



Communication Optimization for Parallel Processing

Lecture 4

Scatter the Data

- ❑ MPI_Scatter
 - ❑ This routine distributes data to all processors.
 - ❑ The root process sends the data via a send buffer.
 - ❑ All the others provide a receive buffer for the incoming data from the root.



Gather the Data

MPI_Gather

- This routine works the opposite way to MPI_Scatter. It collects data from all processes to the root process.

MPI_Allgather

- Similar to MPI_Gather but all the processes receive the result.



Matrix-Vector Multiplication

- ❑ Use `MPI_Scatter` to distribute the rows of the matrix and the segments of the input vector to all processes.
- ❑ Use `MPI_Allgather` to collect the entire vector in order to perform multiplication.
- ❑ Finally the process 0 uses `MPI_Gather` to collect all the fragments from other processes and prints the result.



Variables and Types

```
#define MAX_ORDER 100
```

```
typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];
```

```
main(int argc, char* argv[]) {
```

```
    int my_rank;
```

```
    int p;
```

```
    LOCAL_MATRIX_T local_A;
```

```
    float global_x[MAX_ORDER];
```

```
    float local_x[MAX_ORDER];
```

```
    float local_y[MAX_ORDER];
```

```
    int m, n;
```

```
    int local_m, local_n;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    if (my_rank == 0) {
```

```
        printf("Enter the order of the matrix (m x n)\n");
```

```
        scanf("%d %d", &m, &n);
```

```
    }
```

Matrix-Vector Multiplication

```
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);  
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
local_m = m/p;  
local_n = n/p;
```

```
Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);  
Print_matrix("We read", local_A, local_m, n, my_rank, p);
```

```
Read_vector("Enter the vector", local_x, local_n, my_rank, p);  
Print_vector("We read", local_x, local_n, my_rank, p);
```

```
Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,  
    local_y, local_m, local_n);  
Print_vector("The product is", local_y, local_m, my_rank, p);
```

```
MPI_Finalize();
```

```
} /* main */
```



Read the Matrix

```
void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A, int local_m,
int n, int my_rank, int p)
{
    int i, j;
    LOCAL_MATRIX_T temp;
    for (i = 0; i < p*local_m; i++)
        for (j = n; j < MAX_ORDER; j++)
            temp[i][j] = 0.0;

    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_m; i++)
            for (j = 0; j < n; j++)
                scanf("%f",&temp[i][j]);
    }
    MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT, local_A, local_m*MAX_ORDER,
MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_matrix */
```



Read the Vector

```
void Read_vector(char* prompt, float local_x[], int local_n, int my_rank, int p)
{
    int i;
    float temp[MAX_ORDER];
    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_n; i++)
            scanf("%f", &temp[i]);
    }
    MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT, 0,
               MPI_COMM_WORLD);
} /* Read_vector */
```



Product Computation

```
void Parallel_matrix_vector_prod(LOCAL_MATRIX_T local_A, int m, int n, float local_x[],
    float global_x[], float local_y[], int local_m, int local_n)
{
    /* local_m = m/p, local_n = n/p */

    int i, j;

    MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT,
        MPI_COMM_WORLD);
    for (i = 0; i < local_m; i++) {
        local_y[i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[i] = local_y[i] + local_A[i][j]*global_x[j];
    }
} /* Parallel_matrix_vector_prod */
```



Print the Matrix

```
void Print_matrix(char* title, LOCAL_MATRIX_T local_A, int local_m, int n , int my_rank, int p)
{
    int i, j;
    float temp[MAX_ORDER][MAX_ORDER];

    MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
              local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("%s\n", title);
        for (i = 0; i < p*local_m; i++) {
            for (j = 0; j < n; j++)
                printf("%4.1f ", temp[i][j]);
            printf("\n");
        }
    }
} /* Print_matrix */
```



Print the Vector

```
void Print_vector(char* title, float local_y[], int local_m, int my_rank, int p)
{
    int i;
    float temp[MAX_ORDER];
    MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m, MPI_FLOAT, 0,
              MPI_COMM_WORLD);
    if (my_rank == 0) {
        printf("%s\n", title);
        for (i = 0; i < p*local_m; i++)
            printf("%4.1f ", temp[i]);
        printf("\n");
    }
} /* Print_vector */
```



MPI Communicator

- ❑ MPI_Comm_group
 - ❑ Create a MPI_GROUP from an existing one.
- ❑ MPI_Group_incl
 - ❑ Add a list of processes into a group.
- ❑ MPI_Comm_create
 - ❑ Create a new MPI_COMM from a MPI_GROUP.



Variables

```
main(int argc, char* argv[]) {
    int p;
    int q; /* = sqrt(p) */
    int my_rank;
    MPI_Group group_world;
    MPI_Group first_row_group;
    MPI_Comm first_row_comm;
    int* process_ranks;
    int proc;
    int test = 0;
    int sum;
    int my_rank_in_first_row;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    q = (int) sqrt((double) p);
    /* Make a list of the processes in the new communicator */
    process_ranks = (int*) malloc(q*sizeof(int));
    for (proc = 0; proc < q; proc++)
        process_ranks[proc] = proc;
```

Communicator Creation

```
/* Get the group underlying MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &group_world);

/* Create the new group */
MPI_Group_incl(group_world, q, process_ranks, &first_row_group);

/* Create the new communicator */
MPI_Comm_create(MPI_COMM_WORLD, first_row_group, &first_row_comm);

/* Now check whether we can do collective ops in first_row_comm */
if (my_rank < q) {
    MPI_Comm_rank(first_row_comm, &my_rank_in_first_row);
    if (my_rank_in_first_row == 0) test = 1;
    MPI_Bcast(&test, 1, MPI_INT, 0, first_row_comm);
}

MPI_Reduce(&test, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (my_rank == 0) {
    printf("q = %d, sum = %d\n", q, sum);
}
MPI_Finalize();
} /* main */
```

Split an MPI Communicator

- ❑ [comm_split.c](#)
- ❑ [MPI_Comm_split](#)
 - ❑ Splits an MPI communicator into many sub-communicator.
 - ❑ Each process specifies its row, and its rank within that row.
 - ❑ The communicator for each row is returned.



Variables

```
main(int argc, char* argv[])
{
    int p;
    int my_rank;
    MPI_Comm my_row_comm;
    int my_row;
    int q;
    int test;
    int my_rank_in_row;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    q = (int) sqrt((double) p);
```



Split a Communicator

```
/* my_rank is rank in MPI_COMM_WORLD. q*q = p */
my_row = my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank, &my_row_comm);

MPI_Comm_rank(my_row_comm, &my_rank_in_row);
if (my_rank_in_row == 0)
    test = my_row;
else
    test = 0;

MPI_Bcast(&test, 1, MPI_INT, 0, my_row_comm);
printf("Process %d > my_row = %d, my_rank_in_row = %d, test = %d\n",
       my_rank, my_row, my_rank_in_row, test);

MPI_Finalize();
} /* main */
```



MPI Cartesian Coordinate

- ❑ MPI_Cart_create
 - ❑ Create a new communicator.
 - ❑ The number of dimension, the dimension sizes along each dimension can be specified.
- ❑ MPI_Cart_cords
 - ❑ Convert a rank into coordinates according to a given communicator.
 - ❑ In the following example we use this routine to calculate the row and column number of a process.
- ❑ MPI_Cart_sub
 - ❑ Split a grid into sub-grids.
 - ❑ We need to specify which dimensions need to be split.



MPI Type Construction

- ❑ Generate portable types for data transmission among different machine architectures.
- ❑ MPI_Type_contiguous
 - ❑ Declare the data type to consist of a given number of contiguous elements.
- ❑ MPI_Address
 - ❑ Calculate the offset.



MPI Type Construction

❑ MPI_Type_struct

❑ Construct a data type using the following.

- ❑ The Number of data blocks
- ❑ The number of data in each block
- ❑ The data type of each element in a block.
- ❑ The offset of each element in a block

❑ MPI_Type_commit

❑ Commit a type to the runtime system.



A Complete Example

- ❑ A parallel matrix multiplication
- ❑ We compute the multiplication of two matrices A and B .
- ❑ The elements of A are broadcast among the rows of process matrix, and the elements of B are shifted among the column of process matrix.



Data Types

```
typedef struct {  
    int p; /* Total number of processes*/  
    MPI_Comm comm; /* Communicator for entire grid */  
    MPI_Comm row_comm; /* Communicator for my row */  
    MPI_Comm col_comm; /* Communicator for my col */  
    int q; /* Order of grid */  
    int my_row; /* My row number */  
    int my_col; /* My column number */  
    int my_rank; /* My rank in the grid comm */  
} GRID_INFO_T;
```

```
#define MAX 65536  
typedef struct {  
    int n_bar;  
#define Order(A) ((A)->n_bar)  
    float entries[MAX];  
#define Entry(A,i,j) (*((A)->entries) + ((A)->n_bar)*(i) + (j))  
} LOCAL_MATRIX_T;
```

```
LOCAL_MATRIX_T* temp_mat;
```

Main Program

```
main(int argc, char* argv[]) {
    int p;
    int my_rank;
    GRID_INFO_T grid;
    LOCAL_MATRIX_T* local_A;
    LOCAL_MATRIX_T* local_B;
    LOCAL_MATRIX_T* local_C;
    int n;
    int n_bar;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    Setup_grid(&grid);
    if (my_rank == 0) {
        printf("What's the order of the matrices?\n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    n_bar = n/grid.q;
```

Multiplication

```
local_A = Local_matrix_allocate(n_bar);  
Order(local_A) = n_bar;  
Read_matrix("Enter A", local_A, &grid, n);  
Print_matrix("We read A =", local_A, &grid, n);
```

```
local_B = Local_matrix_allocate(n_bar);  
Order(local_B) = n_bar;  
Read_matrix("Enter B", local_B, &grid, n);  
Print_matrix("We read B =", local_B, &grid, n);
```

```
Build_matrix_type(local_A);  
temp_mat = Local_matrix_allocate(n_bar);
```

```
local_C = Local_matrix_allocate(n_bar);  
Order(local_C) = n_bar;  
Fox(n, &grid, local_A, local_B, local_C);
```

```
Print_matrix("The product is", local_C, &grid, n);  
Free_local_matrix(&local_A);  
Free_local_matrix(&local_B);  
Free_local_matrix(&local_C);  
MPI_Finalize();  
} /* main */
```


Initializations

```
void Setup_grid (GRID_INFO_T* grid /* out */)
{
    int old_rank;
    int dimensions[2];
    int wrap_around[2];
    int coordinates[2];
    int free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    /* We assume p is a perfect square */
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;

    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];
```

```
/* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_grid */
```

Fox's Algorithm

```
void Fox(int n, GRID_INFO_T*grid, LOCAL_MATRIX_T* local_A,  
        LOCAL_MATRIX_T* local_B, LOCAL_MATRIX_T* local_C)  
{  
    LOCAL_MATRIX_T* temp_A;  
    int stage;  
    int bcast_root;  
    int n_bar; /* n/sqrt(p) */  
    int source;  
    int dest;  
    MPI_Status status;  
  
    n_bar = n/grid->q;  
    Set_to_zero(local_C);  
  
    /* Calculate addresses for circular shift of B */  
    source = (grid->my_row + 1) % grid->q;  
    dest = (grid->my_row + grid->q - 1) % grid->q;
```



Fox's Algorithm

```
/* Set aside storage for the broadcast block of A */
temp_A = Local_matrix_allocate(n_bar);

for (stage = 0; stage < grid->q; stage++) {
    bcast_root = (grid->my_row + stage) % grid->q;
    if (bcast_root == grid->my_col) {
        MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                  bcast_root, grid->row_comm);
        Local_matrix_multiply(local_A, local_B,
                              local_C);
    } else {
        MPI_Bcast(temp_A, 1, local_matrix_mpi_t, bcast_root, grid->row_comm);
        Local_matrix_multiply(temp_A, local_B, local_C);
    }
    MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
                          dest, 0, source, 0, grid->col_comm, &status);
} /* for */
} /* Fox */
```



Allocation/Free Functions

```
LOCAL_MATRIX_T* Local_matrix_allocate(int local_order)
{
    LOCAL_MATRIX_T* temp;

    temp = (LOCAL_MATRIX_T*) malloc(sizeof(LOCAL_MATRIX_T));
    return temp;
} /* Local_matrix_allocate */
```

```
void Free_local_matrix(LOCAL_MATRIX_T** local_A_ptr)
{
    free(*local_A_ptr);
} /* Free_local_matrix */
```



Read the Matrix

```
void Read_matrix(char* prompt, LOCAL_MATRIX_T* local_A, GRID_INFO_T* grid, int n)
{
    int mat_row, mat_col;  int grid_row, grid_col;
    int dest;  int coords[2];
    float* temp;  MPI_Status status;
    if (grid->my_rank != 0)
        for (mat_row = 0; mat_row < Order(local_A); mat_row++)
            MPI_Recv(&Entry(local_A, mat_row, 0), Order(local_A), MPI_FLOAT, 0, 0, grid->comm, &status);
    else {
        temp = (float*) malloc(Order(local_A)*sizeof(float));
        printf("%s\n", prompt);  fflush(stdout);
        for (mat_row = 0; mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q; grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords, &dest);
                if (dest == 0) {
                    for (mat_col = 0; mat_col < Order(local_A); mat_col++)
                        scanf("%f", (local_A->entries)+mat_row*Order(local_A)+mat_col);
                } else {
                    for (mat_col = 0; mat_col < Order(local_A); mat_col++)
                        scanf("%f", temp + mat_col);
                    MPI_Send(temp, Order(local_A), MPI_FLOAT, dest, 0, grid->comm);
                }
            }
        }
        free(temp);
    }
} /* Read_matrix */
```

Print the Matrix

```
void Print_matrix(char* title, LOCAL_MATRIX_T* local_A, GRID_INFO_T* grid, int n)
{
    int mat_row, mat_col;  int grid_row, grid_col;
    int source;  int coords[2];
    float* temp;  MPI_Status status;
    if (grid->my_rank == 0) {
        temp = (float*) malloc(Order(local_A)*sizeof(float));
        printf("%s\n", title);
        for (mat_row = 0; mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q; grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords, &source);
                if (source == 0) {
                    for(mat_col = 0; mat_col < Order(local_A); mat_col++)
                        printf("%4.1f ", Entry(local_A, mat_row, mat_col));
                } else {
                    MPI_Recv(temp, Order(local_A), MPI_FLOAT, source, 0, grid->comm, &status);
                    for(mat_col = 0; mat_col < Order(local_A); mat_col++)
                        printf("%4.1f ", temp[mat_col]);
                }
            }
            printf("\n");
        }
        free(temp);
    } else
        for (mat_row = 0; mat_row < Order(local_A); mat_row++)
            MPI_Send(&Entry(local_A, mat_row, 0), Order(local_A), MPI_FLOAT, 0, 0, grid->comm);
} /* Print_matrix */
```

Build Matrix Data Type

```
void Build_matrix_type(LOCAL_MATRIX_T* local_A )
{
    MPI_Datatype temp_mpi_t;
    int          block_lengths[2];
    MPI_Aint     displacements[2];
    MPI_Datatype typelist[2];
    MPI_Aint     start_address;
    MPI_Aint     address;

    MPI_Type_contiguous(Order(local_A)*Order(local_A), MPI_FLOAT, &temp_mpi_t);
    block_lengths[0] = block_lengths[1] = 1;
    typelist[0] = MPI_INT;
    typelist[1] = temp_mpi_t;

    MPI_Address(local_A, &start_address);
    MPI_Address(&(local_A->n_bar), &address);
    displacements[0] = address - start_address;

    MPI_Address(local_A->entries, &address);
    displacements[1] = address - start_address;

    MPI_Type_struct(2, block_lengths, displacements, typelist, &local_matrix_mpi_t);
    MPI_Type_commit(&local_matrix_mpi_t);
} /* Build_matrix_type */
```

Local Computation

```
void Local_matrix_multiply(LOCAL_MATRIX_T* local_A, LOCAL_MATRIX_T* local_B,  
    LOCAL_MATRIX_T* local_C)  
{  
    int i, j, k;  
  
    for (i = 0; i < Order(local_A); i++)  
        for (j = 0; j < Order(local_A); j++)  
            for (k = 0; k < Order(local_B); k++)  
                Entry(local_C,i,j) = Entry(local_C,i,j) + Entry(local_A,i,k)*Entry(local_B,k,j);  
  
} /* Local_matrix_multiply */
```



Print the Local Matrix

```
void Print_local_matrices(char* title, LOCAL_MATRIX_T* local_A, GRID_INFO_T* grid)
{
    int coords[2]; int i, j;
    int source; MPI_Status status;

    if (grid->my_rank == 0) {
        printf("%s\n", title);
        printf("Process %d > grid_row = %d, grid_col = %d\n", grid->my_rank, grid->my_row, grid->my_col);
        for (i = 0; i < Order(local_A); i++) {
            for (j = 0; j < Order(local_A); j++)
                printf("%4.1f ", Entry(local_A,i,j));
            printf("\n");
        }
        for (source = 1; source < grid->p; source++) {
            MPI_Recv(temp_mat, 1, local_matrix_mpi_t, source, 0, grid->comm, &status);
            MPI_Cart_coords(grid->comm, source, 2, coords);
            printf("Process %d > grid_row = %d, grid_col = %d\n",
                source, coords[0], coords[1]);
            for (i = 0; i < Order(temp_mat); i++) {
                for (j = 0; j < Order(temp_mat); j++)
                    printf("%4.1f ", Entry(temp_mat,i,j));
                printf("\n");
            }
        }
        fflush(stdout);
    } else
        MPI_Send(local_A, 1, local_matrix_mpi_t, 0, 0, grid->com
} /* Print_local_matrices */
```