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

SPD Course
03/03/2015
Massimo Coppola

MPI communication semantics

- Message order is not guaranteed,
  - Only communications with same envelope are non-overtaking
- Different communicators do not allow message exchange
  - Unless you consider termination by error and deadlocks forms of communication
- No fairness provided
  - You have to code priorities yourself
  - Implementations *may* be fair, but you can't count on that
- Resources are limited
  - E.g. Do not assume buffers are always available, allocate them explicitly
  - E.g. You shall free structures and objects you are not going to use again
  - The limits are often within the library implementation, hard to discover in advance…

Point to point and communication buffers

- All communication primitives in MPI assume to work with communication buffers
  - How the buffer is used is implementation dependent, but you can specify many constraint
- The structure of the buffer
  - depends on your data structures
  - depends on your MPI implementation
  - depends on your machine hardware and on related optimizazions
  - shall never depend on your programming language
- The MPI Datatype abstractions aims at that

Primitive Data types (C bindings)

MPI_CHAR char
(treated as printable character)
MPI_SHORT signed short int
MPI_INT signed int
MPI_LONG signed long int
MPI_LONG_LONG_INT
signed long long int
MPI_LONG_LONG (as a synonym)
signed long long int
MPI_SIGNED_CHAR signed char
(treated as integral value)
MPI_UNSIGNED_CHAR unsigned char
(treated as integral value)
MPI_UNSIGNED_SHORT
unsigned short int
MPI_UNSIGNED unsigned int
MPI_UNSIGNED_LONG
unsigned long int
MPI_UNSIGNED_LONG_LONG
unsigned long long int
MPI_FLOAT float
MPI_DOUBLE double

MPI_LONG_DOUBLE long double
MPI_WCHAR wchar_t
(ISO C standard, see <stddef.h>)
(treated as printable character)
MPI_C_BOOL _Bool

Many special bit-sized types

MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t

MPI_C_COMPLEX float _Complex
MPI_C_FLOAT_COMPLEX
(as a synonym) float _Complex
MPI_C_DOUBLE_COMPLEX
double _Complex
MPI_C_LONG_DOUBLE_COMPLEX
long double _Complex
MPI_BYTE
MPI_PACKED

Datatype role in MPI

- Datatype
  - a descriptor used by the MPI implementation
  - holds information concerning a given kind of data structure
- Datatypes are opaque objects
  - Some are constant (**PRIMITIVE** datatypes)
  - More are user-defined (**DERIVED** datatypes)
    - to be explicitly defined before use, and destroyed after
- Defining/using a datatype **does not** allocate the data structure itself:
  - Allocation done by the host languages
  - Datatypes provide explicit memory layout information to MPI, more than the host language

Conversion and packing

- Data type information is essential to allow packing and unpacking of data within/from communication buffers
- MPI is a linked library → MPI datatypes provide type information to the runtime
- Data types known to MPI can be converted during communication
- For *derived* datatypes, more complex issues related to memory layout

MPI_SEND

MPI_SEND(buf, count, datatype, dest, tag, comm)

- IN buf — initial address of send buffer
- IN count — number of elements in send buffer (non-negative integer, **in datatypes**)
- IN datatype — datatype of each send buffer element (handle)
- IN dest — rank of destination
- IN tag — message tag
- IN comm — communicator (handle)

- *The amount of transferred data is not fixed*

MPI_RECV

MPI_RECV (buf, count, datatype, source, tag, comm, status)

- OUT buf initial address of receive buffer
- IN count number of elements in receive buffer (non-negative integer, **in datatypes**)
- IN datatype datatype of each receive buffer element (handle)
- IN source rank of source or **MPI_ANY_SOURCE**
- IN tag message tag or **MPI_ANY_TAG**
- IN comm communicator (handle)
- OUT status status object (Status)

- *The amount of received data is not fixed and can exceed the receiver's buffer size*

Return status

- MPI_Status
  structure filled in by many operations
  - not an opaque object, an ordinary C struct
  - special value MPI_IGNORE_STATUS (beware!!)
  - known fields: MPI_SOURCE, MPI_TAG, useful for wildcard Recv, as well as MPI_ERROR
  - additional fields are allowed, but are not defined by the standard or made openly accessible
  - Example: the actual *count* of received objects

- MPI_Get_count(MPI_Status *status,
  MPI_Datatype datatype, int *count)
  - MPI primitive used to retrieve the number of elements actually received

The NULL process

- MPI_PROC_NULL
  - Rank of a fictional process
  - Valid in every communicator and point-to-point
  - Communication will always succeed
  - A receive will always receive no data and not modify its buffer

Derived datatypes

- Abstract definition
  - Type map and type signature
- Program Definition
  - MPI constructors
- Local nature
  - They are not shared
  - In communications, type ***signatures*** and ***type maps*** for the data type used are checked
  - Need to be consolidated before use in communication primitives (MPI_Commit)

MPI TYPE CONSTRUCTORS

- Typemap & typesignatures
- Rules for matching Datatypes
- Size and extent
- Contiguous
- Vector
  - Count, blocklen, stride example
  - Row, column, diagonals (exercises)
  - Multiple rows
  - Stride<blocklen, negative strides
- Examples: composing datatypes
- Hvector
- Indexed
- Hindexed
- Standard send and recv: any_tag, any_source
- Send has modes, recv can be asymmetric, both can be incomplete

Typemaps and type signatures

- A datatype is defined by its memory layout
  - as a list of basic types and displacements
- Typemap

$$TM = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

- Type signature

$$TS = \{(type_0), \ldots, (type_{n-1})\}$$

  - Each $type_i$ is a basic type with a known size
- Size = the sum of sizes of all $type_i$
- Extent = the distance between the earliest and the latest byte occupied by a datatype
- Rules for matching Datatypes

Typemaps and type signatures

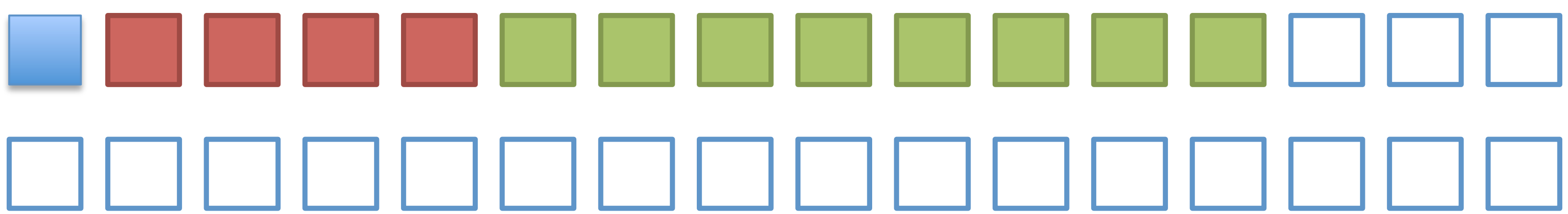
- Type map

$$TM = \{(\text{byte}, 0), (\text{int}, 1), (\text{double}, 5)\}$$

- Type signature

$$TS = \{(\text{byte}), (\text{int}), (\text{double})\}$$

- Size = 1+4+8 = 13
  - Note that we are assuming a 32 bit architecture here!
- Extent = 13



Typemaps and type signatures

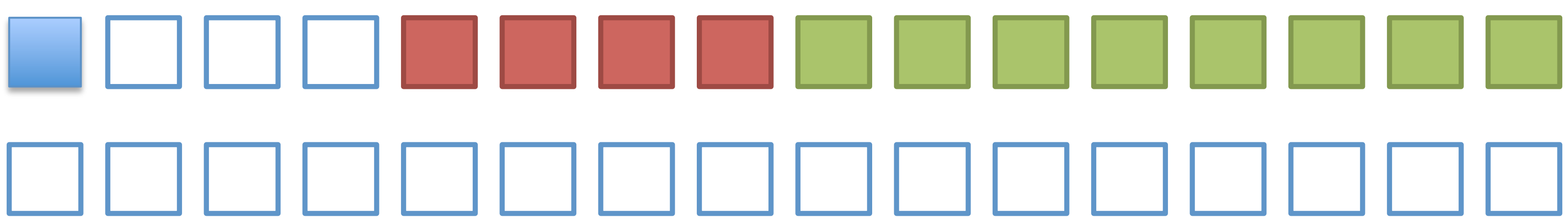
- Your compiler will likely add aligning constraints to basic types: let's assume ints are word aligned, and doubles are double-word aligned
- Type map

$$TM = \{(\textbf{byte}, 0), (\textbf{int}, 4), (\textbf{double}, 8)\}$$

- Type signature

$$TS = \{(\text{byte}), (\text{int}), (\text{double})\}$$

- Size = 1+4+8 = 13
- Extent = 16
- You need the padding for code execution, but you want to leave padding out of communication buffers
  - E.g. when sending large arrays of structures
  - Data packing and unpacking is automated in MPI



Matching rules for datatypes

- Typemaps are essential for **packing** into the communication buffer, and **unpacking**
- datatype in a send / recv couple must match
  - Datatypes are local to the process
  - Datatype descriptors (typemaps) can be passed among process (but not mandatory)
  - What really counts is the **type signature**
    - Do not "break" primitive types
    - "holes" in the data are dealt with by pack /unpack
- Datatype typemaps can have repeats
  - Disallowed on the receiver side!

Datatypes: shake before use!

- Before looking at the the core primitive for defining new derived datatypes, remember
- MPI_TYPE_COMMIT(datatype)
  - Mandatory before every actual use of a datatype!
  - Consolidates the datatype definition, making it permanent
  - Enables the new datatype for use in all non-datatype defining MPI primitives
    - e.g. commit before a point to point or a collective
  - May compile internal information needed to the MPI library runtime
    - e.g. : optimized routines for data packing & unpacking
- MPI_TYPE_FREE(datatype)
  - Free library memory used by a datatype that is no longer needed
  - Be sure that the datatype is not currently in use!

Contiguous Datatype

```
int MPI_Type_contiguous(int count,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- Create a plain array of identical elements
- No extra space between elements
- Overall size is count* number of elements

Contiguous Datatype

```
MPI_Datatype mytype;
MPI_Type_contiguous( 4, MPI_INT, &mytype);
MPI_Type_commit(mytype)
```

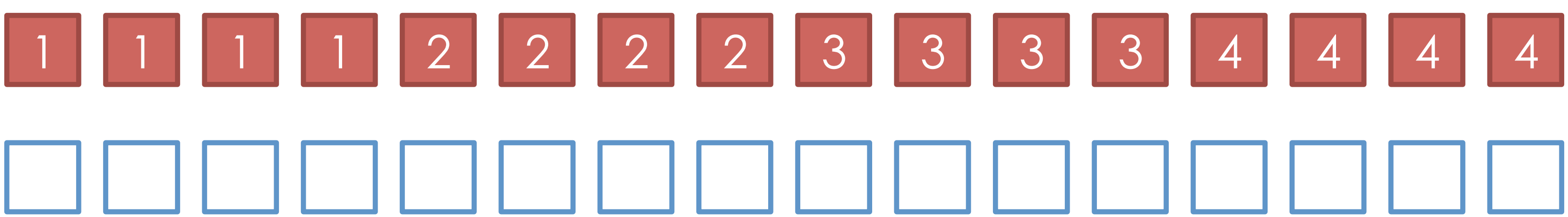
- Type map

$$TM = \{(\textbf{int}, 0), (\textbf{int}, 4), (\textbf{int}, 8), (\textbf{int}, 12),\}$$

- Type signature

$$TS = \{ (\textbf{int}), (\textbf{int}), (\textbf{int}), (\textbf{int})\}$$

- Size = 16
- Extent = 16



Vector Datatype

```
int MPI_Type_vector(int count, int blocklength,
    int stride, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

- Create a spaced array (a series of contiguous blocks with space in between)
- Count = number of blocks
- Blocklength = number of items in each block
- Stride = distance between *the start* of each block
- The size unit is the size of the inner datatype

Vector Datatype

```
MPI_Datatype mytype;
MPI_Type_vector( 4, 2, 4, MPI_BYTE, &mytype);
MPI_Type_commit(mytype)
```

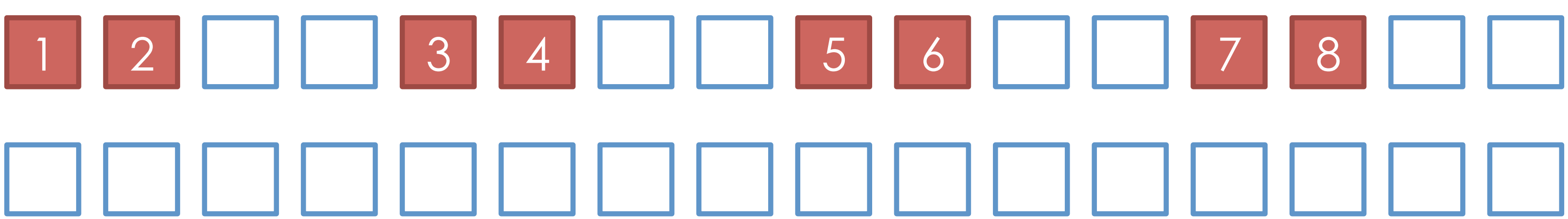
- Type map

  TM = {(**byte**, 0), (**byte**, 1), (**byte**, 4), (**byte**, 5), (**byte**, 8), (**byte**, 9), (**byte**, 12), (**byte**, 13)}

- Type signature

  TS = { (**byte**), (**byte**), (**byte**), (**byte**), (**byte**), (**byte**), (**byte**), (**byte**)}

- Size = 8
- Extent = 13



Vector Datatype

- What if stride is less than the blocklength?
- What if the stride is zero?

Hvector datatype

```
int MPI_Type_create_hvector(
    int count, int blocklength, MPI_Aint stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Create a vector of block with arbitrary alignment
- Same as the vector but:
  - The stride is an offset in **bytes** between each block starts
- Many other datatypes have an "H version" where some parameters are in byte units

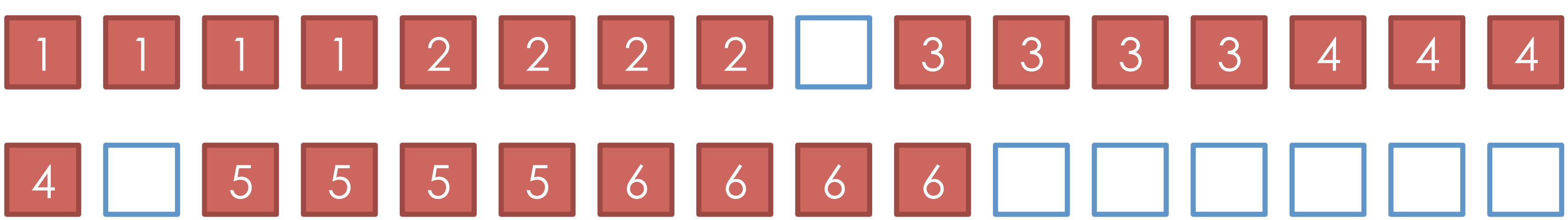
HVector Datatype

```
MPI_Datatype mytype;
MPI_Type_hvector( 3, 2, 9, MPI_INT, &mytype);
MPI_Type_commit(mytype)
```

- Type map

  TM = {(**int**, 0), (**int**, 4), (**int**, 9), (**int**, 13), (**int**, 18), (**int**, 22)}
- Type signature

  TS = { (**int**), (**int**), (**int**), (**int**), (**int**), (**int**)}
- Size = 24
- Extent = 26



Indexed datatype

```
int MPI_Type_indexed(
    int count, int *array_of_blocklengths,
    int *array_of_displacements,

    MPI_Datatype oldtype,MPI_Datatype *newtype)
```

- Blocks of different sizes
- Count is a number of blocks
- Length and position (w.r.t. structure start!) are specified for each block
- All in units of the inner datatype
- Some uses for this datatype: triangular matrixes, arrays of contiguous lists, reordering data structure blocks (e.g. matrix rows) as we communicate

Hindexed Datatype

```
int MPI_Type_create_hindexed(
    int count, int array_of_blocklengths[],
    MPI_Aint array_of_displacements[],

    MPI_Datatype oldtype, MPI_Datatype
    *newtype)
```

- Same as Indexed, but block positions are given in bytes
- Enhanced flexibility in memory layout

Struct Datatype

MPI_TYPE_CREATE_STRUCT (count, array_of _blocklengths, array_of _displacements, array_of _types, newtype)

IN count number of blocks (non-negative integer)

- also number of entries in arrays array_of _types, array_of _displacements and array_of _blocklengths

IN array_of _blocklength elements in each block (array of non-negative integer)

IN array_of _displacements byte displacement of each block (array of integer)

IN array_of _types type of elements in each block (array of handles to datatype objects)

OUT newtype new datatype (handle)

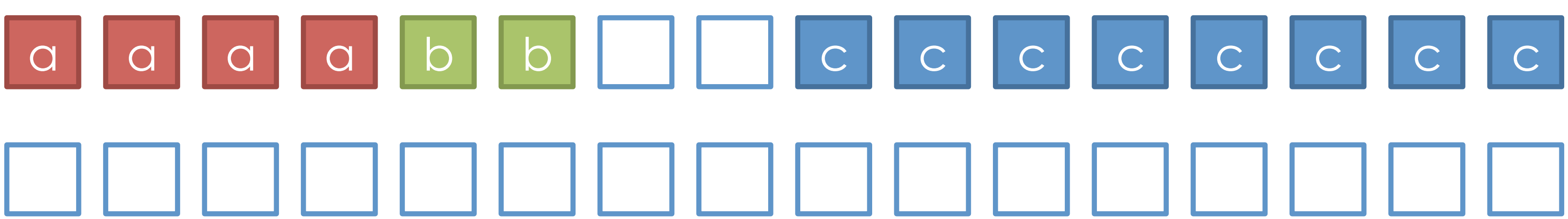
Struct Datatype

```
typedef struct  {
          int a; char b[2]; double c
     }
```

- Assuming 32 bit words, double-word aligned doubles etc…
- Type map

  TM = {(int, 0), (char, 5), (char, 6), (double, 8)}
- Type signature

  TS = { (int), (char), (char), (double)}
- Size = 14
- Extent = 16



MPI TYPE CONSTRUCTORS

- Typemap & typesignatures
- Rules for matching Datatypes
- Size and extent
- Contiguous
- Vector
  - Count, blocklen, stride example
  - Row, column, diagonals (exercises)
  - Multiple rows
  - Stride<blocklen, e.g. negative offsets
- Examples: composing datatypes
- Hvector
- Indexed
- Hindexed
- Struct
- A simple tool to display MPI typemaps : MPIMap
  http://computation.llnl.gov/casc/mpimap/

Exercises

- Start preparing for the lab sessions
  - Install a version of MPI which works on your O.S.
    - OpenMPI (active development)
    - LAM MPI (same team, only maintained)
    - MPICH (active development)
  - Check out details that have been skipped in the lessons
    - How to run programs, how to specify the mapping of processes on machines
    - Usually it is a file listing all available machines
    - How to check a process rank
  - Read the first chapters of the Wilkinson-Allen
    - Write at least a simple program that uses MPI_Comm_World, has a small fixed number of processes and communications and run it on your laptop
    - E.g. a trivial ping-pong program with 2 processes

Reference Texts

- MPI standard Relevant Material for 2^{nd} lesson
  - Chapter 3:
    - section 3.2 (blocking send and recv with details)
    - section 3.3 (datatype matching rules and meaning of conversion in MPI)
  - Chapter 4: sections with the specific datatype constructors discussed