Distributed Memory Programming With MPI (3)

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.
MPI DERIVED DATATYPES
Limitations of MPI Data Types

- Only primitive data types can be exchanged through `MPI_Send/Recv`

- Many programs use more complex data structures
  - Ex. `struct` in C

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

Basic Data Types in MPI
MPI Derived Data Types

- To make more complex data types to be exchanged through MPI communication methods
  - MPI should know the size of a data structure
  - MPI should know the members within the data structure
    - Location
    - Size of each member

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```

<table>
<thead>
<tr>
<th>Member</th>
<th>Offset in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>16</td>
</tr>
<tr>
<td>value</td>
<td>32</td>
</tr>
</tbody>
</table>
MPI_Type create_struct

- Builds a derived datatype consisting of individual elements

```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 */);
```

- **array_of_block_lengths**
  - Each member can be either a variable or an array
  - Ex. {2, 2, 2};

- **array_of_displacements**
  - Offsets of each member from start address
  - Ex. {0, 16, 32}

- **array_of_types**
  - Types of each member
  - Ex. = {MPI_DOUBLE, MPI_DOUBLE, MPI_LONG}

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};
```
**MPI_Get_address**

- To know the address of the memory location referenced by `location_p`.
- The address is stored in an integer variable of type `MPI_Aint`.

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```

```c
struct a {
    MPI_DOUBLE x[2];
    MPI_DOUBLE y[2];
    MPI_LONG value[2];
};

struct a a;
MPI_Get_address(&a.x, &x_addr);
MPI_Get_address(&a.y, &y_addr);
MPI_Get_address(&a.value, &value_addr);
array_of_displacements[0] = x_addr - &a;
array_of_displacements[1] = y_addr - &a;
```
Other methods

- **MPI_Type_commit**
  - To let MPI know the new data type
  - After calling this function, the new data type can be used in MPI communication methods

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

- **MPI_Type_free**
  - When the new data type is no longer used, this function frees any additional storages used for the new data type

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

- An alternative method to send/receive a complex data structure
- Pack multiple data types into a single buffer
- One pair of MPI_Send & MPI_Recv
- Sender and receiver have to know which data types are packed in the single buffer

```c
buffer = malloc()
MPI_Pack(A)
MPI_Pack(B)
MPI_Pack(C)
MPI_Send(buffer, MPI_PACKED)

buffer = malloc()
MPI_Recv(buffer, MPI_PACKED)
MPI_Unpack(A)
MPI_Unpack(B)
MPI_Unpack(C)
```

Sender  |  Pack  |  Buffer  |  Unpack  |  Receiver
---|---|---|---|---
A  |  B  |  C  |  A  |  B  |  C
MPI_Pack/Unpack – con’t

- **MPI_Pack**

```c
int MPI_Pack(
    void* in_buf, /* in */,
    int in_buf_count, /* in */,
    MPI_Datatype datatype, /* in */,
    void* pack_buf, /* out */,
    int pack_buf_sz, /* in */,
    int* position_p /* in/out */
);

MPI_Comm comm /* in */;
```

- **MPI_Unpack**

```c
int MPI_Unpack(
    void* pack_buf, /* in */,
    int pack_buf_sz, /* in */,
    int* position_p /* in/out */,
    void* out_buf, /* out */,
    int out_buf_count, /* in */,
    MPI_Datatype datatype, /* in */
    MPI_Comm comm /* in */;
```
MANIPULATING COMMUNICATORS
Communicators

- All MPI communication is based on a communicator which contains a context and a group
  - Contexts define a safe communication space for message-passing – viewed as system-managed tags
  - Contexts allow different libraries to co-exist
  - Group is just a set of processes
  - Processes are always referred to by the unique rank in a group

- Pre-defined communicators
  - MPI_COMM_WORLD
  - MPI_COMM_NULL // initial value, cannot be used as for comm
  - MPI_COMM_SELF // contains only the local process
Communicator Manipulation

- **Duplicate communicator**
  - `MPI_Comm_dup(comm, newcomm)`
  - Create a new context with similar structure

- **Partition the group into disjoint subgroups**
  - `MPI_Comm_split(comm, color, key, newcomm)`
  - Each sub-communicator contains the processes with the same color
  - The rank in the sub-communicator is defined by the `key`

```c
color = (rank % 2 == 0)? 0 : 1;
key  = rank / 2;
MPI_Comm_split(comm, color, key, &newcomm);
```
Communicator Manipulation – con’t

- Obtain an existing group and free a group
  - MPI_Comm_group(comm, group) – create a group having processes in the specified communicator
  - MPI_Group_free(group) – free a group

- New group can be created by specifying members
  - MPI_Group_incl(), MPI_Group_excl()
  - MPI_Group_range_incl(), MPI_Group_range_excl()
  - MPI_Group_union(), MPI_Group_intersect()
  - MPI_Group_compare(), MPI_Group_translate_ranks()

- Subdivide a communicator
  - MPI_Comm_create(comm, group, newcomm)
PERFORMANCE EVALUATION
**Elapsed parallel time**

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

```c
double MPI_Wtime(void);
```

```c
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);
```
Elapsed serial time

- Returns time in microseconds elapsed from some point in the past

```c
#include "timer.h"
.
.
**double** start, finish;
.
.
GET_TIME(start);
/* Code to be timed */
.
.
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);

#define GET_TIME(now) {
  struct timeval t; \n  gettimeofday(&t, NULL); \n  now = t.tv_sec +
  t.tv_usec/1000000.0; \n}
```
MPI_Barrier

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

```c
int MPI_Barrier(MPI_Comm comm /* in */);
```
Measuring Time using MPI_BARRIER

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);