BigCount Solutions for MPI-4

MPI Forum

Chicago, USA, May 2019

Chattanooga meeting results

A concise summary

Check out these awesome function pointers for MPI!

But you didn’t solve BigCount

...but function pointers are shiny and awesome!

Chattanooga takeaways

1. Function pointers for C / objects for Fortran for MPI are neat
2. Function pointers would require a major rewrite of apps
    a. Backwards compatibility will be… tricky
3. *The real issue is to solve BigCount for MPI-4*

Solving BigCount

List of functions affected (133)

MPI_Accumulate
MPI_Allgather
MPI_Allgatherv
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Alltoallw
MPI_Bcast
MPI_Bsend
MPI_Bsend_init
MPI_CONVERSION_FN_NULL
MPI_Comm_spawn_multiple
MPI_Dist_graph_neighbors_count
MPI_Exscan
MPI_File_iread
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at
MPI_File_iwrite_at_all
MPI_File_iwrite_shared
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin
MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_shared
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_at
MPI_File_write_at_all
MPI_File_write_at_all_begin
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_shared
MPI_Gather
MPI_Gatherv
MPI_Get
MPI_Get_accumulate
MPI_Get_count
MPI_Get_elements
MPI_Get_elements_x
MPI_Graph_neighbors_count
MPI_Iallgather
MPI_Iallgatherv
MPI_Iallreduce
MPI_Ialltoall
MPI_Ialltoallv
MPI_Ialltoallw
MPI_Ibcast
MPI_Ibsend
MPI_Iexscan
MPI_Igather
MPI_Igatherv
MPI_Imrecv
MPI_Ineighbor_allgather
MPI_Ineighbor_allgatherv
MPI_Ineighbor_alltoall
MPI_Ineighbor_alltoallv
MPI_Ineighbor_alltoallw
MPI_Irecv
MPI_Ireduce
MPI_Ireduce_scatter
MPI_Ireduce_scatter_block
MPI_Irsend
MPI_Iscan
MPI_Iscatter
MPI_Iscatterv
MPI_Isend
MPI_Issend
MPI_Mrecv
MPI_Neighbor_allgather
MPI_Neighbor_allgatherv
MPI_Neighbor_alltoall
MPI_Neighbor_alltoallv
MPI_Neighbor_alltoallw
MPI_Pack
MPI_Pack_external
MPI_Pack_external_size
MPI_Pack_size
MPI_Put
MPI_Raccumulate
MPI_Recv
MPI_Recv_init
MPI_Reduce
MPI_Reduce_local
MPI_Reduce_scatter
MPI_Reduce_scatter_block
MPI_Rget
MPI_Rget_accumulate
MPI_Rput
MPI_Rsend
MPI_Rsend_init
MPI_Scan
MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend
MPI_Ssend_init
MPI_Startall
MPI_Status_set_elements
MPI_Status_set_elements_x
MPI_T_cvar_handle_alloc
MPI_T_pvar_handle_alloc
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Type_contiguous
MPI_Type_create_hindexed
MPI_Type_create_hindexed_block
MPI_Type_create_hvector
MPI_Type_create_indexed_block
MPI_Type_create_struct
MPI_Type_get_extent_x
MPI_Type_get_true_extent_x
MPI_Type_indexed
MPI_Type_size_x
MPI_Type_vector
MPI_Unpack
MPI_Unpack_external
MPI_Waitall
MPI_Waitany
MPI_Waitsome

List of functions affected (133)

MPI_Accumulate
MPI_Allgather
MPI_Allgatherv
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Alltoallw
MPI_Bcast
MPI_Bsend
MPI_Bsend_init
MPI_CONVERSION_FN_NULL
MPI_Comm_spawn_multiple
MPI_Dist_graph_neighbors_count
MPI_Exscan
MPI_File_iread
MPI_File_iread_all
MPI_File_iread_at
MPI_File_iread_at_all
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_all
MPI_File_iwrite_at
MPI_File_iwrite_at_all
MPI_File_iwrite_shared
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin

MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_shared
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_at
MPI_File_write_at_a
MPI_File_write_at_all_beg
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_shared
MPI_Gather
MPI_Gatherv
MPI_Get
MPI_Get_accumulate
MPI_Get_count
MPI_Get_elements
MPI_Get_elements_x
MPI_Graph_neighbors_count
MPI_Iallgather
MPI_Iallgatherv
MPI_Iallreduce
MPI_Ialltoall

MPI_Ialltoallv
MPI_Ialltoallw
MPI_Ibcast
MPI_Ibsend
MPI_Iexscan
MPI_Igather
MPI_Igatherv
MPI_Imrec
MPI_Inei allg
MP
M Iredu
MPI_Iredu scatter
MPI_Ireduce_scatter_block
MPI_Irsend
MPI_Iscan
MPI_Iscatter
MPI_Iscatterv
MPI_Isend
MPI_Issend
MPI_Mrecv
MPI_Neighbor_allgather
MPI_Neighbor_allgatherv
MPI_Neighbor_alltoall

MPI_Neighbor_alltoallv
MPI_Neighbor_alltoallw
MPI_Pack
MPI_Pack_external
MPI_Pack_external_size
MPI_Pack_size
MPI_Put
MPI_Raccumulate
MPI cv
Recv_init
s
duce_local
Reduce scatter
Reduce_scatter_block
MPI get
MPI_Rget_accumulate
MPI_Rput
MPI_Rsend
MPI_Rsend_init
MPI_Scan
MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend

MPI_Ssend_init
MPI_Startall
MPI_Status_set_elements
MPI_Status_set_elements_x
MPI_T_cvar_handle_alloc
MPI_T_pvar_handle_alloc
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Type_contiguous
MPI_Type_create_hindexed
MPI_Type_create_hindexed_block
MPI_Type_create_hvector
MPI_Type_create_indexed_block
MPI_Type_create_struct
MPI_Type_get_extent_x
MPI_Type_get_true_extent_x
MPI_Type_indexed
MPI_Type_size_x
MPI_Type_vector
MPI_Unpack
MPI_Unpack_external
MPI_Waitall
MPI_Waitany
MPI_Waitsome

We already have
a few "_X" functions

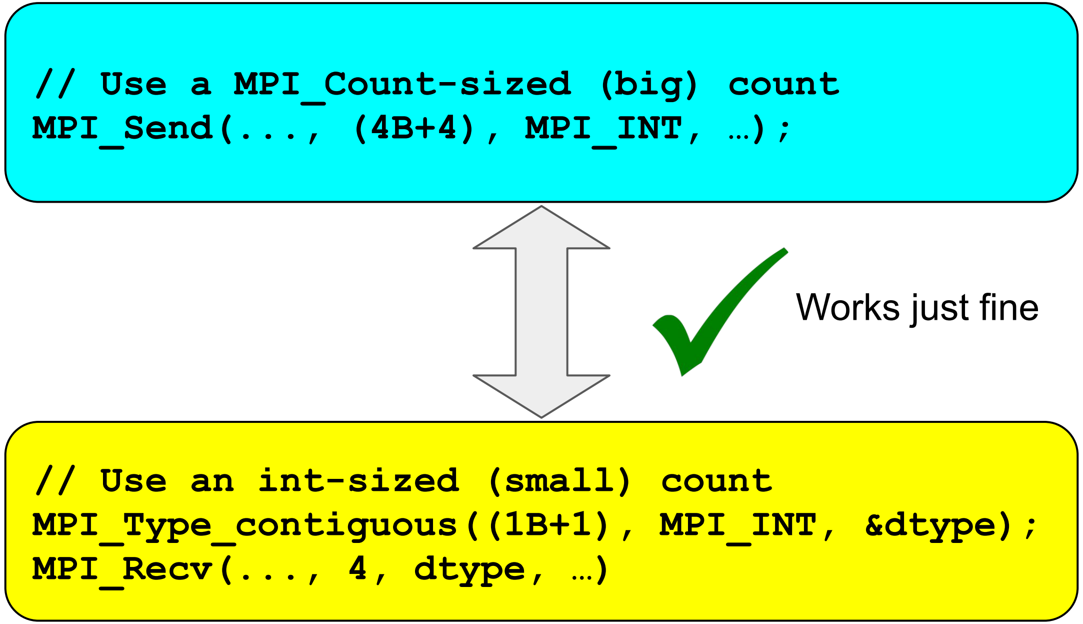
Constraints / Assumptions

1. All “count” arguments in a given API will be either `int` or `MPI_Count`

```
   MPI_Bad(int count1, MPI_Count count2)

      MPI_Good(int count1, int count2)
MPI_Good(MPI_Count count1, MPI_Count count2)
```

Constraints / Assumptions

2. The only thing that matters is the underlying type map



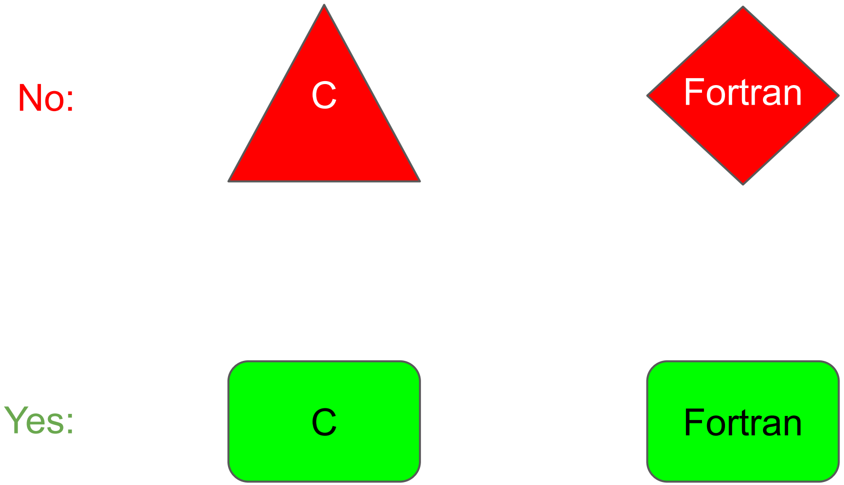
```
// Use a MPI_Count-sized (big) count
MPI_Send(..., (4B+4), MPI_INT, …);
```

Works just fine

```
// Use an int-sized (small) count
MPI_Type_contiguous((1B+1), MPI_INT, &dtype);
MPI_Recv(..., 4, dtype, …)
```

Constraints / Assumptions

3. Must solve for both C and Fortran
    a. Solve it in the “same” way for both languages (from the user perspective)

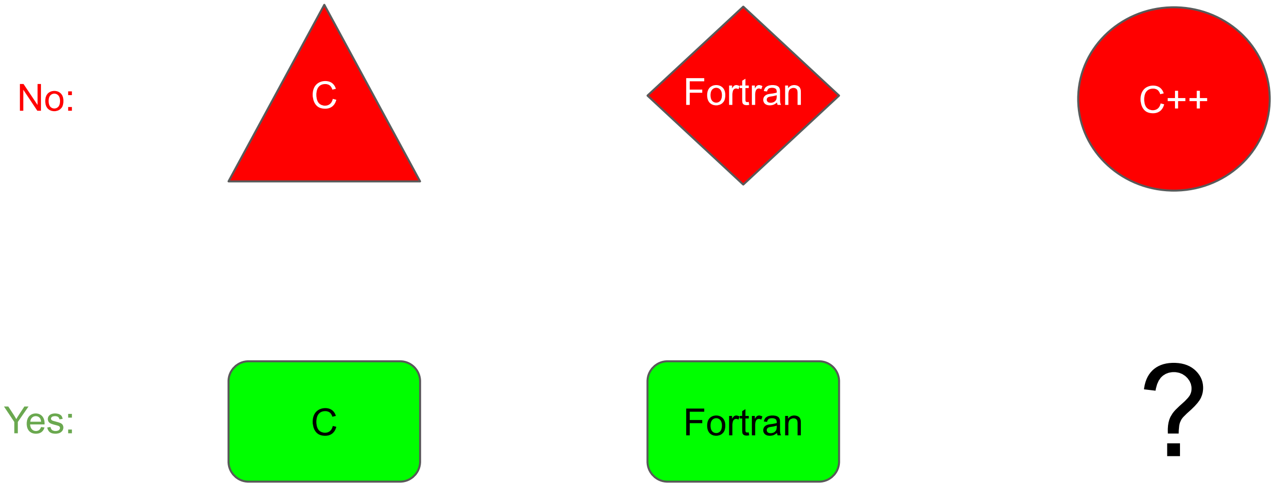


No: C Fortran

Yes: C Fortran

Constraints / Assumptions

4. We need to enable C++ BigCount
    a. Remember: C and C++ are quite different languages at this point
    b. Open question: what does that mean for the C bindings?

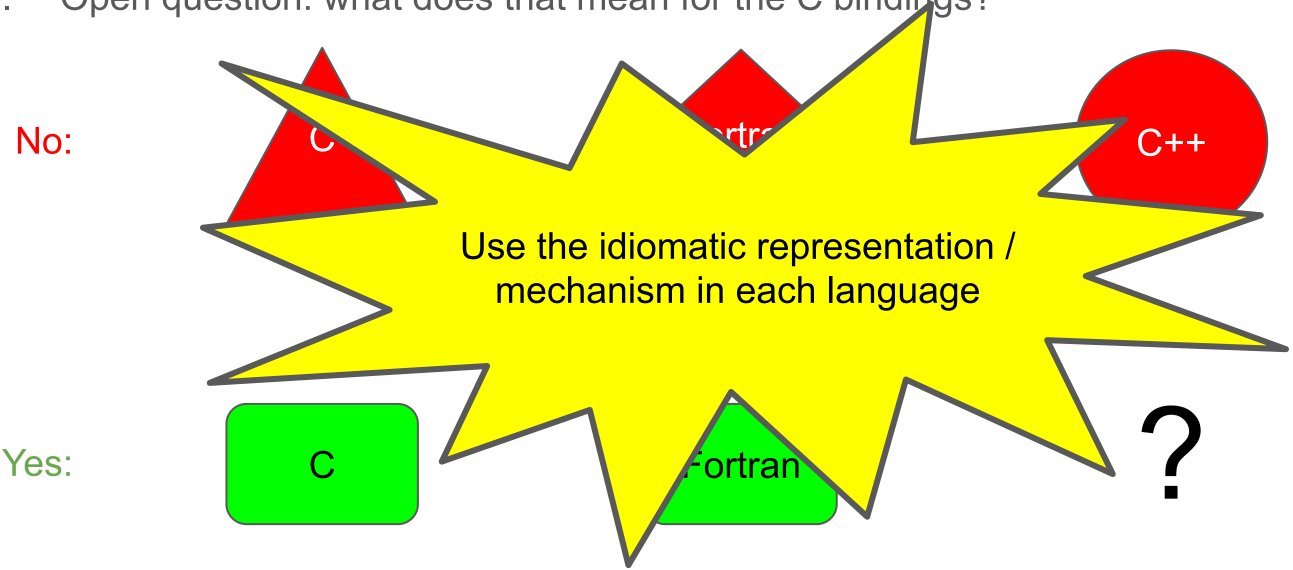


No: C Fortran C++

Yes: C Fortran ?

Constraints / Assumptions

4. We need to enable C++ BigCount
    a. Remember: C and C++ are quite different languages at this point
    b. Open question: what does that mean for the C bindings?



Use the idiomatic representation / mechanism in each language

Constraints / Assumptions

5. May require new(ish) compilers

Five main options

	C	C++	Fortran
1.	No change / keep all counts as `int`		
2.	Change all counts to `MPI_Count`		
3.	“_X” functions	“_X” functions	“_X” functions
4.	Function pointers	Function pointers (?)	Objects
5.	C11 _Generic	Function overloading	Generic functions

1. No changes / keep all counts `int` in MPI-4

Users be like:

2. Change all counts to `MPI_Count`

Users be like:

3. “_X” functions

```
MPI_Send(..., int count, …)
MPI_Send_x(..., MPI_Count count, …)
```

Discussed in prior Forum meetings:

- PRO: Simple
- PRO: Started down this path in MPI-3.0
- CON: Lots of new functions
- CON: Ugly (evil)
- CON: (Very) Bad precedent

4. Function pointers / objects

```
comm.send(..., int count, …)
comm.send(..., MPI_Count count, …)
```

Discussed in Chattanooga / Jan 2019

- PRO: Opens the door for many interesting possibilities (e.g., QMPI, sessions)
- CON: Problems still to be solved, including:
  - Backwards compatibility
  - Incremental app upgrades
- CON: Potentially a *lot* of new text for MPI-4

5. “Polymorphism”

Build upon Jeff Hammond’s prior C11 discussions

Goal: user just calls MPI_Send(...)
with either `int` or `MPI_Count`

5. “Polymorphism”

Turns into three different things:

C:	C11 `_Generic`
C++:	Function overloading
Fortran:	Generic functions

Goal: user just calls MPI_Send(...)
with either `int` or `MPI_Count`

5a. C11 _Generic

- Implementation provides two functions with distinct names
  - Standardize the names for PMPI reasons
- Compiler chooses between them at compile time
- Uses the C11 `_Generic` keyword
  - Does not exist prior to C11
  - Does not exist in C++
- We tell users to still use “main” name (e.g., `MPI_Send`)

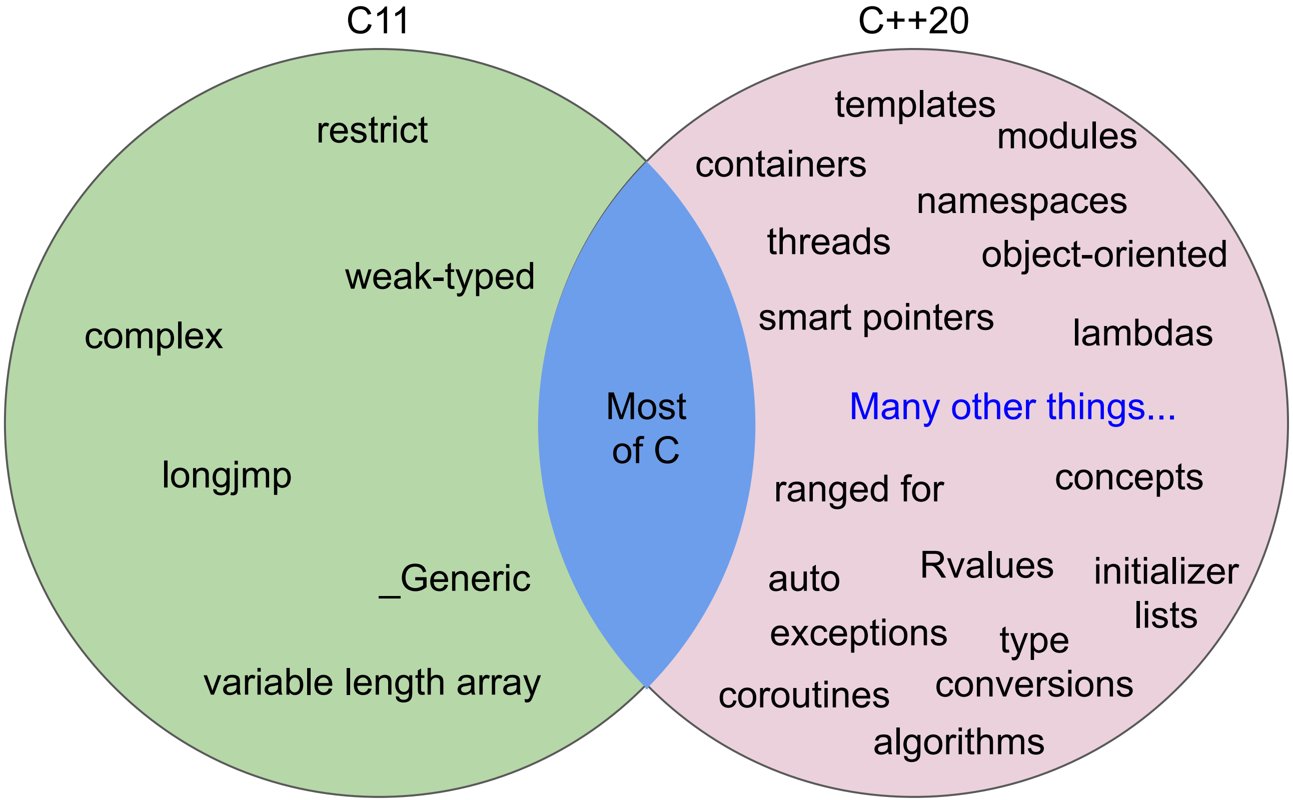
5a. C11 _Generic

- Implementation provides two functions with distinct names
  - Standardize the names for PMPI reasons
- Compiler chooses between them at compile time
- Uses the C11 `_Generic` keyword
  - Does not exist prior to C11 → Pre-C11 implementations get `int`
  - Does not exist in C++
- We tell users to still use “main” name (e.g., `MPI_Send`)

5a. C11 _Generic

- Implementation provides two functions with distinct names
  - Standardize the names for PMPI reasons
- Compiler chooses between them at compile time
- Uses the C11 `_Generic` keyword
  - Does not exist prior to C11
  - Does not exist in C++ → Use a different solution for C++
- We tell users to still use “main” name (e.g., `MPI_Send`)

Sidenote: why is C++ no longer “C with objects”?



C11

restrict

weak-typed

complex

longjmp

_Generic

variable length array

Most of C

C++20

templates

modules

containers

namespaces

threads

object-oriented

smart pointers

lambdas

Many other things...

ranged for

concepts

auto

Rvalues

initializer lists

exceptions

type conversions

coroutines

algorithms

Sidenote: why is C++ no longer “C with objects”?

Let’s not forget things that are spelled the same in C and C++, but are actually different

- Boolean type
- sizeof(struct types)
- Character literal types
- Implicit extern- vs. file-scoping
- inline functions

Dr. Evil Approved™

5a. C11 _Generic: example

```
// mpi.h
typedef long MPI_Count;

#define MPI_Send(buf, count, dt, rank, tag, comm)\
          _Generic(count,                        \
             default: MPI Send int,              \
                 int: MPI_Send_int,              \
           MPI Count: MPI_Send_count             \
          )(buf, count, dt, rank, tag, comm)

int MPI_Send_int(..., int count, ...)
{
    return MPI Send count(...,
       (MPI_Count)count, ...);
}

int MPI_Send_count(..., MPI_Count count, ...)
{
    …
}
```

5a. C11 _Generic: example

```
// mpi.h
typedef long MPI_Count;

#define MPI_Send(buf, count, dt, rank, tag, comm)\
          _Generic(count,                        \
             default: MPI Send int,              \
                 int: MPI_Send_int,              \
           MPI Count: MPI_Send_count             \
          )(buf, count, dt, rank, tag, comm)

int MPI_Send_int(..., int count, ...)
{
    return MPI Send count(...,
       (MPI_Count)count, ...);
}

int MPI_Send_count(..., MPI_Count count, ...)
{
    …
}
```

```
#include <mpi.h>

int main(int argc, char **argv)
{
    …

    int i = 32;
    MPI Send(..., i, ...);
    MPI Send(..., 32, ...);
    MPI_Send(..., (MPI_Count)32, ...);

    …

    MPI_Count bigI = 8589934592;
    MPI Send(..., bigI, ...);
    MPI_Send(..., 8589934592, ...);

    …
}
```

5b. C++ function overloading

- C++ does not have the **`_Generic`** keyword
- MPI implementation provides two C++ functions with the same name
- PMPI interfaces will need to be compiled with same compiler
- `mpi.h` must distinguish between C and C++

5b. C++ function overloading

- C++ does not have the **`_Generic`** keyword
- MPI implementation provides two C++ functions with the same name
- PMPI interfaces will need to be compiled with same compiler
- `mpi.h` must distinguish between C and C++ → ...unless we make **`<mpi.hpp>`**

5b. C++ function overloading: example

```
// mpi.hpp

int MPI_Send(..., int count, ...)
{
    return MPI Send(...,
        static_cast<long>(count), ...);
}

int MPI_Send(..., MPI_Count count, ...)
{
    ...
}
```

5b. C++ function overloading: example

```
// mpi.hpp

int MPI_Send(..., int count, ...)
{
    return MPI Send(...,
        static_cast<long>(count), ...);
}

int MPI_Send(..., MPI_Count count, ...)
{
    ...
}
```

```
#include <mpi.hpp>

int main(int argc, char **argv)
{
    …

    int i = 32;
    MPI Send(..., i, ...);
    MPI Send(..., 32, ...);
    MPI_Send(..., (MPI_Count)32, ...);

    …

    MPI_Count bigI = 8589934592;
    MPI Send(..., bigI, ...);
    MPI_Send(..., 8589934592, ...);

    …
}
```

5a+5b. Note that user code is 99.9% the same

```
#include <mpi.h>

int main(int argc, char **argv)
{
    …

    int i = 32;
    MPI Send(..., i, ...);
    MPI Send(..., 32, ...);
    MPI_Send(..., (MPI_Count)32, ...);

    …

    MPI_Count bigI = 8589934592;
    MPI Send(..., bigI, ...);
    MPI_Send(..., 8589934592, ...);

    …
}
```

```
#include <mpi.hpp>

int main(int argc, char **argv)
{
    …

    int i = 32;
    MPI Send(..., i, ...);
    MPI Send(..., 32, ...);
    MPI_Send(..., (MPI_Count)32, ...);

    …

    MPI_Count bigI = 8589934592;
    MPI Send(..., bigI, ...);
    MPI_Send(..., 8589934592, ...);

    …
}
```

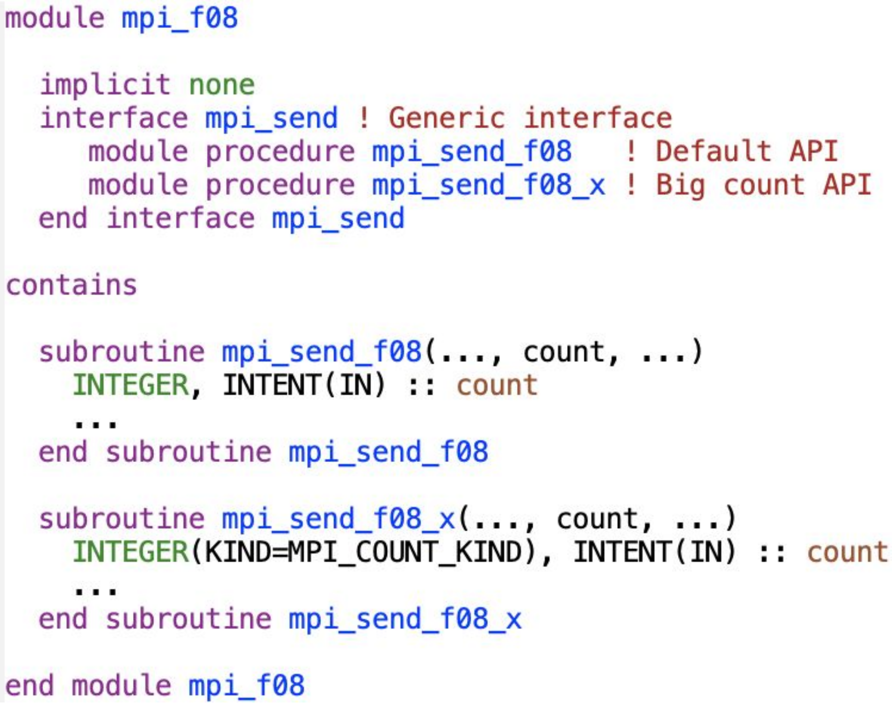
The only difference

5c. Fortran generic functions

- Implementation provides a single interface with two subroutines of distinct names
  - Names standardized for PMPI reasons
- Only do this for mpi_f08
  - Not the mpi module or mpif.h

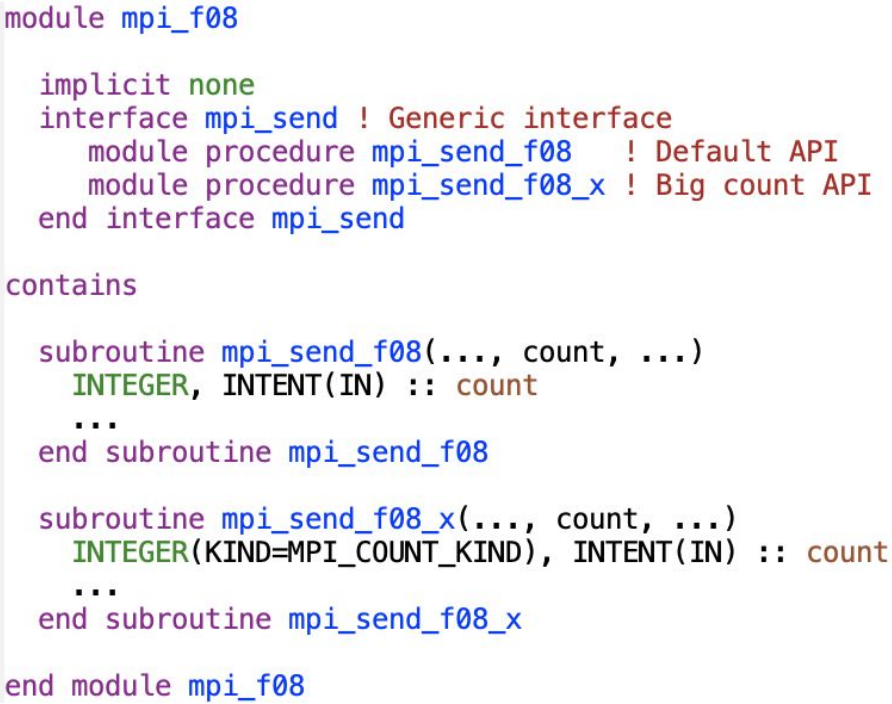
F08

5c. Fortran generic functions: example

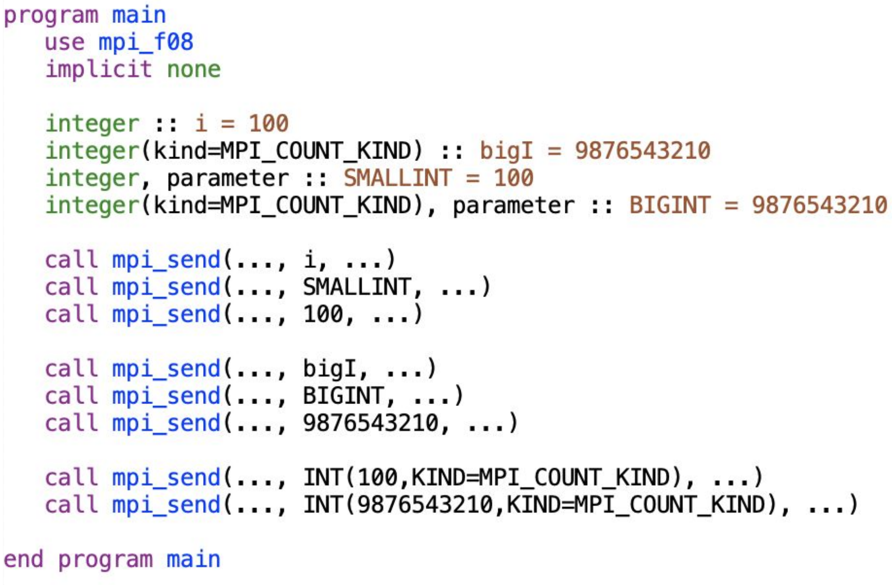


```fortran
module mpi_f08

  implicit none
  interface mpi_send ! Generic interface
     module procedure mpi_send_f08   ! Default API
     module procedure mpi_send_f08_x ! Big count API
  end interface mpi_send

contains

  subroutine mpi_send_f08(..., count, ...)
    INTEGER, INTENT(IN) :: count
    ...
  end subroutine mpi_send_f08

  subroutine mpi_send_f08_x(..., count, ...)
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    ...
  end subroutine mpi_send_f08_x

end module mpi_f08
```

5c. Fortran generic functions: example



```fortran
module mpi_f08

  implicit none
  interface mpi_send ! Generic interface
     module procedure mpi_send_f08   ! Default API
     module procedure mpi_send_f08_x ! Big count API
  end interface mpi_send

contains

  subroutine mpi_send_f08(..., count, ...)
    INTEGER, INTENT(IN) :: count
    ...
  end subroutine mpi_send_f08

  subroutine mpi_send_f08_x(..., count, ...)
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    ...
  end subroutine mpi_send_f08_x

end module mpi_f08
```



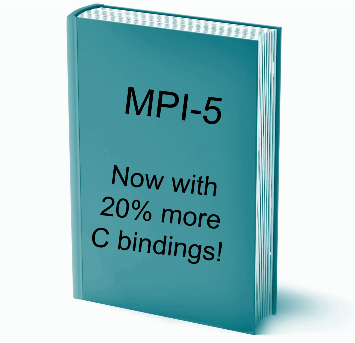
```fortran
program main
   use mpi_f08
   implicit none

   integer :: i = 100
   integer(kind=MPI_COUNT_KIND) :: bigI = 9876543210
   integer, parameter :: SMALLINT = 100
   integer(kind=MPI_COUNT_KIND), parameter :: BIGINT = 9876543210

   call mpi_send(..., i, ...)
   call mpi_send(..., SMALLINT, ...)
   call mpi_send(..., 100, ...)

   call mpi_send(..., bigI, ...)
   call mpi_send(..., BIGINT, ...)
   call mpi_send(..., 9876543210, ...)

   call mpi_send(..., INT(100,KIND=MPI_COUNT_KIND), ...)
   call mpi_send(..., INT(9876543210,KIND=MPI_COUNT_KIND), ...)

end program main
```

There is a long-term goal beyond MPI-4

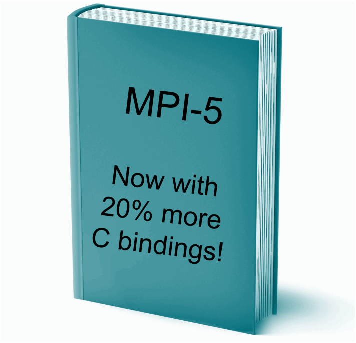
Fixing BigCount via `_Generic` / function overloading is a good first step

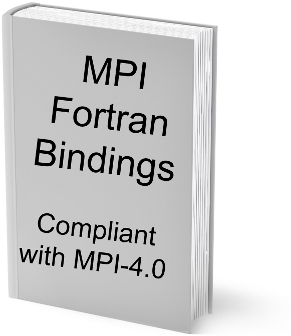
But we envision a future with MORE

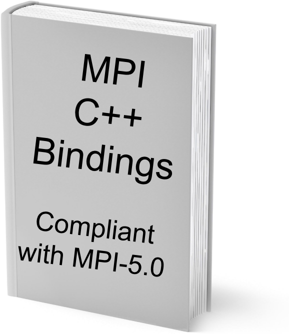
Perhaps something like this:

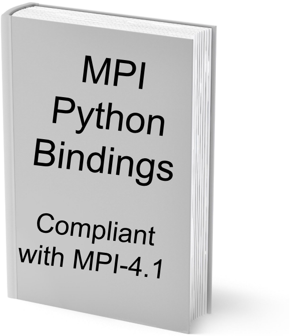


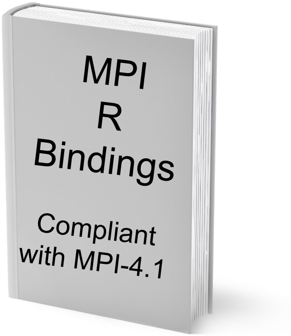
Perhaps something like this:



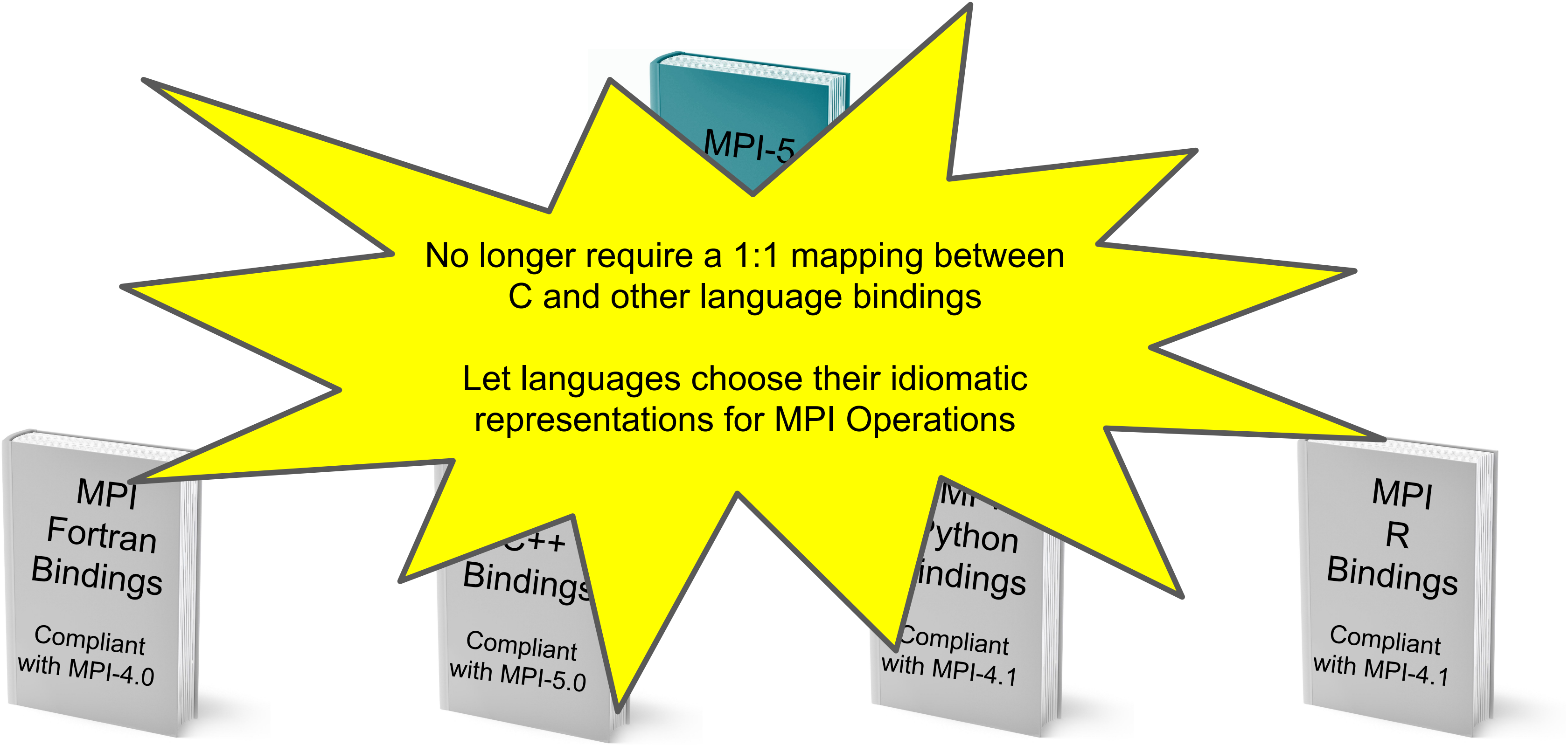








Perhaps something like this:



No longer require a 1:1 mapping between C and other language bindings

Let languages choose their idiomatic representations for MPI Operations

More cool, color-coded books to publish!

Which option does the Forum want us to pursue?

	C	C++	Fortran
1.	No change / keep all counts as `int`		
2.	Change all counts to `MPI_Count`		
3.	“_X” functions	“_X” functions	“_X” functions
4.	Function pointers	Function pointers (?)	Objects
5.	C11 _Generic	Function overloading	Generic functions
6.	Do nothing	Function overloading	Generic functions

Straw poll results

Which way should we go?

- 0: Do nothing / leave int
- 1: Change everything to MPI_Count
- 0: _X functions
- 0: Chattanooga-style function pointers
- 21: (C11 _Generic, C++, Fortran)
- 1: only (C++, Fortran)

Should we add MPI_Count to the “mpi” module or not?

- Yes: 4
- No: 10
- Abstain: 6

Should we make <mpi.hpp> C++ header file?

...we forgot to poll this.

Just in case you wanted to know the differences

	C	C++	Fortran
Mechanism	#define _Generic	overloading, templates	interface
Naming	manual	automagic	manual
Time	compile	compile	compile
Switch	expressions	functions/classes	functions/subroutines
Default	yes	no	no
Glue	manual	automagic	manual
Runtime	no	runtime polymorphism (virtual functions)	no