Lecture 9: MPI - III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adopted and slightly modified from:
• Gerassimos Barlas
• Peter S. Pacheco
Collective

point-to-point
Data distributions

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

```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 */
```

Sequential version
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>

- **Block**: Assign blocks of consecutive components to each process.
- **Cyclic**: Assign components in a round robin fashion.
- **Block-cyclic**: 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 */
```

How will you distribute parts of x[] and y[] to processes?
Scatter

- Read an entire vector on process 0
- **MPI_Scatter** 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 */);
```

Important:
- All arguments are important for the source process (process 0 in our example)
- For all other processes, only `recv_buf_p`, `recv_count`, `recv_type`, `src_proc`, and `comm` are important
Reading and distributing a vector

```c
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 */
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 */);

- **send_buf_p**
  - is not used except by the sender.
  - However, it must be defined or NULL on others to make the code correct.
  - Must have at least communicator size * send_count elements
- All processes must call MPI_Scatter, not only the sender.
- **send_count** the number of data items sent to each process.
- **recv_buf_p** must have at least send_count elements
- **MPI_Scatter** uses block distribution
Process 0: 0 1 2 3 4 5 6 7 0

Process 0: 0 1 2

Process 1: 3 4 5

Process 2: 6 7 8

```
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 */);
```
Gather

- **MPI_Gather** collects all of the components of the vector onto process dest process, ordered in rank order.

```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 */);
```

**Important:**
- All arguments are important for the destination process.
- For all other processes, only `send_buf_p`, `send_count`, `send_type`, `dest_proc`, and `comm` are important.
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\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

\[ A = (a_{ij}) \text{ is an } m \times n \text{ matrix} \]

\[ \mathbf{x} \text{ is a vector with } n \text{ components} \]

\[ \mathbf{y} = A\mathbf{x} \text{ is a vector with } m \text{ components} \]

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

- \text{i-th component of } \mathbf{y} \n- \text{Dot product of the ith row of } A \text{ with } \mathbf{x}.\]
Matrix-vector multiplication

\[
\begin{array}{cccc}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{array}
\times
\begin{array}{c}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{array}
=
\begin{array}{c}
y_0 \\
y_1 \\
\vdots \\
y_{m-1}
\end{array}
\]

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

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

Pseudo-code Serial Version
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_vec_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_vec_mult */
}
```

Let's assume x[] is distributed among the different processes
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_vect_mult */
```
Keep in mind ...

- In distributed memory systems, communication is more expensive than computation.
- Distributing a fixed amount of data among several messages is more expensive than sending a single big message.
Derived datatypes

• Used to represent any collection of data items

• If a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.

• A function that receives data can distribute the items into their correct destinations in memory when they’re received.
Derived datatypes

- A sequence of basic MPI data types together with a displacement for each of the data types.

Address in memory where the variables are stored

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{ (\text{MPI\_DOUBLE}, 0), (\text{MPI\_DOUBLE}, 16), (\text{MPI\_INT}, 24) \}

(We assume we start with a.)
MPI_Type create_struct

• Builds a derived datatype that consists of individual elements that have different basic types.

```c
int MPI_Type_create_struct(
    int count, /* in */,
    int array_of_blocklengths[], /* in */,
    MPI_Aint array_of_displacements[], /* in */,
    MPI_Datatype array_of_types[], /* in */,
    MPI_Datatype* new_type_p, /* out */);
```

an integer type that is big enough to store an address on the system.

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```
Before you start using your new data type

```
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.
When you are finished with your new type

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```

This frees any additional storage used.
```c
void Build_mpi_type(
    double* a_p,  /* in */,
    double* b_p,  /* in */,
    int* n_p,     /* in */,
    MPI_Datatype* input_mpi_t_p, /* out */)
{

    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
```
MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Type_create_struct(3, array_of_blocklengths,
    array_of_displacements, array_of_types,
    input_mpi_t_p);
MPI_Type_commit(input_mpi_t_p);

} /* Build_mpi_type */
The receiving end can use the received complex data item as if it is a structure.
MEASURING TIME IN MPI
We have seen in the past ...

- `time` in Linux
- `clock()` inside your code
- Does MPI offer anything else?
Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```c
double MPI_Wtime(void);

double start, finish;
...
start = MPI_Wtime();
/* Code to be timed */
...
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n", my_rank, finish-start);
```
How to Sync Processes?
MPI_Barrier

• Ensures that no process will return from calling it until every process in the communicator has started calling it.

```c
int MPI_Barrier(MPI_Comm comm /* in */);
```
Let's see how we can analyze the performance of an MPI program.

The matrix-vector multiplication:

define double local_start, local_finish, local_elapsed, elapsed;

MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
            MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
Run-times of serial and parallel matrix-vector multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.1</td>
<td>16.0</td>
<td>64.0</td>
<td>270</td>
<td>1100</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>8.5</td>
<td>33.0</td>
<td>140</td>
<td>560</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
<td>5.1</td>
<td>18.0</td>
<td>70</td>
<td>280</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
<td>3.3</td>
<td>9.8</td>
<td>36</td>
<td>140</td>
</tr>
<tr>
<td>16</td>
<td>1.7</td>
<td>2.6</td>
<td>5.9</td>
<td>19</td>
<td>71</td>
</tr>
</tbody>
</table>

(Seconds)
# Speedups of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_size</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.1</td>
<td>3.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>4.8</td>
<td>6.5</td>
<td>7.5</td>
<td>7.9</td>
</tr>
<tr>
<td>16</td>
<td>2.4</td>
<td>6.2</td>
<td>10.8</td>
<td>14.2</td>
<td>15.5</td>
</tr>
</tbody>
</table>
## Efficiencies of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th></th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.94</td>
<td>0.97</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
<td>0.78</td>
<td>0.89</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
<td>0.61</td>
<td>0.82</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>16</td>
<td>0.15</td>
<td>0.39</td>
<td>0.68</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Conclusions

• Reducing messages is a good performance strategy!
  – Collective vs point-to-point

• Distributing a fixed amount of data among several messages is more expensive than sending a single big message.