1 Introduction to Parallel Algorithms

1.1 Odd-Even Sort

- A complete message-passing program that will sort a list of numbers using the odd-even sorting algorithm.
- The odd-even sorting algorithm sorts a sequence of n elements using p processes in a total of p phases.
- During each of these phases, the odd-or even-numbered processes perform a compare-split step with their right neighbors.
- The MPI program for performing the odd-even sort in parallel is shown below. To simplify the presentation, this program assumes that n is divisible by p.

```
 1  #include <stdlib.h>
 2  #include <mpi.h> /* Include MPI's header file */
 3
 4  main(int argc, char *argv[])
 5  {
 6    int n;         /* The total number of elements to be sorted */
 7    int npes;      /* The total number of processes */
 8    int myrank;    /* The rank of the calling process */
 9    int nlocal;    /* The local number of elements, and the array that stores them */
10    int *elmnts;   /* The array that stores the local elements */
11    int *relmnts;  /* The array that stores the received elements */
12    int oddrank;   /* The rank of the process during odd-phase communication */
13    int evenrank;  /* The rank of the process during even-phase communication */
14    int *wspace;   /* Working space during the compare-split operation */
15    int i;
16    MPI_Status status;
17
18    /* Initialize MPI and get system information */
19    MPI_Init(&argc, &argv);
20    MPI_Comm_size(MPI_COMM_WORLD, &npes);
21    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22
23    n = atoi(argv[1]);
24    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
25
26    /* Allocate memory for the various arrays */
27    elmnts  = (int *)malloc(nlocal*sizeof(int));
```

```
28    relmnts = (int *)malloc(nlocal*sizeof(int));
29    wspace  = (int *)malloc(nlocal*sizeof(int));
30
31    /* Fill-in the elmnts array with random elements */
32    srandom(myrank);
33    for (i=0; i<nlocal; i++)
34      elmnts[i] = random();
35
36    /* Sort the local elements using the built-in quicksort routine */
37    qsort(elmnts, nlocal, sizeof(int), IncOrder);
38
39    /* Determine the rank of the processors that myrank needs to communicate during */
40    /* the odd and even phases of the algorithm */
41    if (myrank%2 == 0) {
42      oddrank  = myrank-1;
43      evenrank = myrank+1;
44    }
45    else {
46      oddrank  = myrank+1;
47      evenrank = myrank-1;
48    }
49
50    /* Set the ranks of the processors at the end of the linear */
51    if (oddrank == -1 || oddrank == npes)
52      oddrank = MPI_PROC_NULL;
53    if (evenrank == -1 || evenrank == npes)
54      evenrank = MPI_PROC_NULL;
55
56    /* Get into the main loop of the odd-even sorting algorithm */
57    for (i=0; i<npes-1; i++) {
58      if (i%2 == 1) /* Odd phase */
59        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
60            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
61      else /* Even phase */
62        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
63            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
64
65      CompareSplit(nlocal, elmnts, relmnts, wspace,
66                   myrank < status.MPI_SOURCE);
67    }
68
69    free(elmnts); free(relmnts); free(wspace);
70    MPI_Finalize();
```

```
71  }
72
73  /* This is the CompareSplit function */
74  CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
75               int keepsmall)
76  {
77    int i, j, k;
78
79    for (i=0; i<nlocal; i++)
80      wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */
81
82    if (keepsmall) { /* Keep the nlocal smaller elements */
83      for (i=j=k=0; k<nlocal; k++) {
84        if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
85          elmnts[k] = wspace[i++];
86        else
87          elmnts[k] = relmnts[j++];
88      }
89    }
90    else { /* Keep the nlocal larger elements */
91      for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
92        if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
93          elmnts[k] = wspace[i--];
94        else
95          elmnts[k] = relmnts[j--];
96      }
97    }
98  }
99
100 /* The IncOrder function that is called by qsort is defined as follows */
101 int IncOrder(const void *e1, const void *e2)
102 {
103   return (*((int *)e1) - *((int *)e2));
104 }
```

1.2 Cannon's Matrix-Matrix Multiplication with MPI's Topologies

- To illustrate how the various topology functions are used.
- Cannon's algorithm views the processes as being arranged in a virtual two-dimensional square array. It uses this array to distribute the matrices A, B, and the result matrix C in a block fashion.

- That is, if $n \times n$ is the size of each matrix and p is the total number of process, then each matrix is divided into square blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ (assuming that p is a perfect square).
- Now, process $P_{i,j}$ in the grid is assigned the $A_{i,j}, B_{i,j}, and C_{i,j}$ blocks of each matrix.
- After an initial data alignment phase, the algorithm proceeds in $\sqrt{p}$ steps. In each step, every process multiplies the locally available blocks of matrices A and B, and then sends the block of A to the leftward process, and the block of B to the upward process.
- The following code segment shows the MPI function that implements Cannon's algorithm.
- The dimension of the matrices is supplied in the parameter n. The parameters a, b, and c point to the locally stored portions of the matrices A, B, and C, respectively.
- The size of these arrays is $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, where p is the number of processes. This routine assumes that p is a perfect square and that n is a multiple of $\sqrt{p}$.
- The parameter *comm* stores the communicator describing the processes that call the MatrixMatrixMultiply function.

```
 1  MatrixMatrixMultiply(int n, double *a, double *b, double *c,
 2                       MPI_Comm comm)
 3  {
 4    int i;
 5    int nlocal;
 6    int npes, dims[2], periods[2];
 7    int myrank, my2drank, mycoords[2];
 8    int uprank, downrank, leftrank, rightrank, coords[2];
 9    int shiftsource, shiftdest;
10    MPI_Status status;
11    MPI_Comm comm_2d;
12
13    /* Get the communicator related information */
14    MPI_Comm_size(comm, &npes);
15    MPI_Comm_rank(comm, &myrank);
16
17    /* Set up the Cartesian topology */
18    dims[0] = dims[1] = sqrt(npes);
```

```
19
20    /* Set the periods for wraparound connections */
21    periods[0] = periods[1] = 1;
22
23    /* Create the Cartesian topology, with rank reordering */
24    MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
25
26    /* Get the rank and coordinates with respect to the new topology */
27    MPI_Comm_rank(comm_2d, &my2drank);
28    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
29
30    /* Compute ranks of the up and left shifts */
31    MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
32    MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
33
34    /* Determine the dimension of the local matrix block */
35    nlocal = n/dims[0];
36
37    /* Perform the initial matrix alignment. First for A and then for B */
38    MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
39    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE, shiftdest,
40        1, shiftsource, 1, comm_2d, &status);
41
42    MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
43    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
44        shiftdest, 1, shiftsource, 1, comm_2d, &status);
45
46    /* Get into the main computation loop */
47    for (i=0; i<dims[0]; i++) {
48      MatrixMultiply(nlocal, a, b, c); /*c=c+a*b*/
49
50      /* Shift matrix a left by one */
51      MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
52          leftrank, 1, rightrank, 1, comm_2d, &status);
53
54      /* Shift matrix b up by one */
55      MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
56          uprank, 1, downrank, 1, comm_2d, &status);
57    }
58
59    /* Restore the original distribution of a and b */
60    MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
61    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
```

```
62        shiftdest, 1, shiftsource, 1, comm_2d, &status);
63
64    MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
65    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
66        shiftdest, 1, shiftsource, 1, comm_2d, &status);
67
68    MPI_Comm_free(&comm_2d); /* Free up communicator */
69  }
70
71  /* This function performs a serial matrix-matrix multiplication c = a*b */
72  MatrixMultiply(int n, double *a, double *b, double *c)
73  {
74    int i, j, k;
75
76    for (i=0; i<n; i++)
77      for (j=0; j<n; j++)
78        for (k=0; k<n; k++)
79          c[i*n+j] += a[i*n+k]*b[k*n+j];
80  }
```

- The following code segment shows the MPI program that implements Cannon's algorithm using non-blocking send and receive operations. The various parameters are identical to those of the previous blocking version.

```
1   MatrixMatrixMultiply_NonBlocking(int n, double *a, double *b,
2                                    double *c, MPI_Comm comm)
3   {
4     int i, j, nlocal;
5     double *a_buffers[2], *b_buffers[2];
6     int npes, dims[2], periods[2];
7     int myrank, my2drank, mycoords[2];
8     int uprank, downrank, leftrank, rightrank, coords[2];
9     int shiftsource, shiftdest;
10    MPI_Status status;
11    MPI_Comm comm_2d;
12    MPI_Request reqs[4];
13
14    /* Get the communicator related information */
15    MPI_Comm_size(comm, &npes);
16    MPI_Comm_rank(comm, &myrank);
17
18    /* Set up the Cartesian topology */
19    dims[0] = dims[1] = sqrt(npes);
```

```
20
21      /* Set the periods for wraparound connections */
22      periods[0] = periods[1] = 1;
23
24      /* Create the Cartesian topology, with rank reordering */
25      MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
26
27      /* Get the rank and coordinates with respect to the new topology */
28      MPI_Comm_rank(comm_2d, &my2drank);
29      MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
30
31      /* Compute ranks of the up and left shifts */
32      MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
33      MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
34
35      /* Determine the dimension of the local matrix block */
36      nlocal = n/dims[0];
37
38      /* Setup the a_buffers and b_buffers arrays */
39      a_buffers[0] = a;
40      a_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
41      b_buffers[0] = b;
42      b_buffers[1] = (double *)malloc(nlocal*nlocal*sizeof(double));
43
44      /* Perform the initial matrix alignment. First for A and then for B */
45      MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource, &shiftdest);
46      MPI_Sendrecv_replace(a_buffers[0], nlocal*nlocal, MPI_DOUBLE,
47          shiftdest, 1, shiftsource, 1, comm_2d, &status);
48
49      MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource, &shiftdest);
50      MPI_Sendrecv_replace(b_buffers[0], nlocal*nlocal, MPI_DOUBLE,
51          shiftdest, 1, shiftsource, 1, comm_2d, &status);
52
53      /* Get into the main computation loop */
54      for (i=0; i<dims[0]; i++) {
55        MPI_Isend(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
56            leftrank, 1, comm_2d, &reqs[0]);
57        MPI_Isend(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
58            uprank, 1, comm_2d, &reqs[1]);
59        MPI_Irecv(a_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
60            rightrank, 1, comm_2d, &reqs[2]);
61        MPI_Irecv(b_buffers[(i+1)%2], nlocal*nlocal, MPI_DOUBLE,
62            downrank, 1, comm_2d, &reqs[3]);
```

```
63
64        /* c = c + a*b */
65        MatrixMultiply(nlocal, a_buffers[i%2], b_buffers[i%2], c);
66
67        for (j=0; j<4; j++)
68          MPI_Wait(&reqs[j], &status);
69      }
70
71      /* Restore the original distribution of a and b */
72      MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource, &shiftdest);
73      MPI_Sendrecv_replace(a_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
74          shiftdest, 1, shiftsource, 1, comm_2d, &status);
75
76      MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource, &shiftdest);
77      MPI_Sendrecv_replace(b_buffers[i%2], nlocal*nlocal, MPI_DOUBLE,
78          shiftdest, 1, shiftsource, 1, comm_2d, &status);
79
80      MPI_Comm_free(&comm_2d); /* Free up communicator */
81
82      free(a_buffers[1]);
83      free(b_buffers[1]);
84  }
```

1.3 One-Dimensional Matrix-Vector Multiplication

- The following code segment is for multiplying a dense $n \times n$ matrix A with a vector b, i.e., $x = Ab$.
- One way of performing this multiplication in parallel is to have each process compute different portions of the product-vector x. In particular, each one of the p processes is responsible for computing n/p consecutive elements of x.
- This algorithm can be implemented in MPI by distributing the matrix A in a row-wise fashion, such that each process receives the n/p rows that correspond to the portion of the product-vector x it computes. Vector b is distributed in a fashion similar to x.
- Code segment shows the MPI program that uses a row-wise distribution of matrix A. The dimension of the matrices is supplied in the parameter n, the parameters a and b point to the locally stored portions of matrix A and vector b, respectively, and the parameter x points to the local

portion of the output matrix-vector product. This program assumes that n is a multiple of the number of processors.

```
 1  RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
 2                          MPI_Comm comm)
 3  {
 4    int i, j;
 5    int nlocal;        /* Number of locally stored rows of A */
 6    double *fb;        /* Will point to a buffer that stores the entire vector b */
 7    int npes, myrank;
 8    MPI_Status status;
 9
10    /* Get information about the communicator */
11    MPI_Comm_size(comm, &npes);
12    MPI_Comm_rank(comm, &myrank);
13
14    /* Allocate the memory that will store the entire vector b */
15    fb = (double *)malloc(n*sizeof(double));
16
17    nlocal = n/npes;
18
19    /* Gather the entire vector b on each processor using MPI's ALLGATHER operation */
20    MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
21        comm);
22
23    /* Perform the matrix-vector multiplication involving the locally stored submatrix
24    for (i=0; i<nlocal; i++) {
25      x[i] = 0.0;
26      for (j=0; j<n; j++)
27        x[i] += a[i*n+j]*fb[j];
28    }
29
30    free(fb);
31  }
```

- An alternate way of computing x is to parallelize the task of performing the dot-product for each element of x. That is, for each element x_i, of vector x, all the processes will compute a part of it, and the result will be obtained by adding up these partial dot-products.

- This algorithm can be implemented in MPI by distributing matrix A in a column-wise fashion. Each process gets n/p consecutive columns of A, and the elements of vector b that correspond to these columns.

- Furthermore, at the end of the computation we want the product-vector x to be distributed in a fashion similar to vector b. The following code segment shows the MPI program that implements this column-wise distribution of the matrix.

```
1   ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                           MPI_Comm comm)
3   {
4     int i, j;
5     int nlocal;
6     double *px;
7     double *fx;
8     int npes, myrank;
9     MPI_Status status;
10
11    /* Get identity and size information from the communicator */
12    MPI_Comm_size(comm, &npes);
13    MPI_Comm_rank(comm, &myrank);
14
15    nlocal = n/npes;
16
17    /* Allocate memory for arrays storing intermediate results. */
18    px = (double *)malloc(n*sizeof(double));
19    fx = (double *)malloc(n*sizeof(double));
20
21    /* Compute the partial-dot products that correspond to the local columns of A.*/
22    for (i=0; i<n; i++) {
23      px[i] = 0.0;
24      for (j=0; j<nlocal; j++)
25        px[i] += a[i*nlocal+j]*b[j];
26    }
27
28    /* Sum-up the results by performing an element-wise reduction operation */
29    MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31    /* Redistribute fx in a fashion similar to that of vector b */
32    MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33        comm);
34
35    free(px); free(fx);
36  }
```

- Comparing these two programs for performing matrix-vector multiplication we see that the row-wise version needs to perform only a

MPI_Allgather operation whereas the column-wise program needs to perform a MPI_Reduce and a MPI_Scatter operation.

- In general, a row-wise distribution is preferable as it leads to small communication overhead. However, many times, an application needs to compute not only Ax but also $A^T x$. In that case, the row-wise distribution can be used to compute Ax, but the computation of $A^T x$ requires the column-wise distribution (a row-wise distribution of A is a column-wise distribution of its transpose A^T).

- It is much cheaper to use the program for the column-wise distribution than to transpose the matrix and then use the row-wise program.

1.4 Two-Dimensional Matrix-Vector Multiplication

- An alternative way of distributing matrix A is to use a two-dimensional distribution, giving rise to the two-dimensional parallel formulations of the matrix-vector multiplication algorithm.

- The following code segment shows how these topologies and their partitioning are used to implement the two-dimensional matrix-vector multiplication.

- The dimension of the matrix is supplied in the parameter n, the parameters a and b point to the locally stored portions of matrix A and vector b, respectively, and the parameter x points to the local portion of the output matrix-vector product.

- Note that only the processes along the first column of the process grid will store b initially, and that upon return, the same set of processes will store the result x. For simplicity, the program assumes that the number of processes p is a perfect square and that n is a multiple of $\sqrt{p}$.

```
1   MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
2                           MPI_Comm comm)
3   {
4     int ROW=0, COL=1; /* Improve readability */
5     int i, j, nlocal;
6     double *px; /* Will store partial dot products */
7     int npes, dims[2], periods[2], keep_dims[2];
8     int myrank, my2drank, mycoords[2];
9     int other_rank, coords[2];
```

```
10     MPI_Status status;
11     MPI_Comm comm_2d, comm_row, comm_col;
12
13     /* Get information about the communicator */
14     MPI_Comm_size(comm, &npes);
15     MPI_Comm_rank(comm, &myrank);
16
17     /* Compute the size of the square grid */
18     dims[ROW] = dims[COL] = sqrt(npes);
19
20     nlocal = n/dims[ROW];
21
22     /* Allocate memory for the array that will hold the partial dot-products */
23     px = malloc(nlocal*sizeof(double));
24
25     /* Set up the Cartesian topology and get the rank & coordinates of the process in
this topology */
26     periods[ROW] = periods[COL] = 1; /* Set the periods for wrap-around connections */
27
28     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
29
30     MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
31     MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
32
33     /* Create the row-based sub-topology */
34     keep_dims[ROW] = 0;
35     keep_dims[COL] = 1;
36     MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
37
38     /* Create the column-based sub-topology */
39     keep_dims[ROW] = 1;
40     keep_dims[COL] = 0;
41     MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
42
43     /* Redistribute the b vector. */
44     /* Step 1. The processors along the 0th column send their data to the diagonal
processors */
45     if (mycoords[COL] == 0 && mycoords[ROW] != 0) { /* I'm in the first column */
46       coords[ROW] = mycoords[ROW];
47       coords[COL] = mycoords[ROW];
48       MPI_Cart_rank(comm_2d, coords, &other_rank);
49       MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
50     }
```

```
51      if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
52        coords[ROW] = mycoords[ROW];
53        coords[COL] = 0;
54        MPI_Cart_rank(comm_2d, coords, &other_rank);
55        MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d,
56            &status);
57      }
58
59      /* Step 2. The diagonal processors perform a column-wise broadcast */
60      coords[0] = mycoords[COL];
61      MPI_Cart_rank(comm_col, coords, &other_rank);
62      MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
63
64      /* Get into the main computational loop */
65      for (i=0; i<nlocal; i++) {
66        px[i] = 0.0;
67        for (j=0; j<nlocal; j++)
68          px[i] += a[i*nlocal+j]*b[j];
69      }
70
71      /* Perform the sum-reduction along the rows to add up the partial dot-products */
72      coords[0] = 0;
73      MPI_Cart_rank(comm_row, coords, &other_rank);
74      MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,
75          comm_row);
76
77      MPI_Comm_free(&comm_2d); /* Free up communicator */
78      MPI_Comm_free(&comm_row); /* Free up communicator */
79      MPI_Comm_free(&comm_col); /* Free up communicator */
80
81      free(px);
82    }
```