Distributed Memory Programming With MPI (2)

2014 Spring

Jinkyu Jeong
(jinkyu@skku.edu)
Roadmap

- Hello World in MPI program
- Basic APIs of MPI
- Example program
  - The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.
COLLECTIVE COMMUNICATION
Tasks and communications for Trapezoidal Rule

- Compute area of trap 0
- Compute area of trap 1
- ... Compute area of trap $n - 1$
- Add areas

Similar to global sum problem
Global Sum

- Serial

- Tree-structured

• Complexity of process 0
  • n communications vs. log(n) communications
Tree-structured global sum

- Which one is faster?
  - Same
  - Or depending on the communication speed between processes
## Collective Communications

- **Coordinated communication among a group of processes**
  - Group is specified by communicator
  - All collective operations are blocking and no message tags
  - All processes in the communicator group call the collective operation

- **Abstraction of various types of communications**
  - ex. Broadcast, gather, ...

- **Hides details of communications**
  - Developer of MPI has better understanding of communications between processes
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);
```
Collective Computation Operations

- Allreduce, Reduce, Scan, ... take combination functions
- Built-in collective computation operations
  - MPI_MAX, MPI_MIN  \(\text{// max, min}\)
  - MPI_MAXLOC, MPI_MINLOC  \(\text{// max, min with location}\)
  - MPI_PROD, MPI_SUM  \(\text{// product, sum}\)
  - MPI_LAND, MPI_LOR, MPI_LXOR  \(\text{// logical and/or/xor}\)
  - MPI_BAND, MPI_BOR, MPI_BXOR  \(\text{// bitwise and/or/xor}\)
- User-defined collective computation operations
  - MPI_Op_create(user_func, commute_flag, user_op);
  - MPI_Op_free(user_op);

```c
user_func (invec, inoutvec, len, datatype)
{
    for (i=0; i<len; i++) inoutvec[i] = invec[i] op inoutvec[i];
}
```
Collective Communications (1)

- **All** processes must call the same collective function.
  - Ex. `MPI_Recv()` in P0 + `MPI_Reduce()` in P1 (X)
- The arguments must be compatible to each other
  - Ex.

```c
MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);
```

**Mismatch!!**

```c
MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 1, MPI_COMM_WORLD);
```
Collective Communications (2)

- Output argument is only used in the destination process
  - But, other processes should provide destination argument, even if it is NULL
  - Ex.

  **Process 0**
  ```c
  MPI_Reduce(in_buf, out_buf, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);
  ```

  **Process 1**
  ```c
  MPI_Reduce(in_buf, NULL, 1, MPI_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);
  ```
Collective vs. Point-to-Point Comm.

- **Point-to-point communications**
  - MPI_Send/Recv are matched on the basis of tags and ranks

- **Collective communications**
  - Do NOT use tags
  - They’re matched solely on the basis of receiver’s rank and order
Example

<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><strong>MPI_Reduce(&amp;a, &amp;b, ...)</strong></td>
<td><strong>MPI_Reduce(&amp;c, &amp;d, ...)</strong></td>
<td><strong>MPI_Reduce(&amp;a, &amp;b, ...)</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>MPI_Reduce(&amp;c, &amp;d, ...)</strong></td>
<td><strong>MPI_Reduce(&amp;a, &amp;b, ...)</strong></td>
<td><strong>MPI_Reduce(&amp;c, &amp;d, ...)</strong></td>
</tr>
</tbody>
</table>

Multiple calls to MPI_Reduce to Process 0

- **Expected result**
  - \( b = 1 + 1 + 1 = 3 \)
  - \( d = 2 + 2 + 2 = 6 \)

- **The memory locations are irrelevant to the matching of the calls to MPI_Reduce**

- **The order of the calls will determine the matching**
  - \( b = 1+2+1 = 4 \)
  - \( d = 2+1+2 = 5. \)
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 */);
```
Communication Example of Allreduce

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 using 3 MPI_Send/Recv Calls

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

    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
} /* Get_input */
```
Get_input using 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\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

- Vector sum

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

- Partition vector and distribute each piece of vector to each process

- Example:

\begin{align*}
(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) \\
(y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7) \\
(z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)
\end{align*}

- Process 0
- Process 1
- Process 2
- Process 3
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.

<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>

Different partitions of a 12-component vector among 3 processes.
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 */
```

- How to distribute each piece of vector to each process
  - MPI_Bcast?
    - Not good; each process does not need entire vector
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 */);
```
void Read_vector(
    double local_a[] /* out */,
    int local_n /* in */,
    int n /* in */,
    char vec_name[] /* in */,
    int my_rank /* in */,
    MPI_Comm comm /* in */) {

    double* a = NULL;
    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
);```

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\n", 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

- P0 calculates $y_0$, P1 calculates $y_1$, P2 calculates $y_3$, ...
- Finally, every processes call `MPI_Gather` to have the result vector $y$
- If the result vector $y$ is used for next multiplication?
  - $Ax = y$ then, $By = z$
  - The result vector should be distributed to all processes
    - `MPI_Gather` & `MPI_Bcast` ??
  - We can use `MPI_Allgather`
An MPI matrix-vector multiplication function

```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_vect_mult */
```
Broadcast and Reduce

- **Basic collective operations**
  - MPI_Bcast(buf, count, datatype, root, comm)
    - Send data from one process (root) to all others
  - MPI_Reduce(buf, result, count, datatype, operation, root, comm)
    - Combine data from all processes, using a specified operation
    - Return the result to a single process (root)

<table>
<thead>
<tr>
<th>rank</th>
<th>send buffer</th>
<th>send buffer</th>
<th>rank</th>
<th>send buffer</th>
<th>result buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>A</td>
<td>0</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>A</td>
<td>1</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>A</td>
<td>2</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>A</td>
<td>3</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Bcast (root=0) → Reduce (root=0)

\[ X = A \circ B \circ C \circ D \]
Scatter and Gather

- **MPI_Scatter**
  - Distribute the data from the root to all processes

- **MPI_Gather**
  - Collect the data from all processes to the root
Other Collective Communications

- **MPI_Allreduce**
  - **rank**: 0, send buffer: A
  - **rank**: 1, send buffer: B
  - **rank**: 2, send buffer: C
  - **rank**: 3, send buffer: D
  - **result buffer**: X
  - **Expression**: \( X = A \circ B \circ C \circ D \)

- **MPI_Allgather**
  - **rank**: 0, send buffer: A
  - **rank**: 1, send buffer: B
  - **rank**: 2, send buffer: C
  - **rank**: 3, send buffer: D
  - **recv buffer**: ABCD

- **MPI_Alltoall**
  - **rank**: 0, send buffer: \( A_0B_0C_0D_0 \)
  - **rank**: 1, send buffer: \( A_1B_1C_1D_1 \)
  - **rank**: 2, send buffer: \( A_2B_2C_2D_2 \)
  - **rank**: 3, send buffer: \( A_3B_3C_3D_3 \)
  - **recv buffer**: \( A_0A_1A_2A_3 \)

- **MPI_Scan**
  - **rank**: 0, send buffer: A
  - **rank**: 1, send buffer: B
  - **rank**: 2, send buffer: C
  - **rank**: 3, send buffer: D
  - **recv buffer**: W
  - **Expression**: \( W = A \)
  - **Expression**: \( X = A \circ B \)
  - **Expression**: \( Y = A \circ B \circ C \)
  - **Expression**: \( Z = A \circ B \circ C \circ D \)