Content

- Introduction to MPI
- Point-Point Communication
  - Send and receive
- Writing your first MPI program
- Using the common MPI functions
- MPI collective operations

Textbook:
An Introduction to Parallel Programming
Author: Peter Pacheco
http://www.cs.usfca.edu/~peter/ipp/
Tree-structured communication

1) In the first phase:
   a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   b) Processes 0, 2, 4, and 6 add in the received values.
   c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   d) Processes 0 and 4 add the received values into their new values.

2) Process 4 sends its newest value to process 0. Process 0 adds the received value to its newest value.

A tree-structured global sum
An alternative tree-structured global sum

Processes

0  1  2  3  4  5  6  7
  5  2  1  3  4  5  7  2
  11 7 5 8 1 7
  3 6 3
  9

MPI_Reduce

```c
int MPI_Reduce(
    void * input_data_p /* in */,
    void * output_data_p /* out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator /* in */,
    int dest_process /* in */,
    MPI_Comm comm /* in */);
```

MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
Predefined reduction operators in MPI

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.
- The arguments passed by each process to an MPI collective communication must be “compatible.”
- For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.
Collective vs. Point-to-Point Communications

- The `output_data_p` argument is only used on `dest_process`.
- However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it’s just `NULL`.

Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications don’t use tags.
- They’re matched solely on the basis of the communicator and the order in which they’re called.

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(a = 1; c = 2)</td>
<td>(a = 1; c = 2)</td>
<td>(a = 1; c = 2)</td>
</tr>
<tr>
<td>1</td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
</tr>
</tbody>
</table>
Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0.

At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6.

However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`.

The order of the calls will determine the matching so the value stored in `b` will be \(1+2+1 = 4\), and the value stored in `d` will be \(2+1+2 = 5\).

**MPI_Allreduce**

Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```c
int MPI_Allreduce(
    void* input_data_p  /* in */,
    void* output_data_p /* out */,
    int count            /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Op operator      /* in */,
    MPI_Comm comm        /* in */);
```
A global sum followed by distribution of the result.

A butterfly-structured global sum.
Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```c
int MPI_Bcast(
    void* data_p /* in/out */,
    int count /* in */,
    MPI_Datatype datatype /* in */,
    int source_proc /* in */,
    MPI_Comm comm /* in */);
```

A tree-structured broadcast.
Get_input that uses MPI_Bcast

```c
void Get_input(
    int    my_rank  /* in */,
    int    comm_sz  /* in */,
    double* a_p     /* out */,
    double* b_p     /* out */,
    int*   n_p      /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_input */
```

Data distributions

\[
x + y = (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1})
\]
\[
= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1})
\]
\[
= (z_0, z_1, \ldots, z_{n-1})
\]
\[
= Z
\]

Compute a vector sum.
Serial implementation of vector addition

```c
void Vector_sum(double x[], double y[], double z[], int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
} /* Vector_sum */
```

Different partitions of a 12-component vector among 3 processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Cyclic</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
<td>0 3 6 9</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6 7</td>
<td>1 4 7 10</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10 11</td>
<td>2 5 8 11</td>
<td>4 5 10 11</td>
</tr>
</tbody>
</table>
Partitioning options

- **Block partitioning**
  - Assign blocks of consecutive components to each process.
- **Cyclic partitioning**
  - Assign components in a round robin fashion.
- **Block-cyclic partitioning**
  - Use a cyclic distribution of blocks of components.

Parallel implementation of vector addition

```c
void Parallel_vector_sum(
    double local_x[] /* in */,
    double local_y[] /* in */,
    double local_z[] /* out */,
    int local_n /* in */
) {
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++)
        local_z[local_i] = local_x[local_i] + local_y[local_i];
} /* Parallel_vector_sum */
```
MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```c
int MPI_Scatter(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int src_proc /* in */,
    MPI_Comm comm /* in */);
```

### Reading and distributing a vector

```c
void Read_vector(
    double* a = NULL,
    int n,
    char vec_name[] /* in */
) { 
    int i;

    if (my_rank == 0) {
        a = malloc(n * sizeof(double));
        printf("Enter the vector %s\n", vec_name);
        for (i = 0; i < n; i++)
            scanf("%lf", &a[i]);
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
                    0, comm);
        free(a);
    } else {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
                    0, comm);
    } /* Read_vector */
```
Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```c
int MPI_Gather(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int dest_proc /* in */,
    MPI_Comm comm /* in */);
```

Print a distributed vector (1)

```c
void Print_vector(
    double local_b[] /* in */,
    int local_n /* in */,
    int n /* in */,
    char title[] /* in */,
    int my_rank /* in */,
    MPI_Comm comm /* in */) {

    double* b = NULL;
    int i;
```
Print a distributed vector (2)

```c
if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    printf("%s
", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
}
/* Print_vector */
```

Allgather

- Concatenates the contents of each process’ `send_buf_p` and stores this in each process’ `recv_buf_p`.
- As usual, `recv_count` is the amount of data being received from each process.

```c
int MPI_Allgather(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    MPI_Comm comm /* in */);
```
Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$x$ is a vector with $n$ components

$y = Ax$ is a vector with $m$ components

$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1}$

$i$-th component of $y$

Dot product of the $i$th row of $A$ with $x$. 

\[
\begin{array}{c|c|c|c}
 a_{00} & a_{01} & \cdots & a_{0,n-1} \\
 a_{10} & a_{11} & \cdots & a_{1,n-1} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
 x_0 & \cdots & x_{n-1} \\
 y_0 & \cdots & y_{m-1} \\
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Multiply a matrix by a vector

```c
/* For each row of A */
for (i = 0; i < m; i++) {
    /* Form dot product of ith row with x */
    v[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Serial pseudo-code

C style arrays

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
\end{pmatrix}
\]

stored as

\[
0 1 2 3 4 5 6 7 8 9 10 11
\]
Serial matrix-vector multiplication

```c
void Mat_vect_mult(
    double A[], /* in */
    double x[], /* in */
    double y[], /* out */
    int m /* in */,
    int n /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```c
void Mat_vect_mult(
    double local_A[] /* in */,
    double local_x[] /* in */,
    double local_y[] /* out */,
    int local_m /* in */,
    int n /* in */,
    int local_n /* in */,
    MPI_Comm comm /* in */) {
    double* x;
    int local_i, j;
    int local_ok = 1;
```
An MPI matrix-vector multiplication function (2)

```c
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
          x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);
/* Mat_vec_mult */
```

Summary

- Collective communication involves all the processes in a communicator
- It usually involves more than two processes
- Many parallel programs use the single-program multiple data or SPMD approach.
- MPI collective communication functions
  - Broadcast
  - Reduce
  - Gather and Scatter
  - Allreduce and Allgather
- None of the collective communication use tags