i < mutex->nprocs - 1) {
178:         int nextrank;
179:
180:         /* find next rank waiting for the lock.  we start with
181:          * rank after ours and look in order to ensure fairness.
182:          */
183:         nextrank = mutex->myrank;
184:         while (nextrank < (mutex->nprocs - 1) &&
185:                waitlistcopy[nextrank] == 0)
186:         {
187:             nextrank++;
188:         }
189:         if (nextrank < mutex->nprocs - 1) {
190:             /* nextrank is off by one from skipping local byte */
191:             nextrank++;
192:         }
193:         else {
194:             nextrank = 0;
195:             while (nextrank < mutex->myrank &&
196:                    waitlistcopy[nextrank] == 0)
197:             {
198:                 nextrank++;
199:             }
200:
201:             assert(nextrank != mutex->myrank);
202:         }
203:
204:
```

```
        /* notify next rank */
        mpi_err = MPI_Send(NULL, 0, MPI_BYTE, nextrank, WAKEUP,
                           mutex->comm);
        if (mpi_err != MPI_SUCCESS) goto err_return;
    }

    free(waitlistcopy);

    return MPI_SUCCESS;

 err_return:
    printf("error!\n");
    if (waitlistcopy) free(waitlistcopy);
    return MPI_ERR_UNKNOWN;
}
```

```
220: /* SLIDE: Mutex Code Walkthrough */
221: int MPIMUTEX_Free(mpimutex_t *mutex_p)
222: {
223:     mpimutex_t mutex = *mutex_p;
224:
225:     if (mutex) {
226:         if (mutex->waitlistwin != MPI_WIN_NULL) {
227:             MPI_Win_free(&mutex->waitlistwin);
228:         }
229:
230:         if (mutex->waitlist) {
231:             MPI_Free_mem(mutex->waitlist);
232:         }
233:
234:         if (mutex->waitlisttype != MPI_DATATYPE_NULL) {
235:             MPI_Type_free(&mutex->waitlisttype);
236:         }
237:
238:         if (mutex->comm != MPI_COMM_NULL) {
239:             MPI_Comm_free(&mutex->comm);
240:         }
241:
242:         free(mutex);
243:     }
244:
245:     *mutex_p = NULL;
246:
247:     return MPI_SUCCESS;
248: }
```

```c
/* SLIDE: Main Program with OpenMP */
/* -*- 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 <string.h>

#include <mpi.h>
#include <omp.h>

#include "mlife.h"
#include "mlife-io.h"

int inline MLIFE_nextstate(int **matrix, int y, int x);
static int MLIFE_parse_args(int argc, char **argv);
double MLIFE_Sweep( int **matrix, int **temp,
                    int myrows, int rows, int cols, int ntimes,
                    char opt_prefix[64], MPI_Comm comm );

/* options */
static int opt_rows = 25, opt_cols = 70, opt_iter = 10;
static int opt_restart_iter = -1;
static char opt_prefix[64] = "mlife";

```

```
29: /* SLIDE: Main Program with OpenMP */
30: int main(int argc, char *argv[])
31: {
32:     int rank;
33:     double time;
34:
35:     MPI_Init(&argc, &argv);
36:     MLIFE_parse_args(argc, argv);
37:     MLIFEIO_Init(MPI_COMM_WORLD);
38:
39:     time = life(opt_rows, opt_cols, opt_iter, MPI_COMM_WORLD);
40:
41:     /* print the total time taken */
42:     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
43:     if (rank == 0)
44:         printf("[%d] Life finished in %lf secs of calculation\n",
45:                rank, time);
46:
47:     MLIFEIO_Finalize();
48:     MPI_Finalize();
49:
50:     return 0;
51: }
52:
```

```
53: /* SLIDE: Main Program with OpenMP */
54: double life(int rows, int cols, int ntimes, MPI_Comm comm)
55: {
56:     int      err, i, j, k, rank, nprocs, next, prev;
57:     int      myrows, myoffset;
58:     int    **matrix, **temp, **addr;
59:     int     *mdata, *tdata;
60:     double   mytime, totaltime=0.0, starttime;
61:
62:     MPI_Comm_size(comm, &nprocs);
63:     MPI_Comm_rank(comm, &rank);
64:
65:     /* set neighbors */
66:     if (rank == 0)
67:         prev = MPI_PROC_NULL;
68:     else
69:         prev = rank-1;
70:     if (rank == nprocs - 1)
71:         next = MPI_PROC_NULL;
72:     else
73:         next = rank+1;
74:
75:     /* determine my part of the matrix, row-block distribution */
76:     myrows   = MLIFE_myrows(rows, rank, nprocs);
77:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
78:
79:     /* allocate memory for the matrix using single blocks */
80:     matrix = (int **) malloc((myrows+2) * sizeof(int *));
81:     temp   = (int **) malloc((myrows+2) * sizeof(int *));
82:     mdata  = (int *) malloc((myrows+2) * (cols+2) * sizeof(int));
83:     tdata  = (int *) malloc((myrows+2) * (cols+2) * sizeof(int));
```

```
 84: /* SLIDE: Main Program with OpenMP */
 85:     /* set up pointers for convenience */
 86:     matrix[0] = mdata;
 87:     temp[0]   = tdata;
 88:     for (i = 1; i < myrows+2; i++) {
 89:         matrix[i] = matrix[i-1] + cols + 2;
 90:         temp[i]   = temp[i-1] + cols + 2;
 91:     }
 92:
 93:     /* Use this pattern (same schedule as used in the compute
 94:        loop) to have the "owning" thread be the first to touch
 95:        an element. */
 96:
 97: #ifdef USE_FIRST_TOUCH
 98: #pragma omp parallel for default(none) private(i,j) \
 99:     firstprivate(myrows,cols) \
100:     shared(temp,matrix)
101:     for (i = 1; i <= myrows; i++) {
102:         for (j = 1; j < cols+1; j++) {
103:             temp[i][j]   = DIES;
104:             matrix[i][j] = DIES;
105:         }
106:     }
107: #endif
108:
109:     /* initialize the boundaries of the life matrix */
110:     for (j = 0; j < cols+2; j++) {
111:         matrix[0][j] = matrix[myrows+1][j] = temp[0][j]
112:                      = temp[myrows+1][j] = DIES;
113:     }
114:     for (i = 0; i < myrows+2; i++) {
115:         matrix[i][0] = matrix[i][cols+1] = temp[i][0]
```

```
                        = temp[i][cols+1] = DIES;
    }

    if (opt_restart_iter == -1) {
        /* initialize the life matrix */
        for (i = 1; i <= myrows; i++)  {
            srand48((long)(1000^(i + myoffset)));

            for (j = 1; j<= cols; j++) {
                if (drand48() > 0.5) matrix[i][j] = BORN;
                else                 matrix[i][j] = DIES;
            }
        }
    }
```

```
/* SLIDE: Main Program with OpenMP */
    else if (MLIFEIO_Can_restart()) {
        /* read state from checkpoint file */
        err = MLIFEIO_Restart(opt_prefix, matrix, rows, cols,
                              opt_restart_iter, MPI_INFO_NULL);
        if (err != MPI_SUCCESS) {
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_IO);
        }
    }
    else {
        fprintf(stderr, "Restart unsupported by I/O backend.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_IO);
    }

    MLIFE_exchange_init(comm, &matrix[0][0], &temp[0][0],
                        myrows, rows, cols, prev, next);

    MPI_Barrier( MPI_COMM_WORLD );
    /* use portable MPI function for timing */

    totaltime = MLIFE_Sweep( matrix, temp, myrows, rows, cols,
                             ntimes, opt_prefix, comm );

    MLIFE_exchange_finalize();
    free(matrix);
    free(temp);
    free(mdata);
    free(tdata);

    return(totaltime/(double) nprocs);
}
```

```
/* SLIDE: Main Program with OpenMP */
int inline MLIFE_nextstate(int **matrix, int row, int col)
{
    int sum;

    /* add values of all eight neighbors */
    sum = matrix[row-1][col-1] + matrix[row-1][col] +
          matrix[row-1][col+1] + matrix[row][col-1] +
          matrix[row][col+1] + matrix[row+1][col-1] +
          matrix[row+1][col] + matrix[row+1][col+1];

    if (sum < 2 || sum > 3) return DIES;
    else if (sum == 3)      return BORN;
    else                    return matrix[row][col];
}

int MLIFE_myrows(int rows, int rank, int nprocs)
{
    int myrows;

    myrows = rows / nprocs;

    if (rank < (rows % nprocs)) myrows += 1;

    return myrows;
}
```

```
/* SLIDE: Main Program with OpenMP */
int MLIFE_myrowoffset(int rows, int rank, int nprocs)
{
    int myoffset;

    myoffset = rank * (rows / nprocs);

    if (rank > (rows % nprocs)) myoffset += (rows % nprocs);
    else                        myoffset += rank;

    return myoffset;
}

```

```
/* SLIDE: Main Program with OpenMP */
/* MLIFE_parse_args
 *
 * Note: Command line arguments are not guaranteed in the MPI
 *       environment to be passed to all processes.  To be
 *       portable, we must process on rank 0 and distribute
 *       results.
 */
static int MLIFE_parse_args(int argc, char **argv)
{
    int ret;
    int rank;
    int myargs[5]; /* array for simple sending of arguments */

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        while ((ret = getopt(argc, argv, "x:y:i:p:r:")) >= 0)
        {
            switch(ret) {
                case 'x':
                    opt_cols = atoi(optarg);
                    break;
                case 'y':
                    opt_rows = atoi(optarg);
                    break;
                case 'i':
                    opt_iter = atoi(optarg);
                    break;
                case 'r':
                    opt_restart_iter = atoi(optarg);
```

```
/* SLIDE: Main Program with OpenMP */
                case 'p':
                    strncpy(opt_prefix, optarg, 63);
                    break;
                default:
                    break;
            }
        }

        myargs[0] = opt_rows;
        myargs[1] = opt_cols;
        myargs[2] = opt_iter;
        myargs[3] = opt_restart_iter;
        myargs[4] = strlen(opt_prefix) + 1;
    }

    MPI_Bcast(myargs, 5, MPI_INT, 0, MPI_COMM_WORLD);
    opt_rows = myargs[0];
    opt_cols = myargs[1];
    opt_iter = myargs[2];

    MPI_Bcast(opt_prefix, myargs[4], MPI_CHAR, 0, MPI_COMM_WORLD);

    return 0;
}
```

```
 1: /* SLIDE: Implementing the Sweep */
 2: #include "mpi.h"
 3: #include <omp.h>
 4:
 5: double MLIFE_Sweep( int **matrix, int **temp,
 6:                     int myrows, int rows, int cols, int ntimes,
 7:                     char opt_prefix[64], MPI_Comm comm )
 8: {
 9:     int i, j, k, err;
10:     int **addr;
11:     double mytime, totaltime=0.0, starttime;
12:
13:     starttime = MPI_Wtime();
14:
15: #pragma omp parallel default(none) \
16:  private(k,i,j) firstprivate(ntimes,myrows,rows,cols,opt_prefix) \
17:     firstprivate(temp,matrix) private(addr)
18:     for (k = 0; k < ntimes; k++)
19:     {
20: #pragma omp single
21:         MLIFE_exchange(matrix, myrows, cols);
22: #pragma omp barrier
23:
24:         /* calculate new state for all non-boundary elements */
25: #pragma omp for
26:         for (i = 1; i <= myrows; i++) {
27:             for (j = 1; j < cols+1; j++) {
28:                 temp[i][j] = MLIFE_nextstate(matrix, i, j);
29:             }
30:         }
```

```
31: /* SLIDE: Implementing the Sweep */
32:         /* swap the matrices */
33:         addr   = matrix;
34:         matrix = temp;
35:         temp   = addr;
36:
37:         /* Exploit wait at end of previous omp for */
38: #pragma omp single private(err)
39:         if (0) {
40:          err = MLIFEIO_Checkpoint(opt_prefix, matrix, rows, cols,
41:                                   k, MPI_INFO_NULL);
42:             }
43: #pragma omp barrier
44:     }
45:
46:     /* return the average time taken/processor */
47:     mytime = MPI_Wtime() - starttime;
48:     MPI_Reduce(&mytime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0,
49:                comm);
50:
51:     return totaltime;
52: }
```

```c
/* SLIDE: Implementing the Sweep: The Simple Version */
#include "mpi.h"
#include <omp.h>

double MLIFE_Sweep( int **matrix, int **temp,
                    int myrows, int rows, int cols, int ntimes,
                    char opt_prefix[64], MPI_Comm comm )
{
    int i, j, k, err;
    int **addr;
    double mytime, totaltime=0.0, starttime;

    starttime = MPI_Wtime();

    for (k = 0; k < ntimes; k++)
    {
        MLIFE_exchange(matrix, myrows, cols);

        /* calculate new state for all non-boundary elements */
#pragma omp parallel for default(none) private(i,j) \
  firstprivate(myrows,cols)                          \
  shared(temp,matrix)
        for (i = 1; i <= myrows; i++) {
            for (j = 1; j < cols+1; j++) {
                temp[i][j] = MLIFE_nextstate(matrix, i, j);
            }
        }

```

```
29: /* SLIDE: Implementing the Sweep: The Simple Version */
30:         /* swap the matrices */
31:         addr   = matrix;
32:         matrix = temp;
33:         temp   = addr;
34:
35:         /* Exploit wait at end of previous omp for */
36:         if (0) {
37:          err = MLIFEIO_Checkpoint(opt_prefix, matrix, rows, cols,
38:                                   k, MPI_INFO_NULL);
39:             }
40:     }
41:
42:     /* return the average time taken/processor */
43:     mytime = MPI_Wtime() - starttime;
44:     MPI_Reduce(&mytime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0,
45:                comm);
46:
47:     return totaltime;
48: }
```

```
 1: /* SLIDE: Implementing the Sweep: Adding Comm/Comp Overlap */
 2: #include "mpi.h"
 3: #include <omp.h>
 4:
 5: double MLIFE_Sweep( int **matrix, int **temp,
 6:                     int myrows, int rows, int cols, int ntimes,
 7:                     char opt_prefix[64], MPI_Comm comm )
 8: {
 9:     int i, j, k, err;
10:     int **addr;
11:     double mytime, totaltime=0.0, starttime;
12:
13:     starttime = MPI_Wtime();
14:
15: #pragma omp parallel default(none) \
16:  private(k,i,j) firstprivate(ntimes,myrows,rows,cols,opt_prefix) \
17:     firstprivate(temp,matrix) private(addr)
18:     for (k = 0; k < ntimes; k++)
19:     {
20: #pragma omp single nowait
21:         MLIFE_exchange(matrix, myrows, cols);
22:
23:         /* calculate new state for all non-boundary elements and
24:            all that do not involve the boundary rows */
25: #pragma omp for schedule(dynamic)
26:         for (i = 2; i < myrows; i++) {
27:             for (j = 1; j < cols+1; j++) {
28:                 temp[i][j] = MLIFE_nextstate(matrix, i, j);
29:             }
30:         }
31:
32: #pragma omp barrier
```

```
33: /* SLIDE: Implementing the Sweep: Adding Comm/Comp Overlap */
34:
35:         /* We might want this to be an omp single - there is much
36:            less work here */
37: #pragma omp for
38:         for (j = 1; j < cols+1; j++) {
39:           temp[1][j] = MLIFE_nextstate(matrix, 1, j);
40:           temp[myrows][j] = MLIFE_nextstate(matrix, 1, j );
41:         }
42:
43:         /* swap the matrices */
44:         addr   = matrix;
45:         matrix = temp;
46:         temp   = addr;
47:
48:         /* Exploit wait at end of previous omp for */
49: #pragma omp single private(err)
50:         if (0) {
51:           err = MLIFEIO_Checkpoint(opt_prefix, matrix, rows, cols,
52:                                    k, MPI_INFO_NULL);
53:             }
54: #pragma omp barrier
55:     }
56:
57:     /* return the average time taken/processor */
58:     mytime = MPI_Wtime() - starttime;
59:     MPI_Reduce(&mytime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0,
60:                comm);
61:
62:     return totaltime;
63: }
```