pt2pt wg

FP16

Virtual MPI Forum Meeting
MPI Forum, Feb., 2018

Updates from the virtual meeting on Jan 31

Atsushi Hori
RIKEN AICS

Related Tickets

- #65 16-bit floating-point support for C/C++
  - ➡ MPI standard will not define FP16, but implementor can decide to support or not
- #66 define language-agnostic, IEEE types
  - ➡ Implementors can decide to implement new types
- #69 Definition and Explanation of External32
  - ➡ Left unchanged because of backward compatibility (wording might be updated)

New Ticket #74

- Replacement of ticket-65 and -66
- Stop introducing new types into the standard
  MPI standard only defines *fundamental* datatypes such as MPI_INT, MPI_FLOAT, and so on
  - decouple new type definitions from standard
  - give implementors the freedom of choice when to add and what to add (new basic datatypes)
- Instead,
  - Standard defines the *naming rule* of basic datatype names
  - Implementors can add new basic datatypes having the names following the naming tule

Definition of Terms

NEW

- *Basic datatypes*
  - primitive datatypes
    - can be appeared in the type maps
  - Two kinds
    - *~~Fundamental~~ Predefined datatypes*
    - *~~Additional~~ Optional datatypes*
- *Predefined datatypes*
  - Defined in the standard
- *Optional datatypes*
  - Added by an implementor

Modification to the standard

- Adding one section into Section 3.3 Data Type Matching and Data Conversion (and Chapter 4. Datatypes)

- Predefined datatype definition
- Optional datatype names
  - Datatype naming rule
  - Old basic datatype names become synonyms
- Introducing one new function

Impact to users

- Old datatype names are left unchanged as synonyms and no impact to current users

Details (1)

- Predefined Datatypes (in old names here)
  - C MPI_CHAR, MPI_INT, MPI_LONG, ...
  - Fortran MPI_CHARACTER, MPI_INTEGER, ...
  - MPI MPI_BYTE, MPI_PACKED, MPI_AINT, ...
- *Implementation-defined* Optional Datatypes
  - *Data Type Naming Rule* (later)
  - An implementation may add basic datatype(s) if the implementation and associated compiler support it (them).
  - The current data type names not compliant with the rule are thought to be synonyms.

Details (2)

- Advice to users (draft)
  - Excepting external32, MPI standard only defines the datatype names, regardless to the sizes of datatypes, data representation format, and if the data type is computed as is or any format conversion(s) is (are) accompanied with computations (4 calculus, bit ops, etc.). An implementation may add basic datatypes supported by the accompanied compiler. The datatype names added by an implementation must follow the datatype naming rule in the standard so that users can identify the datatypes of the underlying programming language from the datatype names used in MPI. Documentation on these additional datatypes should be provided by the implementor.
  - Actual storage size of a datatype depends on an implementation, and accompanied compiler (see also MPI_Type_size(x)). For example, MPI_LONG_INT has the size of 4 bytes or 8 bytes depending on the compiler and/or processor. In the usage of MPI-IO with “external32,” however, datatype sizes and formats are explicitly defined in this standard for compatibility and portability (see also MPI-IO Chapter XYZ).

Naming Rule Requirements

- From users' viewpoint
  - Users can identify the underlying datatypes from the MPI datatype names
- From implementors' viewpoint
  - Implementors can easily make new unique datatype name when they want to introduce new datatype.

Type Naming Rule (1/3)

- **MPI_<KIND>_<TYPENAME>[_<EXTRA>]**, or **MPI_<KIND>__<TYPENAME>[__<EXTRA>]**
  - MPI_C_LONG_LONG or MPI_C__LONG_LONG
  - MPI_CXX_FLOAT or MPI_CXX__FLOAT
  - MPI_F_DOUBLE or MPI_F__DOUBLE
  - MPI_F_REAL_2 or MPI_F__REAL__2, …
- MPI-defined types
  - MPI_MPI_BYTE or MPI_MPI__BYTE
  - MPI_MPI_PACKED, or MPI_MPI__PACKED, …
- Language agnostic types
  - MPI_IEEE_BINARY32 or MPI_IEEE__BINARY32
  - MPI_ISO_DECIMAL64 or MPI_ISO__DECIMAL64, …

Type Naming Rule (2/3)

- **MPI_<KIND>_<TYPENAME>[_<EXTRA>]** or **MPI_<KIND>__<TYPENAME>[__<EXTRA>]**
- KIND

Language	"C", "CXX", "F", … (filename extension?)
MPI-defined	"MPI"
Standard	"ISO", "IEEE", "ANSI", …

- Type Name (defined by language, shortest one)

C/C++	"int", "long long", …
Fortran	"INTEGER", "REAL", …
ISO/IEEE	"binary16", "decimal32", ..

- Extra

Fortran	"REAL(4)", "COMPLEX(8)", …

Type Naming Rule (3/3)

- MIN_LOC, MAX_LOC (if still needed)
  - **<TYPENAME0>__AND__<TYPENAME1>** or **<TYPENAME0>__AT__<TYPENAME1>**

  - MPI_C_FLOAT_AND_MPI_C_INT or MPI_C__FLOAT__AND__MPI_C__INT
  - MPI_C_FLOAT_AT_MPI_C_INT or MPI_C__FLOAT__AT__MPI_C__INT

Old and New Type Names (1/2)

C Kind		Ext32
Current Name	Proposed New Name	Size
MPI_CHAR	MPI_C_CHAR	1
MPI_SHORT	MPI_C_SHORT	2
MPI_INT	MPI_C_INT	4
MPI_LONG	MPI_C_LONG	4
MPI_LONG_LONG_INT	MPI_C_LONG_LONG	8
MPI_LONG_LONG	MPI_C_LONG_LONG	8
MPI_SIGNED_CHAR	MPI_C_SINGED_CHAR	1
M:PI_UNSIGNED_CHAR	MPI_C_UNSIGNED_CHAR	1
MPI_UNSIGNED_SHORT	MPI_C_UNSIGNED_SHORT	2
MPI_UNSIGNED	MPI_C_UNSIGNED	4
MPI_UNSIGNED_LONG	MPI_C_UNSIGNED_LONG	4
MPI_UNSIGNED_LONG_LONG	MPI_C_UNSIGNED_LONG_LONG	8
MPI_FLOAT	MPI_C_FLOAT	4
MPI_SHORT_FLOAT	MPI_C_SHORT_FLOAT	(2)
MPI_DOUBLE	MPI_C_DOUBLE	8
MPI_LONG_DOUBLE	MPI_C_LONG_DOUBLE	16
MPI_WCHAR	MPI_C_WCHAR	2
MPI_C_BOOL	MPI_C_BOOL	1
MPI_INT8_T	MPI_C_INT8_T	1
MPI_INT16_T	MPI_C_INT16_T	2
MPI_INT32_T	MPI_C_INT32_T	4
MPI_INT64_T	MPI_C_INT64_T	8
MPI_UINT8_T	MPI_C_UINT8_T	1
MPI_UINT16_T	MPI_C_UINT16_T	2
MPI_UINT32_T	MPI_C_UINT32_T	4
MPI_UINT64_T	MPI_C_UINT64_T	8
MPI_C_COMPLEX	MPI_C_COMPLEX	2*4
MPI_C_FLOAT_COMPLEX	MPI_C_COMPLEX	2*4
MPI_C_SHORT_COMPLEX	MPI_C_SHORT_COMPLEX	(2*2)
MPI_C_DOUBLE_COMPLEX	MPI_C_DOUBLE_COMPLEX	2*8
MPI_C_LONG_DOUBLE_COMPLEX	MPI_C_LONG_DOUBLE_COMPLEX	2*16

Fortran Kind		Ext32
Current Name	Proposed New Name	Size
MPI_CHARACTER	MPI_F_CHARACTER	1
MPI_LOGICAL	MPI_F_LOGICAL	4
MPI_INTEGER	MPI_F_INTEGER	4
MPI_REAL	MPI_F_REAL	4
MPI_DOUBLE_PRECISION	MPI_F_DOUBLE_PRECISION	8
MPI_COMPLEX	MPI_F_COMPLEX	2*4
MPI_DOUBLE_COMPLEX	MPI_F_DOUBLE_COMPLEX	2*8
(followings are optional datatypes)		
MPI_INTEGER1	MPI_F_INTEGER_1	1
MPI_INTEGER2	MPI_F_INTEGER_2	2
MPI_INTEGER4	MPI_F_INTEGER_4	4
MPI_INTEGER8	MPI_F_INTEGER_8	8
MPI_INTEGER16	MPI_F_INTEGER_16	16
MPI_REAL2	MPI_F_REAL_2	2
MPI_REAL4	MPI_F_REAL_4	4
MPI_REAL8	MPI_F_REAL_8	8
MPI_REAL16	MPI_F_REAL_16	16
MPI_COMPLEX4	MPI_F_COMPLEX_4	2*2
MPI_COMPLEX8	MPI_F_COMPLEX_8	2*4
MPI_COMPLEX16	MPI_F_COMPLEX_16	2*8
MPI_COMPLEX32	MPI_F_COMPLEX_32	2*16

- The new type names in blue are *fundamental datatypes*
- Ext32 sizes in brackets are not defined in the current standard (and future)

Old and New Type Names (2/2)

C++ Kind		Ext32
Current Name	**Proposed New Name**	**Size**
MPI_CXX_BOOL	**MPI_CXX_BOOL**	1
MPI_CXX_FLOAT_COMPLEX	**MPI_CXX_FLOAT_COMPLEX**	2*4
MPI_CXX_DOUBLE_COMPLEX	**MPI_CXX_DOUBLE_COMPLEX**	2*8
MPI_CXX_LONG_DOUBLE_COMPLEX	**MPI_CXX_LONG_DOUBLE_COMPLEX**	2*16

MPI Kind		Ext32
Current Name	**Proposed New Name**	**Size**
MPI_BYTE	**MPI_MPI_BYTE**	1
MPI_PACKED	**MPI_MPI_PACKED**	(1)
MPI_AINT	**MPI_MPI_AINT**	8
MPI_COUNT	**MPI_MPI_COUNT**	8
MPI_OFFSET	**MPI_MPI_OFFSET**	8

New Data Type Function

- int MPI_Type_equivalent(type0, type1, *flag)
  - *flag* is set, if *type0* and *type1* are having the same binary format and size. Otherwise *flag* is reset.
  - If derived datatypes are given, then the type maps of both types must be equivalent. Here, "equivalent type maps" means all basic datatype and offset pairs in two type maps are the "equivalent" defined by MPI_Type_equivalent().
  - ex)
    - MPI_Type_equivalent(MPI_SHORT_FLOAT,
      MPI_FLOAT, flag)
    - MPI_Type_equivalent(MPI_REAL,
      MPI_FLOAT, flag)

NG

New Data Type Functions NEW

- int MPI_Type_match(type0, type1, *flag)
  - On basic datatypes
    - if **type0** and **type1** have the same binary format and size, then it returns true.
  - On derived datatypes
    - if **type0** and **type1** have the same *datatype signatures* (not *type maps*), then it returns true.

- Beneficial to users
- Implementation msy not be easy

MPI_Type_match

NEW

- This leads to change the current type matching rule in the standard

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

The types of a send and receive match (phase two) if both operations use ***identical names***. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Section 4.2: the type MPI_PACKED can match any other type.

3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

The types of a send and receive match (phase two) if both operations use **the types on which MPI_Type_match() returns true**. That is, MPI_INTEGER matches MPI_INTEGER, MPI_REAL matches MPI_REAL, and so on. There is one exception to this rule, discussed in Section 4.2: the type MPI_PACKED can match any other type.

MPI_Type_match

NEW

- This leads to change the current type matching rule in the standard

MPI_TYPE_GET_CONTENTS

Rationale.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (End of rationale.)

↓

MPI_TYPE_GET_CONTENTS

Rationale.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, call MPI_Type_match function to determine the datatype involved. (End of rationale.)

MPI_TYPE_GET_NAME() *NEW*

- Advice to implementors
  - On predefined datatypes, a high quality implementation returns the distinct type name of equivalent types.

- Example
  - on MPI_INT and MPI_INT32_T
  - HQI: returned string depends on the type specified
    - MPI_TYPE_GET_NAME(MPI_INT, “MPI_INT”)
    - MPI_TYPE_GET_NAME(MPI_INT32_T, “MPI_INT32_T”)
  - LQI: returned string MAY NOT depend on the type specified
    - MPI_TYPE_GET_NAME(MPI_INT, “MPI_INT”)
    - MPI_TYPE_GET_NAME(MPI_INT32_T, “MPI_INT”)

Straw Votes (1/6)

- Do you agree that the overall of this proposal ?
  - Implementors can define any arbitrary optional basic datatypes.
  - Old datatype names are left unchanged as synonyms and no impact to current users
- Or, just add FP16 datatypes
  - Should wait for the compiler supports

Straw Votes (2/6)

- Definition of terms
  - *Predefined datatypes*
    - Defined in the standard
  - *Optional datatypes*
    - Added by an implementor

Straw Votes (3/6)

- int MPI_Type_get_name(MPI_Datatype datatype, char *type_name, int *resultlen)
  - Advice to implementors
    - A high quality implementation returns the distinct type name in the equivalent types.

Straw Votes (4/6)

- MPI_<KIND>_<TYPENAME>[_<EXTRA>] or MPI_<KIND>__<TYPENAME>[__<EXTRA>]
  - KIND
    - Language "C", "CXX", "F", … (filename extension?)
    - MPI-defined "MPI"
    - Standard "ISO", "IEEE", "ANSI", …
  - Type Name (defined by language, shortest one)
    - C/C++ "int", "long long", …
    - Fortran "INTEGER", "REAL", …
    - ISO/IEEE "binary16", "decimal32", ..
  - Extra
    - Fortran "REAL(4)", "COMPLEX(8)", …

Straw Votes (5/6)

- MIN_LOC, MAX_LOC (if still needed)
  - single underscore
    - <TYPENAME0>_AND_<TYPENAME1> or
    - <TYPENAME0>_AT_<TYPENAME1>
  - double underscore
    - <TYPENAME0>__AND__<TYPENAME1> or
    - <TYPENAME0>__AT__<TYPENAME1>

Straw Votes (6/6)

- int MPI_Type_match(type0, type1, *flag)
  - On basic datatypes, if **type0** and **type1** have the same binary format and size, then it returns true.
  - On derived datatypes, if they have the same *datatype signatures* (not *type maps*), then it returns true.

- Yes, we should have this
- No
  - Only works on basic datatypes
  - We do not need this at all