Library Development with MPI:

Attributes, Request objects, Group communicator creation, Local reductions and Datatypes

Jesper Larsson Träff, Ioannis Vardas
traff@par.tuwien.ac.at

TU Wien (Vienna University of Technology)
Faculty of Informatics, Institute of Computer Engineering
Research Group Parallel Computing

MPI for (function-based) library writing

Torsten Hoefler, Marc Snir: Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. EuroMPI 2011: 345-355

MPI for (function-based) library writing

```
int MYCOMMLIB_OpX(arg1, *arg2, …,
                  MPIobject1, *MPIobject2, …)
{
  static var;

  // All the code

  func(arg1,arg2,…);

  MPI_Operation(MPIobject1);
  MPI_Operation(*MPIobject2);
}
```

Bunch of functions and object definitions to support application specific algorithms

Arguments for information transfer between library and application, and between different library operations

```
int MYCOMMLIB_OpX(arg1, *arg2, …,
                  MPIobject1, *MPIobject2, …)
{
  static var; // internal state

  // All the code

  func(arg1,arg2,…);

  MPI_Operation(MPIobje  t1);
  MPI_Operation(*MPIobj  t2);
}
```

Library may or may not have init-operation for initializing internal state

MPI objects:
Communicators, windows, groups, requests, datatypes

```
int MYCOMMLIB_OpX(arg1, *arg2, …,
                  MPIobject1, *MPIobject2, …)
{
  static var; // internal state

  // All the code

  func(arg1,arg2,…);

  MPI_Operation(MPIobject1);
  MPI_Operation(*MPIobject2);
}
```

Progress/completion semantics (blocking, non-blocking)

```
int MYCOMMLIB_OpX(arg1, *arg2, …,
                  MPIobject1, *MPIobject2, …)
{
  static var; // internal state

  // All the code

  func(arg1,arg2,…);

  MPI_Operation(MPIobject1);
  MPI_Operation(*MPIobject2);
}
```

Operations involving (opaque) MPI objects

Experience, three concrete (types of) libraries

1. (Collective) communication libraries
2. A profiling library
3. Libraries for linear algebra (non-consecutive data)

Observation:
Many parts of MPI (collectives, virtual topology functionality) can be implemented, exactly as specified, as libraries

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. Parallel Comput. 107: 102826 (2021)

Collective communication libraries (I)

```
Bcast_lane(buffer,…,comm);
Allgather_lane(sendbuf,…,recvbuf,…,comm);
Reduce_scatter_lane(sendbuf,recvbuf,…,comm);
…
```

New implementations of (all) MPI collectives with same signatures (interfaces), better for hierarchical, multi-lane clusters

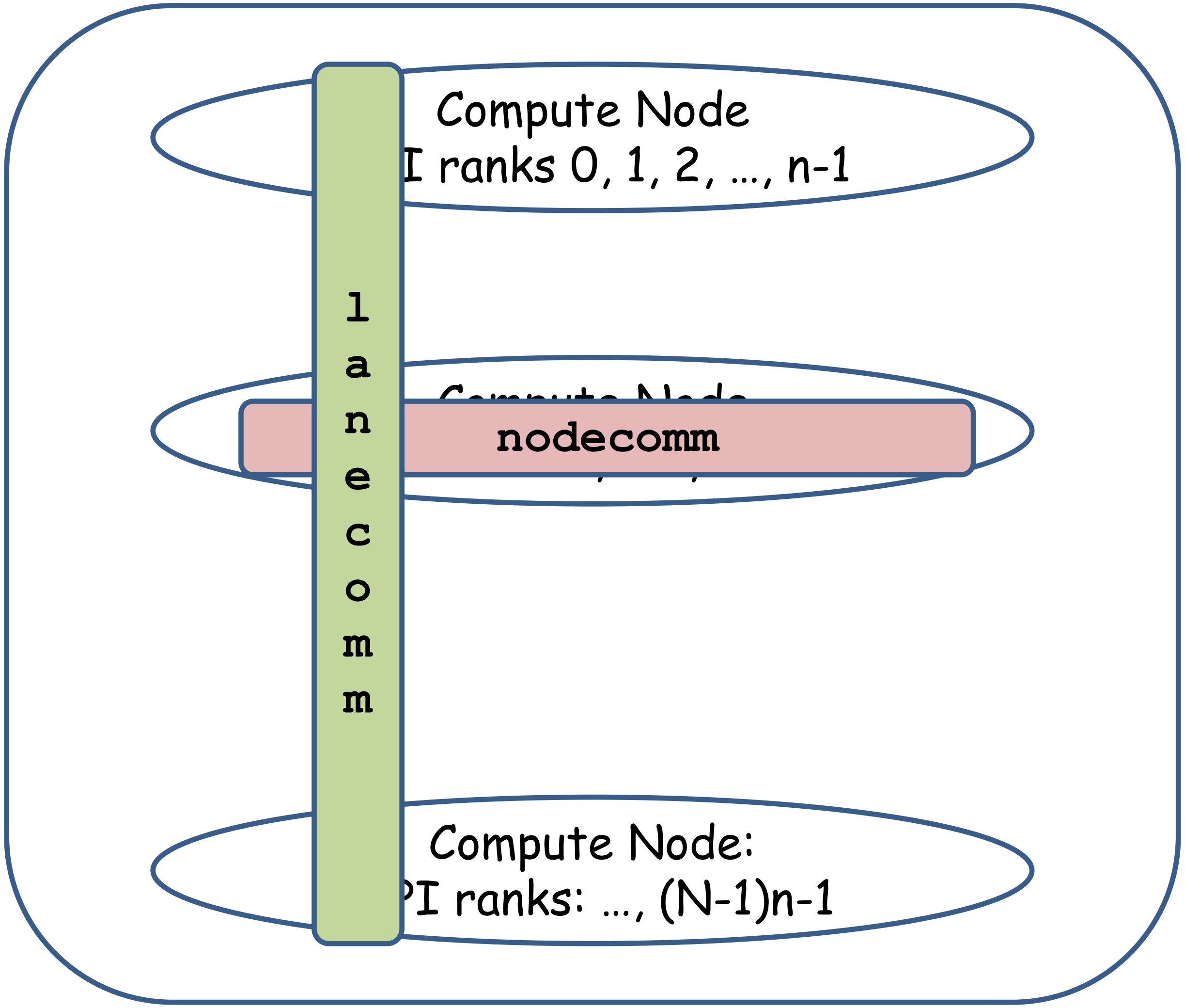
Jesper Larsson Träff, Sascha Hunold: Decomposing MPI Collectives for Exploiting Multi-lane Communication. CLUSTER 2020: 270-280

Idea:
Decompose the communication into intra-node and inter-node (lane) collective communication, with no redundancy in inter-node communication. Each process shall belong to a **`nodecomm`** and a **`lanecomm`**.

For each collective:

1. Split comm into **`nodecomm`** and **`lanecomm`** with **`MPI_Comm_split_type`** and **`MPI_Comm_split`**
2. Collective on (parts of data) on **`nodecomm`**
3. Collective on (parts of data) on **`lanecomm`**

Regular communicator **comm**



Each process in one **lanecomm** and one **nodecomm**.

As many **lanecomm** as processes per node.

As many **nodecomm** as nodes

Idea:
Decompose the communication into intra-node and inter-node (lane) communication, no redundancy in inter-node communication. Each process shall belong to a **`nodecomm`** and a **`lanecomm`**.

For each collective:

1. Split comm into **`nodecomm`** and **`lanecomm`** with **`MPI_Comm_split_type`** and **`MPI_Comm_split`**
2. Collective on (parts of data) on **`nodecomm`**
3. Collective on (parts of data) on **`lanecomm`**

Very(!) expensive

Solution (as in MPI Standard):

1. Split communicator lazily at first call of library collective on **`comm`**
2. Create new attribute key
3. Cache **`nodecomm`** and **`lanecomm`** as MPI attribute on **`comm`**

Subsequent collectives look up (and store as internal state) **`nodecomm`** and **`lanecomm`**

<u>Usage for libraries</u>:
Attributes transfer information on MPI objects across library functions

Latency of attribute caching

Collectives must be fast, also for small data: Latency!

Attribute caching and retrieval should be fast

Question: How good are current MPI libraries?

(what can we expect: linear, logarithmic, constant time in number of attributes?)

Benchmark them!

Our benchmark times this sequence of attribute operations:

1. Generate n keys
2. Cache n (empty) attributes
3. m lookups (in order of creation)
4. m lookups (in reverse order of creation)
5. Free n attributes
6. Delete n keys

Total: 4n+2mn attribute operations. Benchmark is repeated 95 times, average and minimum time recorded

Our system:
One node (out of 36) with 2x16-core Intel Xeon Gold 6130F processors at 2.1GHz

Our MPI libraries: MPICH 4.0.2, MVAPICH 2.3.7, IntelMPI 2021.8, OpenMPI 4.1.4

Results for MPICH 4.0.2, MVAPICH 2.3.7, IntelMPI 2021.8 similar (here: MVAPIC 2.3.7)

p	n	m	ops	Time/op (μs)
1	10	1000	20040	0.025
1	100	1000	200400	0.155
1	1000	1000	2004000	2.048
32	10	1000	20040	0.026
32	100	1000	200400	0.163
32	1000	1000	2004000	2.083

More than linear in number of attributes

Independent of number of processes per node!

Qualitatively similar results with Cray MPICH 8.1.23 on faster AMD EPYC system (LUMI)

Results for OpenMPI 4.1.4

OpenMPI seems to use better data structures than MPICH (variants)

p	n	m	ops	Time/op (µs)
1	10	1000	20040	0.046
1	100	1000	200400	0.046
1	1000	1000	2004000	0.053
32	10	1000	20040	0.047
32	100	1000	200400	0.046
32	1000	1000	2004000	0.054

Constant in number of attributes

Independent of number of processes per node!

Panel tomorrow?

What we did not implement:

How good is MPI support for writing non-blocking libraries?

```
Ibcast_lane(buffer,…,comm);
Iallgather_lane(sendbuf,…,recvbuf,…,comm);
Ireduce_scatter_lane(sendbuf,recvbuf,…,comm);
…
```

Collective library with non-blocking semantics

Blocking **`MPI_Comm_split()`** etc. would compromise non-blocking semantics (one problem out of many)

MPI standard does not have non-blocking versions of **`MPI_Comm_split()`** etc.

The profiling library **`mpisee`**

Idea:
Profile communication operations on their communicators.
Record for each communicator in application:

- size of communicator
- number of calls for each MPI operation on communicator
- data volume for each MPI operation on communicator (bucketed)
- total time for each MPI operation on communicator

Old-fashioned MPI profiling interface to intercept calls and manage profiling data

Ioannis Vardas, Sascha Hunold, Jordy I. Ajanohoun, Jesper Larsson Träff: `mpisee`: MPI Profiling for Communication and Communicator Structure. IPDPS Workshops 2022: 520-529

The case for attributes on MPI_Request objects (?)

```
MPI_Ibcast(buffer,…,comm,&request);

…

MPI_Wait(&request,&status);
```

Latency, time spent(wasted) in `MPI_Ibcast()` recorded here

Nobody else had this problem?

`mpiisee`:
From **`request`** object needs to find communicator (to attach profiling information to) and operation

No way to do this in current MPI

`mpisee` solution (unsatisfactory):
Use own hash-table with **`request`** handle as key to establish link between **`request`** object and communicator

Fully portable MPI solution: Either

- cache communicator as attribute on request object, **`MPI_Request_set_attr()`** …, or
- make provision to retrieve communicator/window (and operation) from **`request`** object

MPI supports attributes (only) for communicators, windows, datatypes

Why? Rationale?

Attributes on request objects (rationale not to?):

- Request object disappears (**`MPI_REQUEST_NULL`**) at completion with **`MPI_Wait(&request,&status);`**
- Attributes would have to be freed at completion, possibly too harmful to performance

Retrieving communicator from request object (viable solution?):

- Separate function **`MPI_Request_comm(request,&comm`**);, or
- let communicator/window be part of **`status`** object

```
MPI_Test(&request,&flag,&status);

MPI_Get_comm(&status,&comm);
MPI_Get_window(&status,&window);
```

Attribute on **`request`** object better solution? Copy attribute to **`status`**?

But what about the operation?

Handle to implement non-blocking libraries?

Partially collective semantics

`mpisee`:
Our implementation need a unique name/key for each new (sub)communicator.

This is a well-known problem

Markus Geimer, Marc-André Hermanns, Christian Siebert, Felix Wolf, Brian J. N. Wylie: Scaling Performance Tool MPI Communicator Management. EuroMPI 2011: 178-187
James Dinan, David Goodell, William Gropp, Rajeev Thakur, Pavan Balaji: Efficient Multithreaded Context ID Allocation in MPI. EuroMPI 2012: 57-66
James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, Abhinav Vishnu: Noncollective Communicator Creation in MPI. EuroMPI 2011: 282-291

`mpisee`:
Our implementation need a unique name/key for each new (sub)communicator.

Problematic MPI operations:

- **`MPI_Comm_create_group()`**
- **`MPI_Comm_create_from_group()`**

Our solution requires collective operation on “parent”, calling communicator; but these calls are not collective on the calling communicator

`mpisee`:
Our implementation need a unique name/key for each new (sub)communicator.

Better:
Let MPI do the work, use attributes on new communicator to store profiling information (see paper for sketch, to be done).

Collective communication libraries (II)

Collective reduction-like operations may require

1. A = A+B;
2. A = B+A;
3. A = B+C;

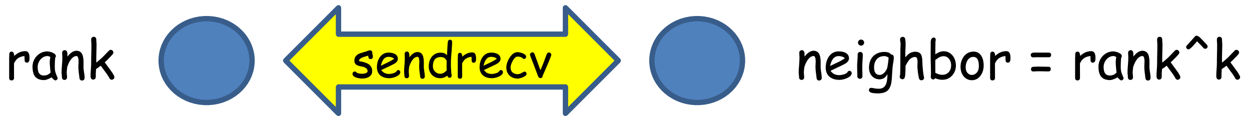
May be different if "+" is not commutative

on MPI objects with MPI operations (`MPI_SUM`, user-defined, …)

`MPI_Reduce_local(B,A,…,op);` is only A = B+A; (2)

Jesper Larsson Träff, Sascha Hunold, Ioannis Vardas, Nikolaus manes Funk: Uniform Algorithms for Reduce-scatter and (most) other Collectives for MPI. CLUSTER 2023. To appear

Example: Butterfly-like algorithm, order matters A+B≠B+A



rank

sendrecv

neighbor = rank^k

```
neighbor = rank^k;
MPI_Sendrecv(recvbuf,…,neighbor,
             tempbuf,…,neighbor,…,comm);
if (rank<neighbor) {
  MPI_Reduce_local(tempbuf,recvbuf,…,op);
} else {
  MPI_Reduce_local(recvbuf,tempbuf,…,op);
  MPICPY(recvbuf,tempbuf,…);
}
// invariant: Partial result in recvbuf
```

Extra, special copy needed

The case for a 3-argument local reduction function

Would be convenient to have “natural”, 3-argument local reduction in MPI.

But does the extra copy really hurt performance?

Benchmark!

Our benchmark implements A = B+C in three different ways:

```
(1)    for (i<n) A[i] = C[i];
       for (i<n) A[i] = B[i]+A[i];

(2)    for (i<n) A[i] = B[i]+C[i];

(3)    MPICPY(A,C,n);
       MPI_Reduce_local(B,A,n);
```

Times for n=10, 100, 1000, 10000, 100000, 1000000 (`int`) and 95 repetitions, average and minimum times recorded

OpenMPI 4.1.4

All (min)times in µs

p	n	2-arg	3-arg	MPI_Reduce_local
1	10	0.050	0.042	0.336
1	100	0.065	0.059	0.353
1	1000	0.293	0.218	0.475
1	10000	4.018	2.683	3.327
1	100000	45.98	34.48	44.67
1	1000000	1082	548	1039
32	10	0.055	0.051	0.371
32	100	0.070	0.061	0.370
32	1000	0.319	0.223	0.526
32	10000	3.919	2.668	3.668
32	100000	48.33	39.09	46.78
32	1000000	4255	3280	4178

Suspiciously high latency

Extra copy can hurt

Proposal, with full flexibility avoiding overlapping send- and receive buffers (use **`MPI_IN_PLACE`** for this):

```
MPI_Reduce_locals(A,B,C,n,datatype,op);
is C = A+B;
MPI_Reduce_locals(MPI_IN_PLACE,B,A,
                  n,datatype,op);
is A = A+B;
MPI_Reduce_locals(B,MPI_IN_PLACE,A,
                  n,datatype,op);
is A = B+A;

MPI_Reduce_locals(B,B,A,n,datatype,op);
is A = B+B;
MPI_Reduce_locals(MPI_IN_PLACE,MPI_IN_PLACE,
                  n,datatype,op);
is A = A+A;
```

Exploiting the MPI standard (typed, local copy)

There is no

```
MPICPY(recvbuf,recvcount,recvtype,
       sendbuf,sendcount,sendtype);
```

in MPI standard

But none is needed:

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,0,TAG,
             recvbuf,recvcount,recvtype,0,TAG,
             MPI_COMM_SELF,MPI_STATUS_IGNORE);
```

indicates process local operation

Exploiting the MPI standard (typed, local copy)

There is no

```
MPICPY(recvbuf,recvcount,recvtype,
       sendbuf,sendcount,sendtype);
```

in MPI standard

But none is needed (even better):

NB:
Not in paper

```
MPI_Allgather(sendbuf,sendcount,sendtype,
              recvbuf,recvcount,recvtype,
              MPI_COMM_SELF);
```

indicates process local operation

How good is typed `MPICPY(recvbuf,…,sendbuf,…);`?

(What can we expect? Comparable to `memcpy();` for simple, consecutive datatypes?)

Benchmark!

Our benchmark compares against **`memcpy();`** between contiguous buffers of double's and MPI derived, dense datatypes over **`MPI_DOUBLE`** that enumerates an mxn matrix:

- double: mxn **`MPI_DOUBLE`**
- col: **`MPI_Vector`** describing one column of m rows
- swap: **`MPI_Indexed`** describing swap of first and last row

Average and minimum times, 95 repetitions

Expectations:

- `MPI_DOUBLE` comparable to `memcpy(double);`
- When sendtype=recvtype, comparable to `memcpy();` because the datatypes describe the full matrix (so order does not matter)

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (µs)	Time (µs)
memcpy	1	20	100	0.061	
double-double	1			0.343	
double-col	1			1.534	
col-double	1			1.507	
col-col	1			2.691	
double-swap	1			0.529	
swap-double	1			0.525	
swap-swap	1			0.715	
col-swap	1			1.686	
swap-col	1			1.723	

`MPICPY()` as **`MPI_Sendrecv(MPI_COMM_SELF)`**

As `MPI_Allgather(MPI_COMM_SELF)`

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (µs)	Time (µs)
memcpy	1	20	100	0.061	0.060
double-double	1			0.343	0.091
double-col	1			1.534	1.305
col-double	1			1.507	1.267
col-col	1			2.691	1.143
double-swap	1			0.529	0.294
swap-double	1			0.525	0.299
swap-swap	1			0.715	0.122
col-swap	1			1.686	1.324
swap-col	1			1.723	1.347

`MPICPY()` as **`MPI_Sendrecv(MPI_COMM_SELF)`**

As `MPI_Allgather(MPI_COMM_SELF)`

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (µs)	Time (µs)
memcpy	32	20	100	0.069	0.078
double-double	32			0.391	0.111
double-col	32			1.637	1.371
col-double	32			1.582	1.327
col-col	32			2.801	1.186
double-swap	32			0.609	0.373
swap-double	32			0.585	0.351
swap-swap	32			0.781	0.153
col-swap	32			1.789	1.409
swap-col	32			1.823	1.444

`MPICPY()` as **`MPI_Sendrecv(MPI_COMM_SELF)`**

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (μs)	Time (μs)
memcpy	1	200	10000	3044	
double-double	1			2983	
double-col	1			15607	
col-double	1			12524	
col-col	1			33162	
double-swap	1			5737	
swap-double	1			5674	
swap-swap	1			14679	
col-swap	1			25022	
swap-col	1			30281	

`MPICPY()` as **`MPI_Sendrecv(MPI_COMM_SELF)`**

As `MPI_Allgather(MPI_COMM_SELF)`

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (µs)	Time (µs)
memcpy	1	200	10000	3044	2986
double-double	1			2983	2547
double-col	1			15607	12161
col-double	1			12524	11707
col-col	1			33162	13019
double-swap	1			5737	3363
swap-double	1			5674	3387
swap-swap	1			14679	2914
col-swap	1			25022	11684
swap-col	1			30281	21193

`MPICPY()` as **`MPI_Sendrecv(MPI_COMM_SELF)`**

As `MPI_Allgather(MPI_COMM_SELF)`

Results with OpenMPI 4.1.4 (there are differences between libraries!)

	p	m	n	Time (μs)	Time (μs)
memcpy	32	200	10000	7539	7271
double-double	32			7549	7586
double-col	32			23751	17345
col-double	32			22017	14207
col-col	32			45734	19413
double-swap	32			13924	7493
swap-double	32			13921	7503
swap-swap	32			28363	7300
col-swap	32			41302	14198
swap-col	32			43699	17344

`MPICPY()` as `MPI_Sendrecv(MPI_COMM_SELF)`

“Advice to users”:

Use

`MPI_Allgather(sendbuf,…,recvbuf,…,MPI_COMM_SELF);`

for correctly typed, local copy operations

“Advice to implementors”: More can be done…

Faisal Ghias Mir, Jesper Larsson Träff: Constructing MPI Input-output Datatypes for Efficient Transpacking. PVM/MPI 2008: 141-150

Linear algebra libraries with MPI derived data types

MPI derived datatypes are really opaque objects!

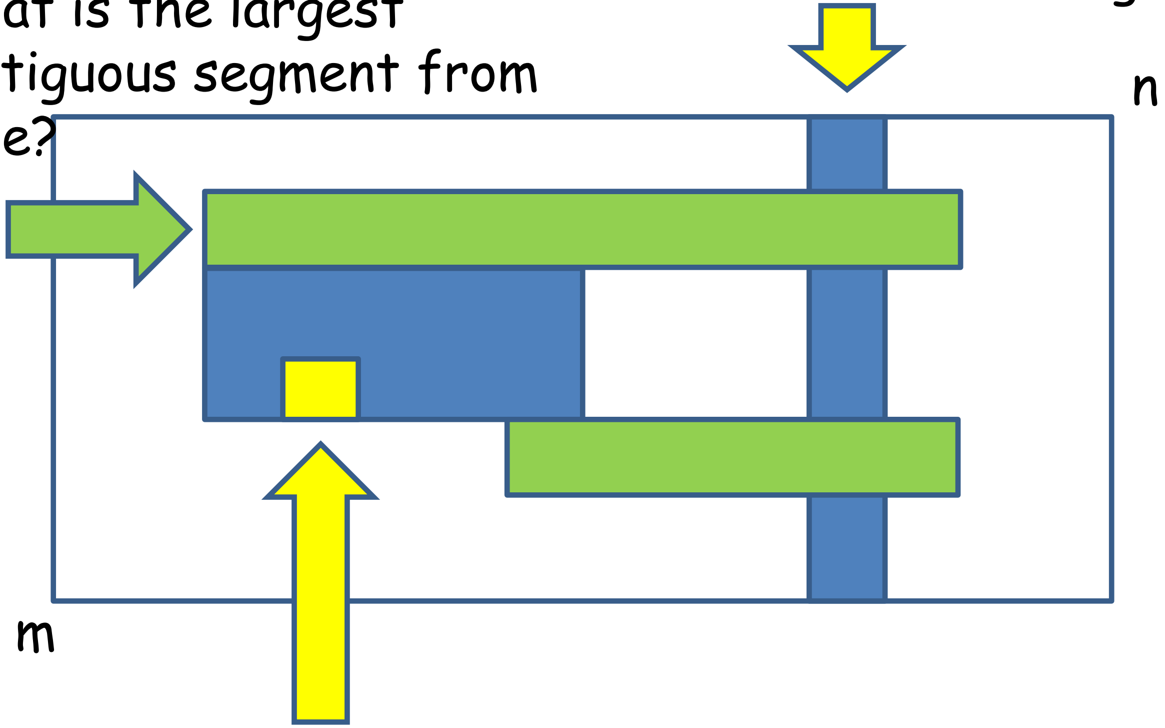
- No concept of type equivalence, no comparison functions
- No navigation
- No basetype query

Only (heavy) decoding functionality (that does not even guarantee that a user-defined datatype can be rebuilt exactly as constructed)

Jesper Larsson Träff: A Library for Advanced Datatype Programming. EuroMPI 2016: 98-107
Jesper Larsson Träff: Signature Datatypes for Type Correct Collective Operations, Revisited. EuroMPI 2020: 81-88

- Is this row contiguous?
- What is the largest contiguous segment from here?

Is this column homogeneous?



- What is the type of this element (**`MPI_DOUBLE`**)?
- What is the offset/displacement of this element

Conclusion

For MPI Standard, orthogonality:

- Attributes on all objects?
- If some feature is defined on some arguments/operations from a class of arguments/operations, it should be defined on all arguments/operations from the class
- Consistent progress semantics for classes of MPI operations
- Non-blocking communicator creation, **`MPI_Comm_Isplit();`** etc.

Panel tomorrow?

- Typed **`MPICPY()`** as collective **`MPI_Allgather();`** performs quite well…
- 3-argument local reduction operation, please!
- Functionality for working with complex data described by MPI derived data types