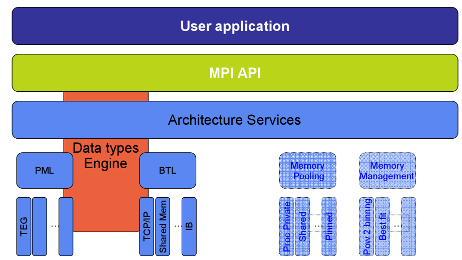
Data-types and conversion

George Bosilca

Point-to-Point Architecture



MPI Data-types

- How are they created ?
- Where are they used:
  - Point-to-point communications
  - One sided communications
  - MPI I/O
- They have different requirements !
- How are they used to convert the data ?
  - Efficiently represent and transfer data
  - Minimize memory usage

Some of MPI's Pre-Defined Datatypes

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER*2
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

User-Defined Datatypes

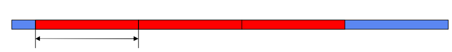
- Applications can define unique datatypes
  - Composition of other datatypes
  - MPI functions provided for common patterns
    - Contiguous
    - Vector
    - Indexed
    - …

➔ Always reduces to a type map of pre-defined datatypes

Contiguous Blocks

- Replication of a datatype into a contiguous buffer

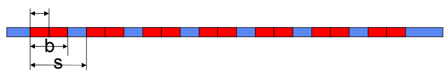
MPI_Type_contiguous(3, oldtype, newtype)



Vectors

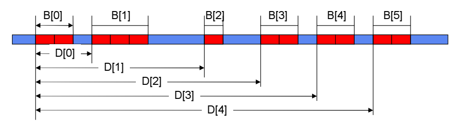
- Replication of a datatype into locations that consist of equally spaced blocks

MPI_Type_vector(7, 2, 3, oldtype, newtype)



Indexed Blocks

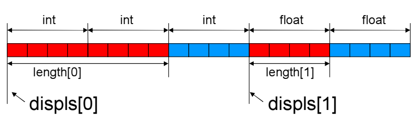
- Replication of an old datatype into a sequence of blocks
  - Each block can contain a different number of copies and have a different displacement



Arbitrary Structures

- The most general datatype constructor
- Allows each block to consist of replication of different datatypes

```
struct {
  int i[3];
  float f[2];
} array[100];
```



Data Representation

- Different across different machines
  - Length: 32 vs. 64 bits (vs. …?)
  - Endian: big vs. little
  - Architecture description
- Problems
  - No standard about the data length in the programming languages (C/C++)
  - No standard floating point data representation
    - IEEE Standard 754 Floating Point Numbers
      - Subnormals, infinities, NANs …
    - Same representation but different lengths for long doubles

Datatype Conversion

- "Data sent = data received"
- 2 types of conversions:
  - Representation conversion: change the binary representation (e.g., hex floating point to IEEE floating point)
  - Type conversion: convert from different types (e.g., int to float)

➔ Only representation conversion is allowed

Datatype Conversion

```
if (my_rank == root)
   MPI_Send(msg, 1, MPI_INT, …)
else
   MPI_Recv(msg, 1, MPI_INT, …)
```

✔

```
if (my_rank == root)
   MPI_Send(msg, 1, MPI_INT, …)
else
   MPI_Recv(msg, 1, MPI_FLOAT, …)
```

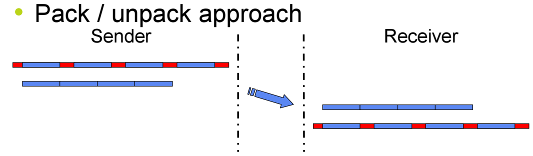
✘

What About Performance?

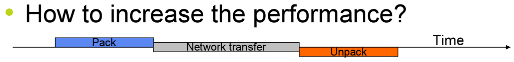
- Bad (old) way
  - User manually copies data to a pre-allocated buffer, or
  - User calls MPI_PACK and MPI_UNPACK
- Good (new) way
  - Trust the [modern] MPI library
  - Uses high performance MPI “datatypes”

What About Performance?

- Pack / unpack approach

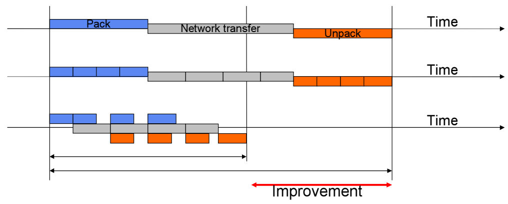


- 3 distinct steps: pack, network xfer, unpack
- No computation / communication overlap
- How to increase the performance?



Improving Performance

- Pipeline
  - Create computation / communication overlap
  - Split the computations in small slices



Improving Performance

- Other questions:
  - How to adapt to the network layer?
  - How to support RDMA operations?
  - How to handle heterogeneous communications?
  - How to split the data pack / unpack?
- Who handles all this?
  - MPI implementation can solve these problems
  - User-level applications cannot

Benefits

- Worst case: the most scattered data representation in memory (ie. one byte per line of cache) leads to 80-85% of the optimal bandwidth starting from message of size 256 bytes.
- Usualy, for HPL like data-types, Open MPI run at between 90 and 100% of the maximal bandwidth (depending on the size of the message)
- Up to 3 times faster than other MPI implementations, depending on the memory layout.

Internal Representation

- All information related to the MPI description: alignment, lower bound, upper bound, true lower bound, true upper bound, flags
- MPI args: used for get_content operation
- We create the data-type by adding new information on an already defined data-type (different than MPI).

MPI Combiner

- Describe how the data-type was created
- Store all the arguments of the MPI function, so the data can be recreated.
- We store it in a contiguous array.

One sided communication

- We need to move the data representation on the remote node
- We parse the combiner struct to create a contiguous array with all the information down to the predefined data-types.
- This packed array is send on the remote side, where it will be parsed to recreate the data description.
- For homogeneous architectures we can pass directly the optimized data description.

Loops and data-types

- Predefined data field

common	count	displ	extent

- Loop start

common	loops	extent	items

- Loop end

common	items	first displ	size

Example

- Contiguous 10 MPI_INT

LOOP,C	10	4	2
MPI_INT,C	1	0	4
END_LOOP	2	0	4

MPI_INT,C	10	0	4

Data optimizations

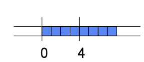
- MPI_Commit …
  - Optimize the representation in order to decrease the number of independent data and to increase the size of each of them.
  - Unroll loops
  - Rewrite loops with their prolog and epilog
  - Lose the type information if we are in an homogeneous environment.

MPI
DDT

Data optimizations

- Type collapse:
  - 2 similar types with identical properties will be mixed

INT,C	1	0	4
INT,C	1	4	4

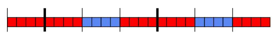


INT,C	2	0	4

Data optimizations

- Loop unrolling / reordering

MPI_INT,C	1	0	4
LOOP,C	10	12	3
MPI_INT,C	1	4	8
MPI_INT,C	1	12	4
END_LOOP	3	4	8



MPI_INT,C	2	0	4
LOOP,C	9	12	2
MPI_INT,C	2	12	4
END_LOOP	2	4	8
MPI_INT,C	1	120	4

Data optimizations

- How do we move inside this structure ?
- How do we know how many items are inside ?

MPI_INT,C	2	0	4
LOOP,C	9	12	2
MPI_INT,C	2	12	4
END_LOOP	2	4	8
MPI_INT,C	1	120	4
END_LOOP	5	0	84

Conversion

- The data representation is not enough in order to perform representation conversion
  - Endianness
  - Shrink/Expand the number of bits in the exponent and mantissa
  - Change the size of the data
- The conversion is done by a convertor
- No XDR
- Receiver make right (easy to send)

Convertor

- Created based on 2 architectures: local and remote.
- Once the data-type is attached is can compute the local and remote size
- Can convert the data segment by segment: iovec conversion
  - For performance reasons there is no room for recursivity

Convertor

- The stack:

index	count	end_loop	displ

MPI_INT,C	2	0	4
LOOP,C	9	12	2
MPI_INT,C	2	12	4
END_LOOP	2	4	8
MPI_INT,C	1	120	4

-1	1	5	0
1	9	4	0

Convertor: How to

- Creating a convertor is a costly operation
  - Should be avoided in the critical path
  - Master convertor
  - Then clone it or copy it (!)
  - Once we have a initialized convertor we can prepare it by attaching the data and count
    - Specialized preparation: pack and unpack
- Position in the data: another costly operation
  - Problem with the data boundaries …

Convertor: How to

- Once correctly setup
  - Pack
  - Unpack
- Checksum computation
- CRC
- Predefined data-type boundaries problem
- Convertor personalization
  - Memory allocation function
  - Using NULL pointers

Convertor: How to

- Sender
  - Create the convertor and set it to position 0
  - Until the end call ompi_convertor_pack in a loop
  - Release the convertor
- Receiver
  - Create the convertor and set it to position 0
  - Until the end call ompi_convertor_unpack in a loop
  - Release the convertor

Easy isn't it ?!

Convertor: How to

- In fact the receive is more difficult
  - Additional constraints
    - Fragments not received in the expected order
    - Fragments not received (dropped packets)
    - Fragments corrupted
    - Fragments stop in the middle of a predefined data-type …
  - Do we look for performance ?