The MPI Message-passing Standard
Practical use and implementation (VI)

SPD Course
21/03/2019
Massimo Coppola

SCUOLA SUPERIORE SANT'ANNA PISA
UNIVERSITÀ DI PISA
ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"
HPC laboratory

Datatypes

REFINING DERIVED DATATYPES LAYOUT FOR COMPOSITION

Derived D.type Extent and composition

- For complex derived datatypes, extent plays an important role
  - Plain definition : distance between the first (smallest address) byte and last (largest address) byte used in memory
  - Actual use : the offset between two items of the given datatype when they are stored consecutively in memory
    - E.g. whenever a contiguous datatype is created or a communication buffer with more instances is used
- Setting extent manually (MPI1, MPI2>)
- Querying extent
- Examples with derived datatypes

Get extent

```
Int MPI_Type_get_extent_x (MPI_Datatype datatype,
      MPI_Count *lb, MPI_Count *extent)
```

- Get the lower bound and extent of a datatype
  - By default, lower bound = lowest-address location of a datatype
  - Extent = distance from lower bound to highest address location used by the datatype

Modify extent

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
      MPI_Aint lb, MPI_Aint extent,
      MPI_Datatype *newtype)
```

- Modify the lower bound and extent of a datatype
- Reset lower bound and extent of the datatype to new arbitrary values, for the sake of data structures composition

Retrieve original extent

```
int MPI_Type_get_true_extent(MPI_Datatype datatype,
      MPI_Aint *true_lb, MPI_Aint *true_extent)
```

- Retrieve the true lower bound and extent values from a datatype
- MPI always keeps the information as it is needed for the actual packing and unpacking operations

Intracommunicators

COLLECTIVE PRIMITIVES WITH COMMUNICATION AND COMPUTATION

Reduce

- ```
  int MPI_Reduce(
        const void* sendbuf, void* recvbuf, int count,
        MPI_Datatype datatype,
        MPI_Op op, int root, MPI_Comm comm)
  ```

- reduce operation across all processes of a Communicator
  - Reduces the elements in the same position of each process' buffer, leaving results in root's buffer
- count, datatype, op, root, comm arguments must match
  - If count == 1 we have a classical element-wise reduction
  - If count>1 we have several reductions at the same time
- As with any collective, the communication pattern is implementation dependent (but is op commutative ?)
- MPI provides most basic operators
  - Operators are associative
  - Operators may be commutative → potential optimizations
  - Note that: floating point op.s may be seen as non-commutative
  - *Datatype* must be compatible with *op*
  ```

MPI Scan/Reduce operators

- Arithmetic operations
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
    - Generally allowed on MPI integral and floating point types (including complex)
- Logic (L) and bit wise (B) operations
  - MPI_LAND, MPI_LOR, MPI_LXOR
    - Generally allowed on C integers and on logical types
  - MPI_BAND, MPI_BOR, MPI_BXOR
    - Generally allowed on C/Fortran integers

MPI_MINLOC and MAXLOC

- Operators defined on couples (value, index)
  - MPI_MAXLOC, MPI_MINLOC
  - Value is any integral or floating point type
  - Index is an integral type
  - chars used as integers require special attention
    - e.g. explicitly using MPI_SIGNED _CHAR / UNSIGNED_CHAR
- MINLOC : compute the global minimum of v and the index attached to it
- MAXLOC : compute the global maximum of v and the index attached to it
- Lexicographic order
  - when more values hit the minimum (maximum) the lower one is chosen
- Example application
  - pass (value, rank) to detect the rank of the process with the minimum/maximum value

MPI couple types

- These couple types are supported by the MPI_MINLOC and MPI_MAXLOC operators

- MPI_FLOAT_INT - struct { float, int }
- MPI_LONG_INT - struct { long, int }
- MPI_DOUBLE_INT - struct { double, int }
- MPI_SHORT_INT - struct { short, int }
- MPI_2INT - struct { int, int }
- MPI_LONG_DOUBLE_INT
  - struct { long double, int }
  - this is an OPTIONAL type

Operators semantics

- Operators are called within the reduction collective by the instances of the MPI library of the processes of the program
- Each operators receives two local buffers and performs a reduction step on their contents
  - The buffers are possibly allocated by the library implementation as temporaries
  - Many operators are polymorphic, so they have to detect the type of data in the buffer
  - Datatype is a parameter passed from the collective down to the operator, but remember it is a handle
    - Easy case: MPI basic datatypes are globally known to MPI runtime and to the program
    - Besides, MPI standard operators are easy

MPI operators and computing-collective primitives

- MPI operators (including user-defined ones) are used by all MPI collectives performing distributed computation
  – MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN
  – All the non blocking version of those collectives (since MPI-3)
  – MPI_REDUCE_LOCAL (special case actually designed for MPI implementers)

User-defined operators

- MPI allows you to define your own operators
  - They can apply to basic and user-defined datatypes
- What do you need to do
  - (Possibly) provide relevant datatype definitions
  - Provide MPI with a definition of the operator
    - a compiled function with a specific signature
    - the operator definition this is local to each process
  - Detect and recognize the MPI_Datatype within the operator code
    - To detect errors
    - If the operator needs to be polymorphic
  - Combine each couple of elements in the same position of the two input buffers
- Operator code can call **no** MPI communication primitives; only MPI_ABORT() in case of error

MPI_Op_create

- ```
  int MPI_Op_create(MPI_User_function* user_fn,
        int commute, MPI_Op* op)
  ```

- MPI primitive for defining operators
- Takes a user function pointer as first argument
- Can specify non-commuting operators
- Returns the operator handle

- MPI_Op_free allows to free operators
  ```

Operator signature

- ```
  typedef void MPI_User_function(
          void* invec,
          void* inoutvec,
          int *len,
          MPI_Datatype *datatype);
  ```
- row-wise combines data from two buffers
  - results are placed in the second buffer
- The datatype handle comes from the collective call (e.g. reduction) and may be unknown at compile time
  - For user-defined datatypes, polymorphic operators need to access a table of datatypes handles that are defined by the program
  ```

Example : rewriting MPI_SUM

```
/* this follows the MPI_User_function typedef */
void my_sum_op(void * b_in, void * b_inout,
               int * count, MPI_Datatype * d) {
    if (d == MPI_INT) {
        for (i=0; i<count; i++) {
            ((int*)b_inout)[i]+=((int *)b_in)[i];   }
    } else if (d == MPI_FLOAT) {
        for (i=0; i<count; i++) {
            ((float*)b_inout)[i]+=((float *)b_in)[i];}
    } else MPI_Abort (MPI_COMM_WORLD, -12345);
}

... ...

MPI_Op * op_sum;
MPI_Create_op (* my_sum_op, MPI_FALSE, op_sum)
```

- Very limited example: it only accepts INT and FLOAT types
- Can call specialized functions in each case (code reuse, hardware acceleration)

Datatypes and polymorphic operators

- Check the datatype
  - Compare the received datatype handle to a list of allowed handles, execute proper code
  - Simple if/else error if only one type is allowed
- Check and switch for polymorphic op.s
  - Operators that can handle several datatypes should employ data structures that avoid any excessive comparison overhead
    - E.g. an hash-map (perfect hash?) associating handles with code (function pointers) implementing each case of use of the operator
  - The overhead is usually negligible with respect to the communication overhead of a reduction or scan

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

Example : rewriting MPI_SUM (II)

```
/* this follows the MPI_User_function typedef */
void my_sum_op(void * b_in, void * b_inout,
               int * count, MPI_Datatype * d) {
    int my_type = my_hashtable_get(d);
    case (SUMOP_INT_T) : // MPI_INT
        sumintarrays((int *)b_in, (int*) b_inout, count);
        break;
    case (SUMOP_FLOAT_T) : // MPI_FLOAT
        sumfloatarrays((float *) b_in, (float *) b_inout, count);
            break;
    case (SUMOP_4BY4_FLOAT_T) : // user type example, 4*4 matrix
        sumfloatarrays((float *)b_in, (float*)b_inout, count*16);
        break;
    default : MPI_Abort (MPI_COMM_WORLD, -12345);
}
```

- In the example we assume that
  - a hashtable is filled with custom values for each recognized datatype
  - if the type is not in the hashtable, the default value returned (*unknown datatype*) triggers the default case

Scan

- ```
  int MPI_Scan(const void* sendbuf, void* recvbuf,
   int count,
           MPI_Datatype datatype, MPI_Op op,
           MPI_Comm comm)
  ```

- Applies a scan (parallel prefix) to the elements in corresponding position of the send buffers of the processes
- The scan works according to process rank
  - Process i receives the result of the combination of data from processes { 0, .. i }
  - Scan is the identity for process with rank 0
- If count>1 we have multiple scans within the same communication pattern
- With MPI_INPLACE in sendbuf, only the receive buffer is used
  ```

ExScan

- ```
  int MPI_Exscan(const void* sendbuf, void* recvbuf,
         int count,
         MPI_Datatype datatype, MPI_Op op,
         MPI_Comm comm)
  ```

- Same as MPI_Scan, but results are accumulated on the *following* process
  - Process with rank i gets the parallel prefix result of data contributed from processes 0.. i-1
  - Mnemonics: think of it as a “scan and shift”
  - Process 0 receives no data, and its receive buffer is not used by MPI_Exscan
  - MPI_IN_PLACE can be used in sendbuf
  ```

Other reduce collective operations

- MPI_Allreduce
  - Semantically equivalent to a reduce followed by a broadcast
  - May be implemented more efficiently, of course
- MPI_Reduce_scatter_block
  - Performs a reduction, then scatters the result buffer across the processes
  - Requires n*recvcount elements by each process, scatters the n blocks of recvcount elements of the result
  - Parameter recvcount is the number of elements received per process after the scatter
    - the overall reduction is computed on recvcount*N elements, where N is the communicator size.
- MPI_Reduce_scatter
  - Generalizes the scatter_block to a variable scatter (each process can receive a block of different size)
  - The recvcount is now an array of block sizes (the array is the same size as the communicator, see MPI_Scatterv)

References

- **With respect to MPI-3 standard**
  - Section 5.9 (Global reduction operators)
  - You can skip reduce_local