MPI Derived Data Types and Collective Communication
Why Derived Data Types?

• Elements in an MPI message are of the same type.
• Complex data, requires two separate messages.

• Bad example:
  ```c
  typedef struct {
    float a,b;
    int n;
  } Mine;
  MPI_Send(&data, 1, Mine, 0, 99, MPI_COMM_WORLD);
  ```

  **Compilation error:**
  A type can’t be passed as a parameter

• Reminder:
  • `int MPI_Send(void *sndbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
Why Derived Data Types?

• Elements in an MPI message are of the same type.
• Complex data, requires two separate messages.

• Bad example:
  ```c
  typedef struct {
    float a,b;
    int n;
  } Mine;
  
  MPI_Send(&data, sizeof(Mine), MPI_BYTE, 0, 99, MPI_COMM_WORLD);
  ```

  Will fail when communicating between different kind of machines

• Reminder:
  ```c
  int MPI_Send(void *sndbuf, int count, MPI_Datatype datatype, int dest, int tag, 
               MPI_Comm comm)
  ```
Defining Derived Data Types

// Build a derived datatype for two floats and an int
void build_derived_type(Mine* indata,
    MPI_Datatype* message_type_ptr) {
    int block_lengths[3];
    MPI_Aint displacements[3];
    MPI_Datatype typelist[3];
    MPI_Aint addresses[4]; // Helper array

    // First specify the types
    typelist[0] = MPI_FLOAT;
    typelist[1] = MPI_FLOAT;
    typelist[2] = MPI_INT;

    // Specify the number of elements of each type
    block_lengths[0] = block_lengths[1] =
        block_lengths[2] = 1;

    // Calculate the displacements of the members relative to indata
    MPI_Address(indata, &addresses[0]);
    MPI_Address(&(indata->a), &addresses[1]);
    MPI_Address(&(indata->b), &addresses[2]);
    MPI_Address(&(indata->n), &addresses[3]);

    displacements[0] = addresses[1] - addresses[0];
    displacements[1] = addresses[2] - addresses[0];

    // Create the derived type
    MPI_Type_struct(3, block_lengths,
        displacements, typelist, message_type_ptr);

    // Commit it so that it can be used
    MPI_Type_commit(message_type_ptr);
}
Defining Derived Data Types

Mine data1, data2, data3[10];
MPI_Datatype my_type;
Build_derived_type(&data1, &my_type);

// Using the new data type
MPI_Send (&data1, 1, my_type, 0, 98, MPI_COMM_WORLD);
MPI_Ssend (&data2, 1, my_type, 0, 99, MPI_COMM_WORLD);
MPI_Isend (data3, 10, my_type, 0, 97, MPI_COMM_WORLD);
MPI Collective Communication

• Reuse code for patterns that appear in different types of applications
• Built by using point-to-point communication routines
• Collective communication routines in MPI:
  • Barrier synchronization
  • Broadcast from one member to all other members
  • Scatter data from one member to all other members
  • Gather data from all members to one member
  • All-to-all exchange of data
  • Global reduction (e.g., sum, min of "common" data element to one node)
  • Scan across all members of a group
• Synchronization, data movement, and global computation
Collective communication routines characteristics

- Coordinated communication within a group of processes
  - Identified by an MPI communicator
  - No message tags are needed
- Substitute for a more complex sequence of point-to-point calls
- All routines block until they are locally completed
  - Starting from MPI 1.7, non-blocking collective communication is supported
- In some cases, a root process originates or receives all data
- Amount of data sent must exactly match amount of data specified by receiver
Barrier synchronization routine

• A synchronization primitive
  • Blocking until all the nodes in the group have called it

• `MPI_Barrier(MPI_Comm comm)`
  • comm: a communicator
Data movement routines - Broadcast

- One processor sends data to all the processes in a group

- `int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  - Must be called by each node in the group, specifying the same `comm`, `root` and `count`

```c
MPI_Comm comm; int array[100]; int root=1;
MPI_Bcast( array, 100, MPI_INT, root, comm);
```
Gather and Scatter

- `int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)`
- `int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)`

- `scount` and `rcount` specifies the amount of elements to send/recv to/from each process (not the total)
**MPI_Scatter**

- Scatter sets of 100 ints from the root to each process in the group
- At all processes
  - `MPI_Scatter(NULL, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm)`
- At root = 0
  - `MPI_Scatter(sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm)`
Gatherv and Scatterv

• Allow each node to send/receive a different number of elements

• int MPI_Gatherv(void* sbuf, int scount, MPI_Datatype stype,
  void* rbuf, int *rcount, int* displs, MPI_Datatype rtype, int root,
  MPI_Comm comm)

• int MPI_Scatterv(void* sbuf, int* scount, int* displa,
  MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype,
  int root, MPI_Comm comm)
MPI_Gatherv

• At all processes: `scount`=100 or 150
  • `MPI_Gatherv(sbuf, scount, MPI_INT, NULL, NULL, NULL, MPI_INT, root, comm);`
• At root = 0
  • `rcount[3] = {50,100,150}`
  • `MPI_Gatherv(sbuf, 50, MPI_INT, rbuf, rcount, displs, MPI_INT, root, comm);`

![Diagram showing MPI_Gatherv operation]

Displacement differences must be larger than the corresponding `rcounts` entry.
Allgather

- int MPI_Allgather(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
- int MPI_Allgatherv(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int* rcount, int* displs, MPI_Datatype rtype, MPI_Comm comm)
All to All

- `int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype, void* rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)`

![Diagram showing data distribution for All to All communication](image)

1. Processes 1 to 4 distribute data to each other.
2. Each process receives data from all other processes.

Data distribution:
- `sbuf`: 1A 1B 1C 1D
- `rbuf`: 4A 4B 4C 4D

Processes 1 to 4 exchange data as follows:
- Process 1 sends data to Process 2, 3, and 4.
- Process 4 receives data from Process 3 and Process 1, and sends data to Process 1.
Global computation routines Reduce

- $D(i,j)$ is the $j$th data item in process $i$
- In the root process: $D_j = D(0,j) \odot D(1,j) \odot ... \odot D(n-1,j)$
- Operator must be associative: $(a \odot b) \odot c = a \odot (b \odot c)$
- MPI predefined operators are also commutative: $a \odot b = b \odot a$

- int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm)

- int MPI_Allreduce(void* sbuf, void* rbuf, int count, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)

- int MPI_Reduce_scatter (void* sbuf, void* rbuf, int* rcounts, MPI_Datatype stype, MPI_Op op, MPI_Comm comm)
  - rcounts indicate the number of elements in result distributed to each process (vector)
  - Array must be identical on all calling processes
Scan

• $D(k, j)$ - the $j^{th}$ data item in process $k$ after returning from scan
• $d(k, j)$ - the $j^{th}$ data item in process $k$ before the scan
• For processor $k$:
  • $D(k, j) = d(0, j) \odot d(1, j) \odot \cdots \odot d(k, j)$

• int MPI_Scan(void* sbuf, void* rbuf, int count, 
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
## Predefined reduce operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td>integer, float</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bit-wise or</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bit-wise xor</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
<td>combination of int, float, double, and long double</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
<td>combination of int, float, double, and long double</td>
</tr>
</tbody>
</table>
User-defined operations

• **typedef void MPI_User_function( void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);**
  • $\text{inoutvec}[i] \leftarrow \text{invec}[i] \otimes \text{inoutvec}[i], i \in [0 \ldots \text{len} - 1]$
  • No MPI communication function may be called inside the user function
  • MPI_ABORT may be called inside the function in case of an error

• **MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)**
  • Should be called on all processors

• User-defined operation is deallocated using:
  • **int MPI_op_free( MPI_Op *op)**
Performance issues

• A great deal of hidden communication takes place with collective communication

• Performance depends greatly on the particular implementation of MPI

• Because there may be forced synchronization, not always best to use collective communication