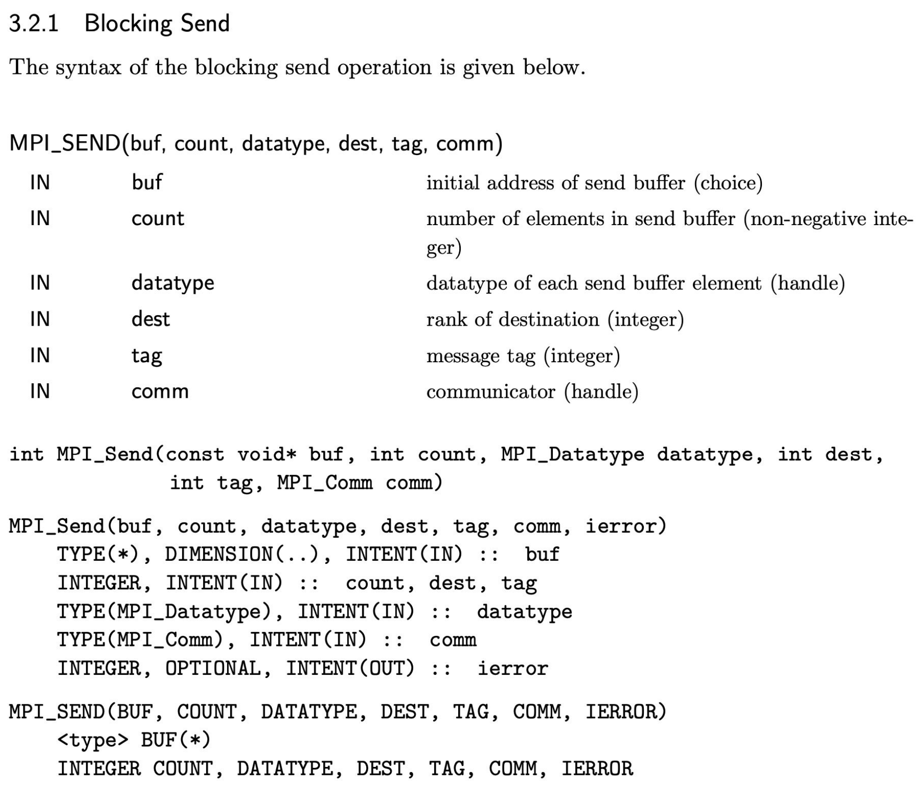
MPI LaTeX Pythonization

MPI Forum Meeting
December 9, 2019

What is this?

(the short version)

From this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

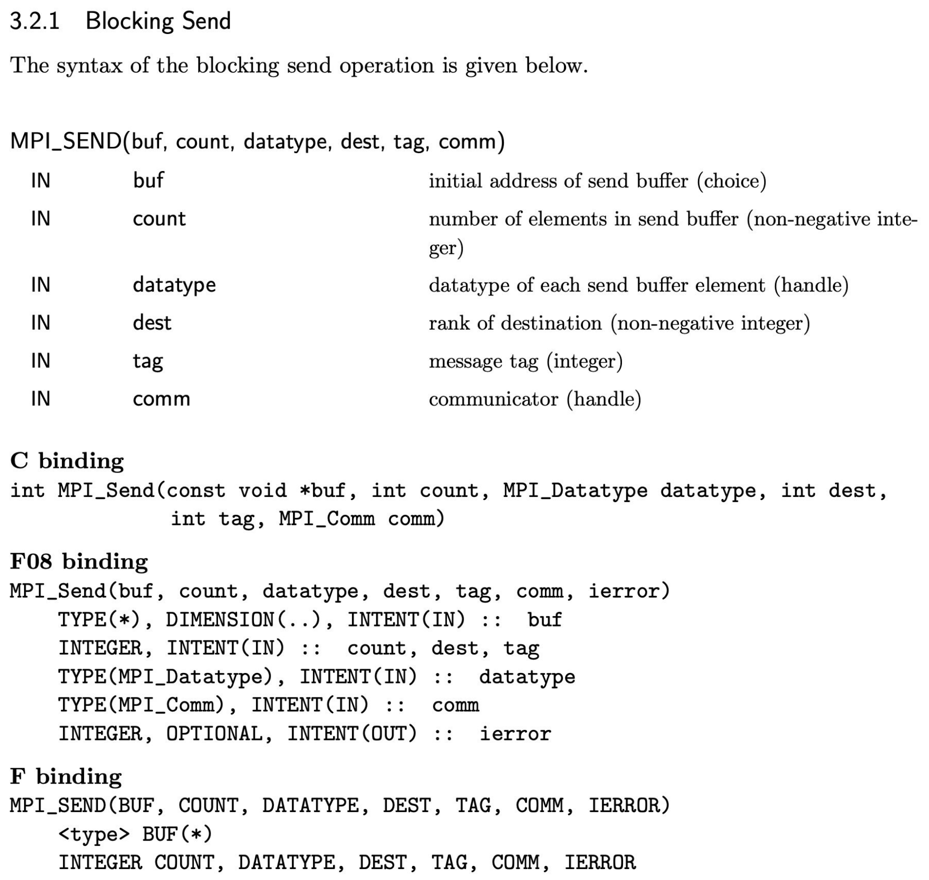
IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

To this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

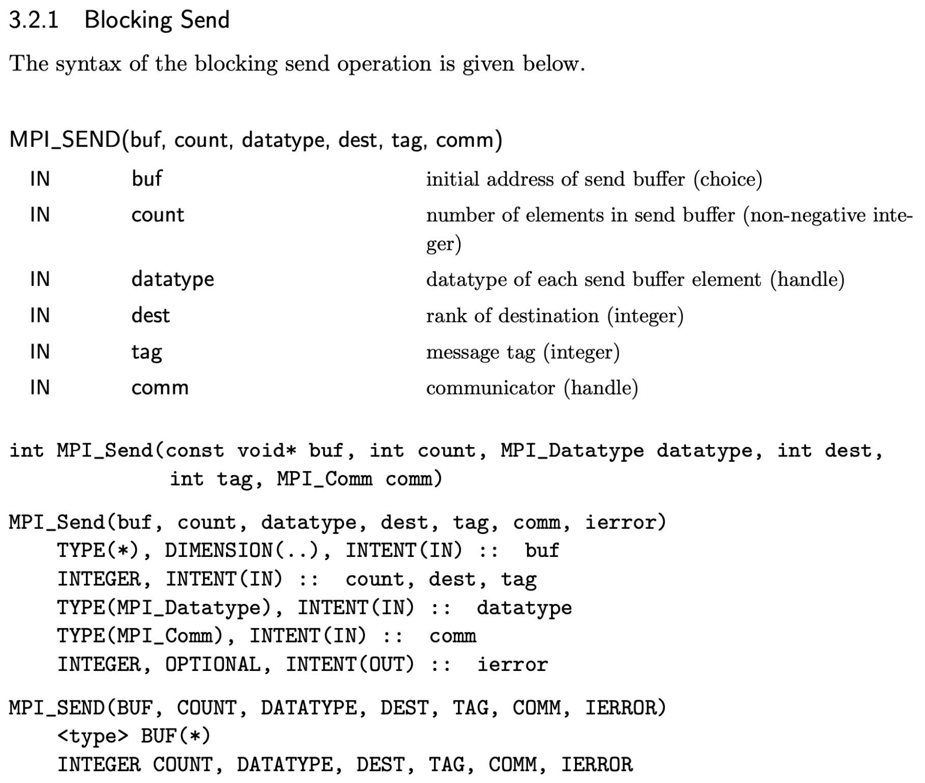
F08 binding

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

From this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

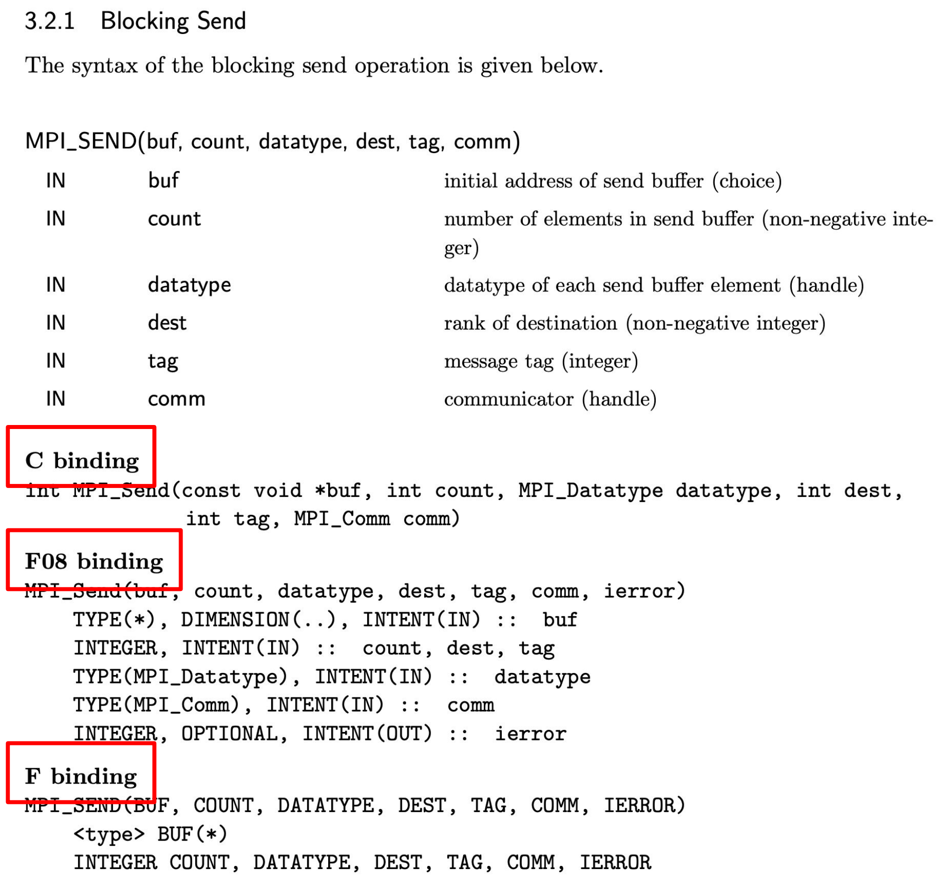
IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

To this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

F08 binding

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Ummm….

Aside from the titles, what's the difference?

From this:

```
\begin{funcdef}{MPI\_SEND(buf, count, datatype, dest, tag, comm)}
\funcarg{\IN}{buf}{initial address of send buffer (choice)}
\funcarg{\IN}{count}{number of elements in send buffer (non-negative integer)}
\funcarg{\IN}{datatype}{datatype of each send buffer element (handle)}
\funcarg{\IN}{dest}{rank of destination (integer)}
\funcarg{\IN}{tag}{message tag (integer)}
\funcarg{\IN}{comm}{communicator (handle)}
\end{funcdef}

\cdeclmainindex{MPI\_Comm}%
\mpibind{MPI\_Send(const~void*~buf, int~count, MPI\_Datatype~datatype, int~dest, int~tag, MPI\_Comm~comm)}

\mpifnewbind{MPI\_Send(buf, count, datatype, dest, tag, comm, ierror) \fargs TYPE(*), DIMENSION(..),
INTENT(IN) :: buf \\ INTEGER, INTENT(IN) :: count, dest, tag \\ TYPE(MPI\_Datatype), INTENT(IN) :: datatype \\
TYPE(MPI\_Comm), INTENT(IN) :: comm \\ INTEGER, OPTIONAL, INTENT(OUT) :: ierror}
\mpifbind{MPI\_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)\fargs <type> BUF(*) \\ INTEGER  COUNT,
DATATYPE, DEST, TAG, COMM, IERROR}
\mpicppemptybind{MPI::Comm::Send(const void*~buf, int~count, const MPI::Datatype\&~datatype, int~dest,
int~tag) const}{void}
```

To this:

```
\begin{mpi-binding}
    function_name('MPI_Send')

    parameter('buf', 'BUFFER', desc='initial address of send buffer', constant=True)
    parameter('count', 'POLYXFER_NUM_ELEM', desc='number of elements in send buffer')
    parameter('datatype', 'DATATYPE', desc='datatype of each send buffer element')
    parameter('dest', 'RANK', desc='rank of destination')
    parameter('tag', 'TAG', desc='message tag')
    parameter('comm', 'COMMUNICATOR')
\end{mpi-binding}
```

To this:

```
\begin{mpi-binding}
    function_name('MPI_Send')

    parameter('buf', 'BUFFER', desc='initial address of send buffer', constant=True)
    parameter('count', 'POLYXFER_NUM_ELEM', desc='number of elements in send buffer')
    parameter('datatype', 'DATATYPE', desc='datatype of each send buffer element')
    parameter('dest', 'RANK', desc='rank of destination')
    parameter('tag', 'TAG', desc='message tag')
    parameter('comm', 'COMMUNICATOR')
\end{mpi-binding}
```

Inside the "mpi-binding" block is Python code

Why bother?

Immediate benefits

- Only type the bindings once
  - Versus typing them 5x (language-independent, C, F08, and F90 [and C++!])
  - This ***significantly*** improves the lives of chapter authors
- The Python is rendered into the appropriate LaTeX for:
  - Language Independent Specification (LIS) binding
  - C binding
  - Fortran ‘90 binding
  - Fortran ‘08 binding

Future benefits

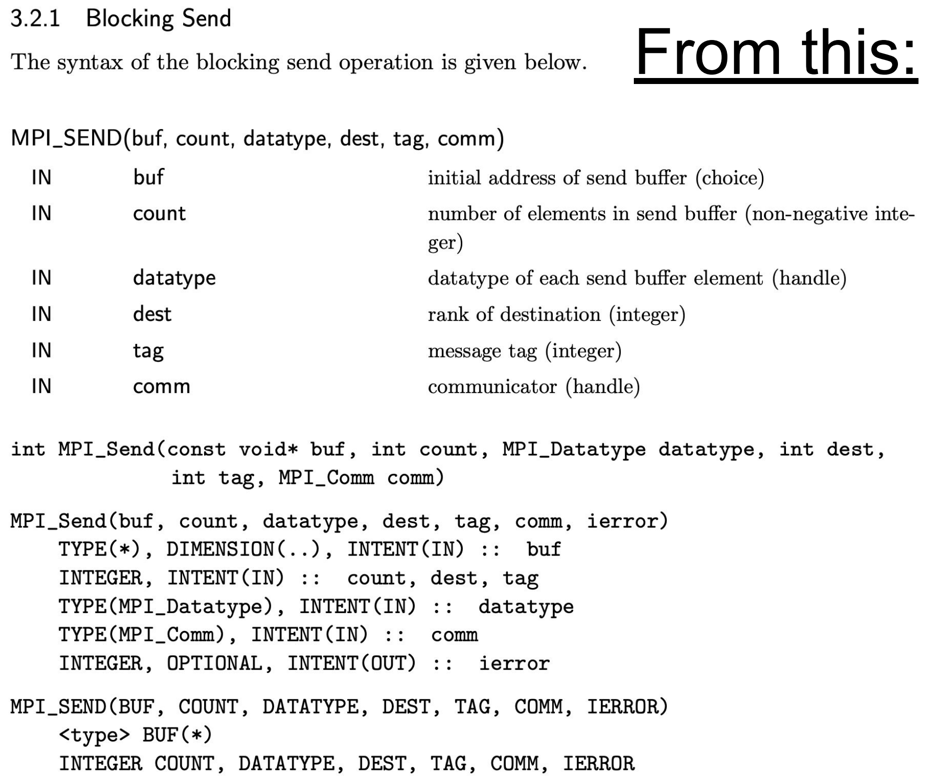
- Programatically compare versions of the MPI standard
- Can make global changes to rendering style
- Generate machine-parsable listing of all MPI routines
- Generate reference C, Fortran ‘90, Fortran ‘08 listings of all MPI routines
    - I.e., most of mpi.h, mpif.h, mpi module, mpi_f08 module
    - Can also make machine-parsable versions (e.g., JSON)
- Continuous integration for LaTeX Github pull requests
    - Specifically call out changes to bindings to help prevent mistakes

Future benefits

- Enable The Embiggenment™ (MPI "ExaCount") with only minor code changes
- Specifically: render multiple bindings of the same MPI routine:
    - `MPI_Send(...int...)`
    - `MPI_Send_l(...MPI_Count...)`

```
    //   ^^ That's a lower case "L", for "large"
```

From this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

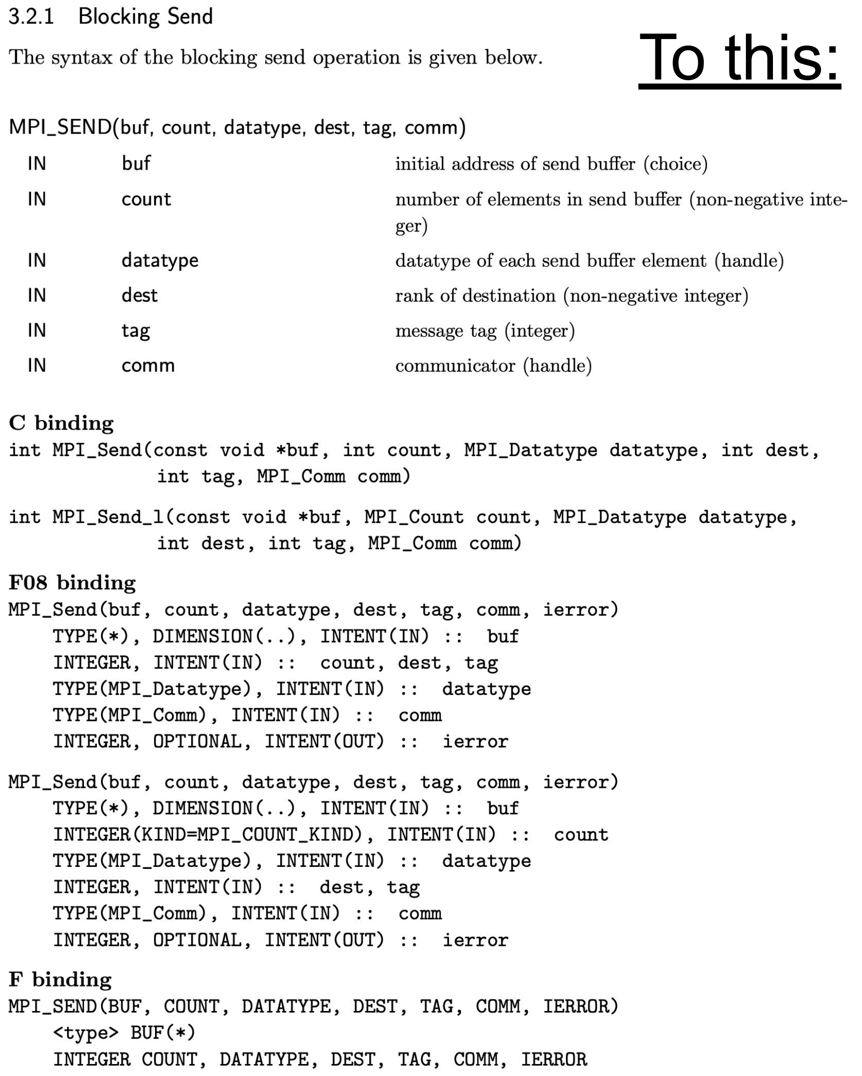
IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

To this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
int MPI_Send_l(const void *buf, MPI_Count count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

F08 binding

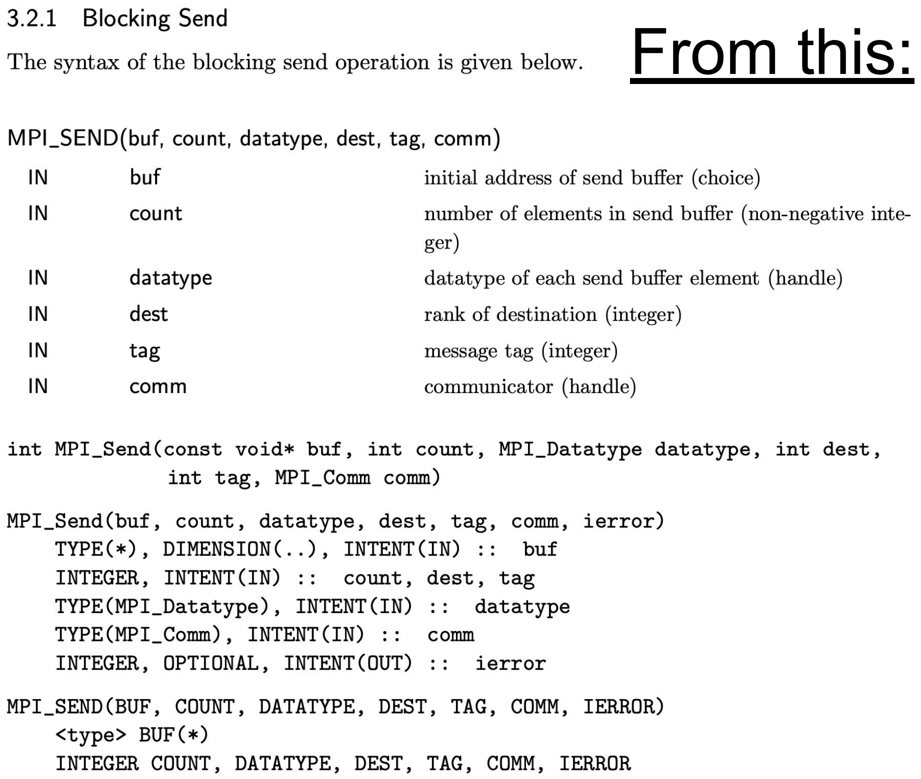
```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  dest, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

From this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

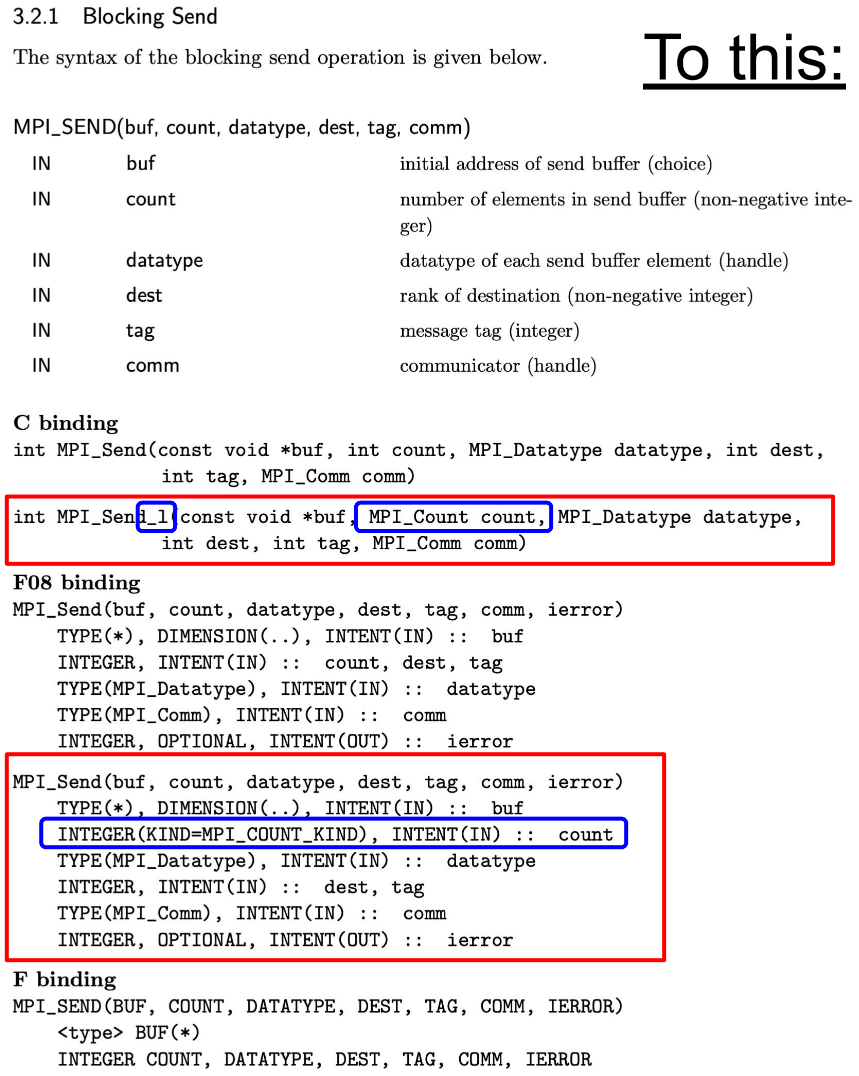
IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

To this:



3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
int MPI_Send_l(const void *buf, MPI_Count count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

F08 binding

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  dest, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Chapter authors be like

- Wiki with instructions for chapter authors
  - Including tips for those who don’t know Python

This is *only* for MPI-4.0 and forward

Multi-step process

1. Pythonize the LaTeX for the existing MPI routines
   a. Make it (mostly**) comparable to MPI-3.1
2. Embiggen
   a. The BigCount / LargeCount / ExaCount / WhateverCount issue
   b. Modify the Python rendering code to emit multiple bindings when relevant

** There are some minor differences (described later)

Multi-step process

1. Pythonize the LaTeX for the existing MPI routines
    a. Make it (mostly**) comparable to MPI-3.1
2. Embiggen
    a. The BigCount / LargeCount / ExaCount / WhateverCount issue
    b. Modify the Python rendering code to emit multiple bindings when relevant

we're only talking about step 1 right now

** There are some minor differences (described later)

CHAPTER AUTHORS
and MPI-4 PR AUTHORS

PAY ATTENTION

(stop reading your email)

Simple example: MPI_SEND

```
\begin{mpi-binding}
    function_name('MPI_Send')

    parameter('buf', 'BUFFER', desc='initial address of send buffer', constant=True)
    parameter('count', 'POLYXFER_NUM_ELEM', desc='number of elements in send buffer')
    parameter('datatype', 'DATATYPE', desc='datatype of each send buffer element')
    parameter('dest', 'RANK', desc='rank of destination')
    parameter('tag', 'TAG', desc='message tag')
    parameter('comm', 'COMMUNICATOR')
\end{mpi-binding}
```

Simple example: MPI_SEND

```
\begin{mpi-binding}
    function_name('MPI_Send')

    parameter('buf',      'BUFFER',            desc='initial address of send buffer',      constant=True)
    parameter('count',    'POLYXFER_NUM_ELEM', desc='number of elements in send buffer')
    parameter('datatype', 'DATATYPE',          desc='datatype of each send buffer element')
    parameter('dest',     'RANK',              desc='rank of destination')
    parameter('tag',      'TAG',               desc='message tag')
    parameter('comm',     'COMMUNICATOR')
\end{mpi-binding}
```

Things to note:

1. One call to function_name(), a bunch of calls to parameter()
2. Two mandatory params to parameter():
   - MPI routine parameter name
   - MPI routine parameter kind
3. *Most* parameter() calls have “desc” params
   - If you don’t specify it, it simply echos the kind

Simple example: MPI_SEND

```
\begin{mpi-binding}
    function_name('MPI_Send')

    parameter('buf',      'BUFFER',                desc='initial address of send buffer',      constant=True)
    parameter('count',    'POLYXFER_NUM_ELEM', desc='number of elements in send buffer')
    parameter('datatype', 'DATATYPE',              desc='datatype of each send buffer element')
    parameter('dest',     'RANK',                  desc='rank of destination')
    parameter('tag',      'TAG',                   desc='message tag')
    parameter('comm',     'COMMUNICATOR')
\end{mpi-binding}
```

Things to note:

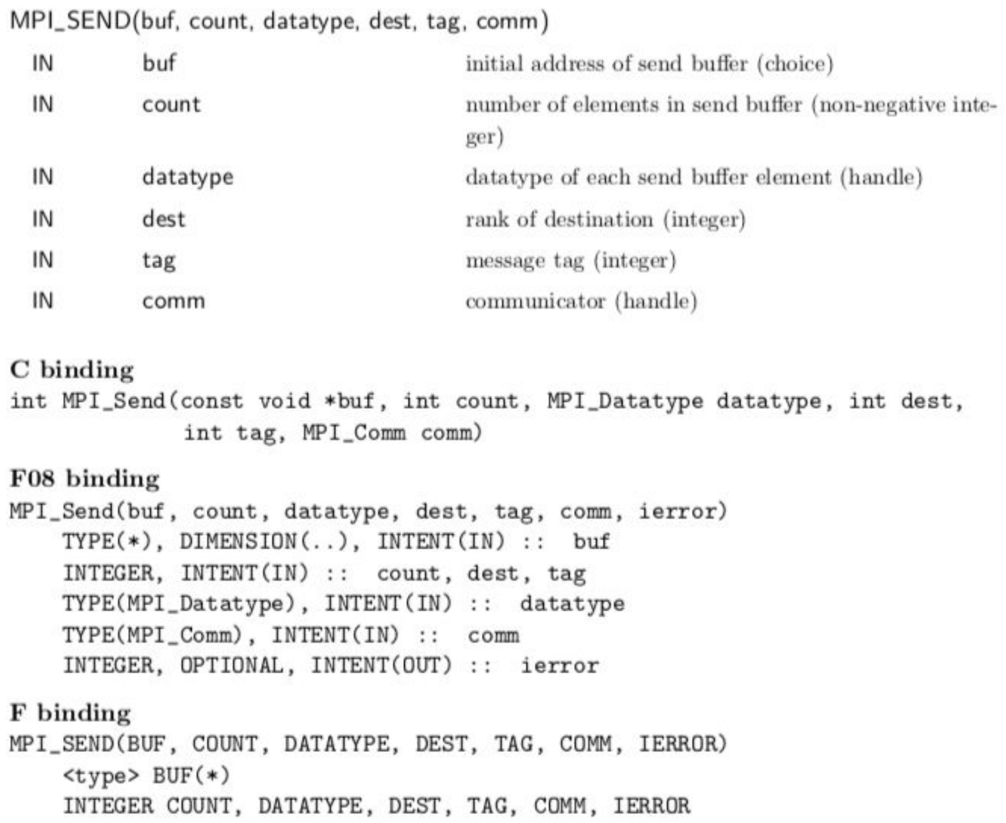
1. One call to function_name(), a bunch of calls t
2. Two mandatory params to parameter():
   - MPI routine parameter name
   - MPI routine parameter kind
3. *Most* parameter() calls have "desc" params
   - If you don't specify it, it simply echos the

There is a lengthly list of MPI parameter "kinds".

It is still evolving.

Link to full reference guide.

Simple example: MPI_SEND renders to this PDF



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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

F08 binding

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

More detailed example: MPI_TYPE_INDEXED

```
\begin{mpi-binding}
  function_name('MPI_Type_indexed')

  parameter('count', 'ARRAY_LENGTH',
            desc='number of blocks -- also number of entries in '
                 '\mpiarg{array_of_displacements} and \mpiarg{array_of_blocklengths}')
  parameter('array_of_blocklengths', 'BLOCKLENGTH', length='count', constant=True,
            desc='number of elements per block')

  # Note that this param is POLYDISPLACEMENT.
  # MPI-3.1 defined this param as int.
  parameter('array_of_displacements', 'POLYDISPLACEMENT', length='count', constant=True,
            desc='displacement for each block, in multiples of \mpiarg{oldtype}')
  parameter('oldtype', 'DATATYPE', desc='old datatype')
  parameter('newtype', 'DATATYPE', direction='out', desc='new datatype')
\end{mpi-binding}
```

More detailed example: MPI_TYPE_INDEXED

```
\begin{mpi-binding}
  function_name('MPI_Type_indexed')

  parameter('count', 'ARRAY_LENGTH',
            desc='number of blocks -- also number of entries in '
                 '\mpiarg{array_of_displacements} and \mpiarg{array_of_blocklengths}')
  parameter('array_of_blocklengths', 'BLOCKLENGTH', length='count', constant=True,
            desc='number of elements per block')

  # Note that this param is POLYDISPLACEMENT.
  # MPI-3.1 defined this param as int.
  parameter('array_of_displacements', 'POLYDISPLACEMENT', length='count', constant=True,
            desc='displacement for each block, in multiples of \mpiarg{oldtype}')
  parameter('oldtype', 'DATATYPE', desc='old datatype')
  parameter('newtype', 'DATATYPE', direction='out', desc='new datatype')
\end{mpi-binding}
```

Python catenates strings together

Python-style comments

“length” to specify array lengths

“constant” to specify constant params

“direction” to specify OUT (and INOUT) params

More detailed example: MPI_TYPE_INDEXED

```
\begin{mpi-binding}
  function_name('MPI_Type_indexed')

  parameter('count', 'ARRAY_LENGTH',
            desc='number of blocks -- also number of entries in '
                 '\mpiarg{array_of_displacements} and \mpiarg{array_of_blocklengths}')
  parameter('array_of_blocklengths', 'BLOCKLENGTH', length='count', constant=True,
            desc='number of elements per block')

  # Note that this param is POLYDISPLACEMENT.
  # MPI-3.1 defined this param as int.
  parameter('array_of_displacements', 'POLYDISPLACEMENT', length='count', constant=True,
            desc='displacement for each block, in multiples of \mpiarg{oldtype}')
  parameter('oldtype', 'DATATYPE', desc='old datatype')
  parameter('newtype', 'DATATYPE', direction='out', desc='new datatype')
\end{mpi-binding}
```

No need to escape "_"

More detailed example: MPI_TYPE_INDEXED

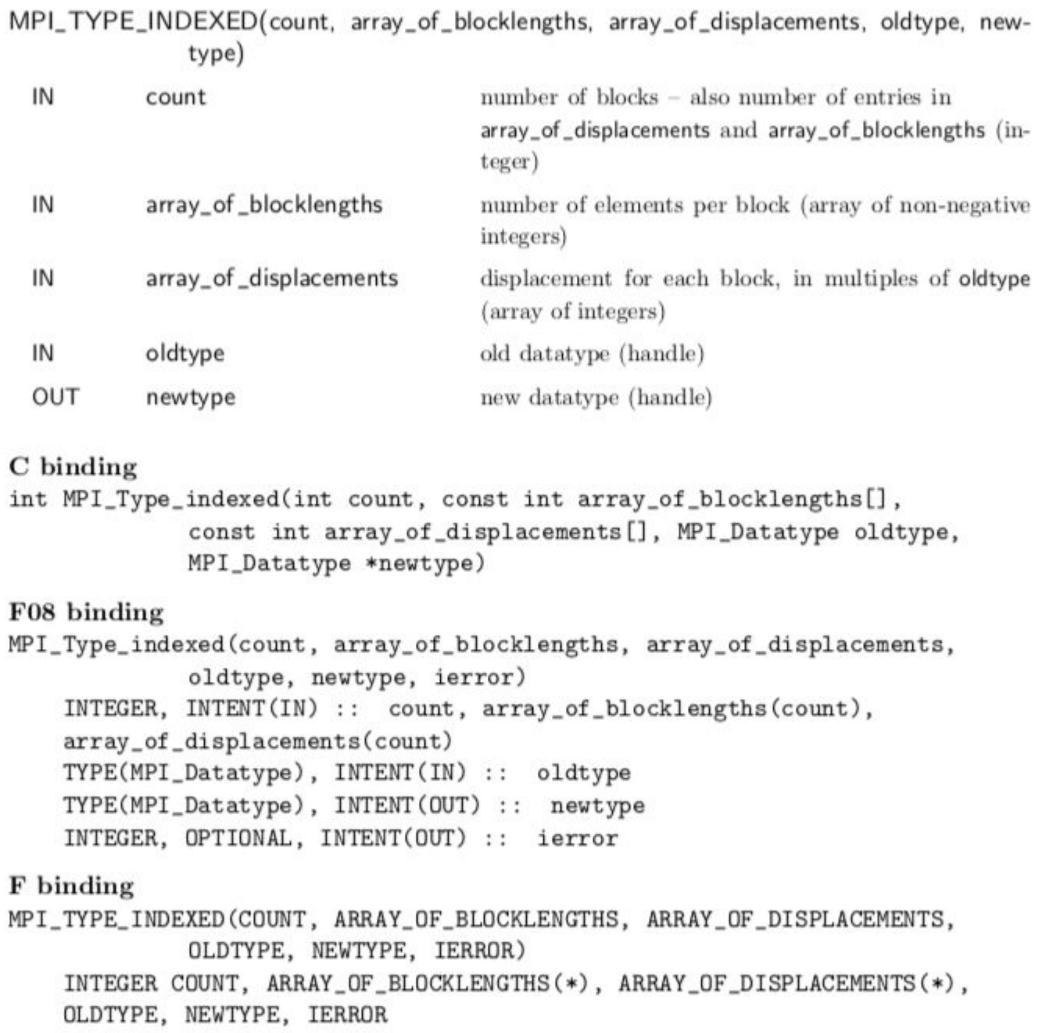
```
\begin{mpi-binding}
  function_name('MPI_Type_indexed')

  parameter('count', 'ARRAY_LENGTH',
            desc='number of blocks -- also number of entries in '
                 '\mpiarg{array_of_displacements} and \mpiarg{array_of_blocklengths}')
  parameter('array_of_blocklengths', 'BLOCKLENGTH', length='count', constant=True,
            desc='number of elements per block')

  # Note that this param is POLYDISPLACEMENT.
  # MPI-3.1 defined this param as int.
  parameter('array_of_displacements', 'POLYDISPLACEMENT', length='count', constant=True,
            desc='displacement for each block, in multiples of \mpiarg{oldtype}')
  parameter('oldtype', 'DATATYPE', desc='old datatype')
  parameter('newtype', 'DATATYPE', direction='out', desc='new datatype')
\end{mpi-binding}
```

Still use LaTeX where relevant

More detailed example rendering



MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (integer)
IN	array_of_blocklengths	number of elements per block (array of non-negative integers)
IN	array_of_displacements	displacement for each block, in multiples of oldtype (array of integers)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

C binding

```
int MPI_Type_indexed(int count, const int array_of_blocklengths[],
             const int array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

F08 binding

```
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(count),
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

Why are we boring the entire Forum with this?

This is essentially a “ticket 0”

(compared to MPI-3.1)

BUT THERE ARE DIFFERENCES

And therefore we need a vote

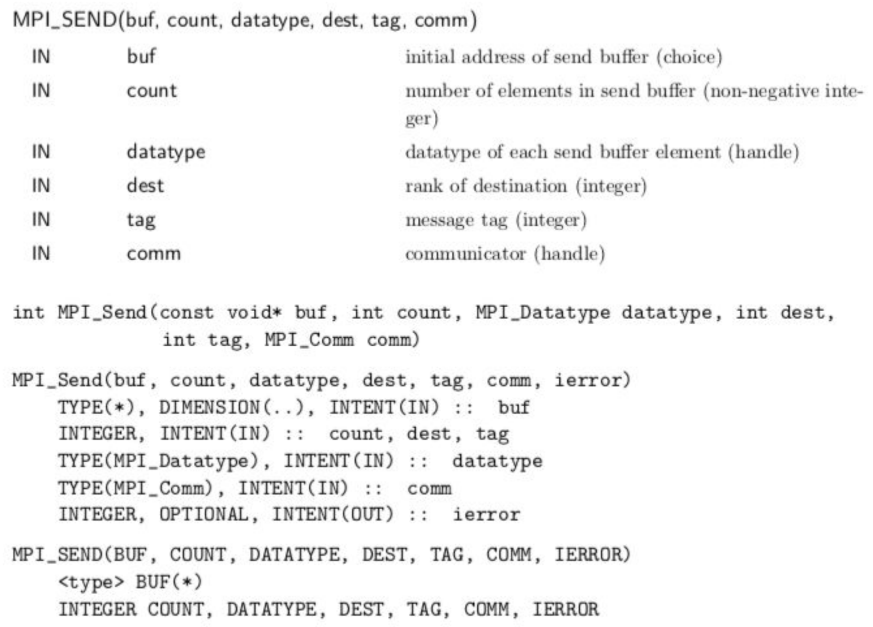
(stay tuned for the full plan)

Rendering differences (“RenDiff”) compared to MPI-3.1

- New 4.0 bindings are not included in MPI-3.1 (obviously)
- Minor editorial changes made by Bill
  - And other ticket 0 changes that have been made over time
- (Hopefully) Non-contentious changes:
  - New sub-heading for each binding
    - C binding
    - F08 binding
    - F binding
  - Humans were not consistent in MPI-3.1
    - Sometimes in C bindings, * is on the left; sometimes it is on the right
    - Sometimes the listing order/grouping of Fortran parameters is… arbitrary
    - LIS descriptions are wildly inconsistent
  - Indenting on 2nd line of a group of Fortran params
    - Bill is working on this
  - Hyphentization at end of lines

RenDiff: Binding sub-headings

MPI-3.1



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

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

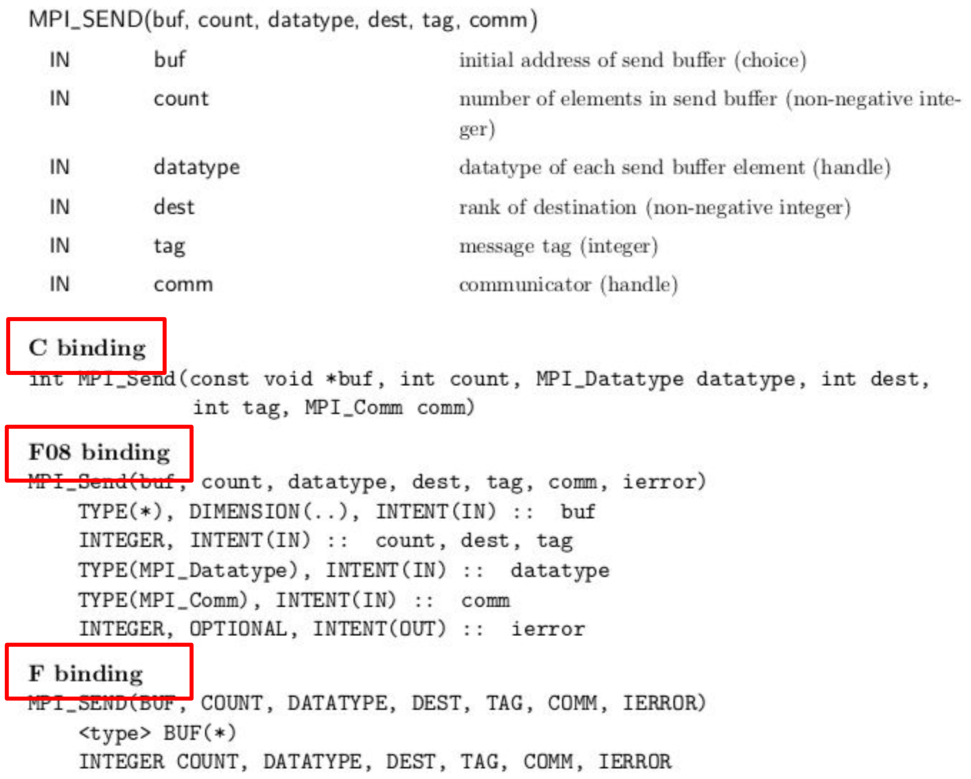
```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

This is in anticipation for The Embiggening™ where we may (will) have multiple bindings for a given language

Pythonized



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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

F08 binding

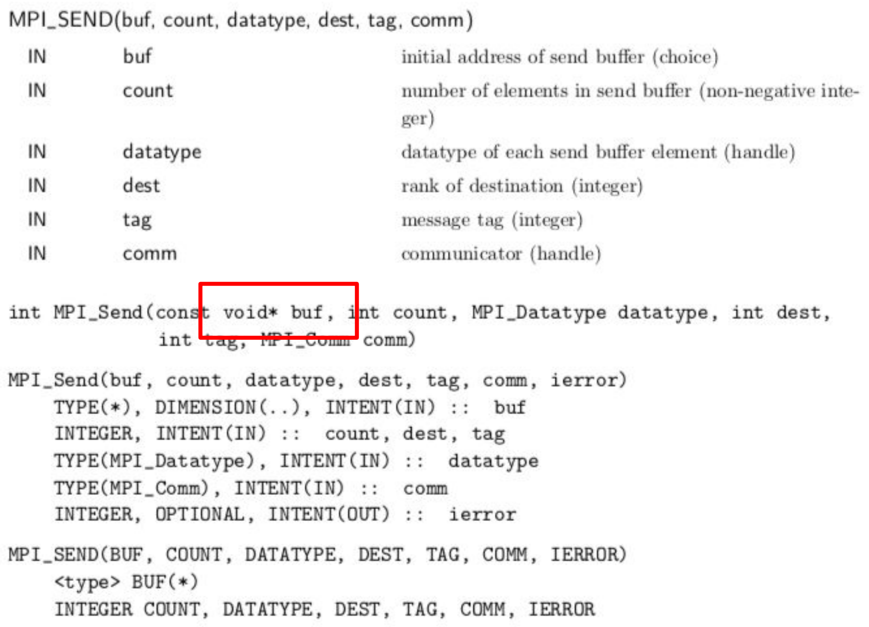
```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

RenDiff: C bindings * placement

MPI-3.1



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

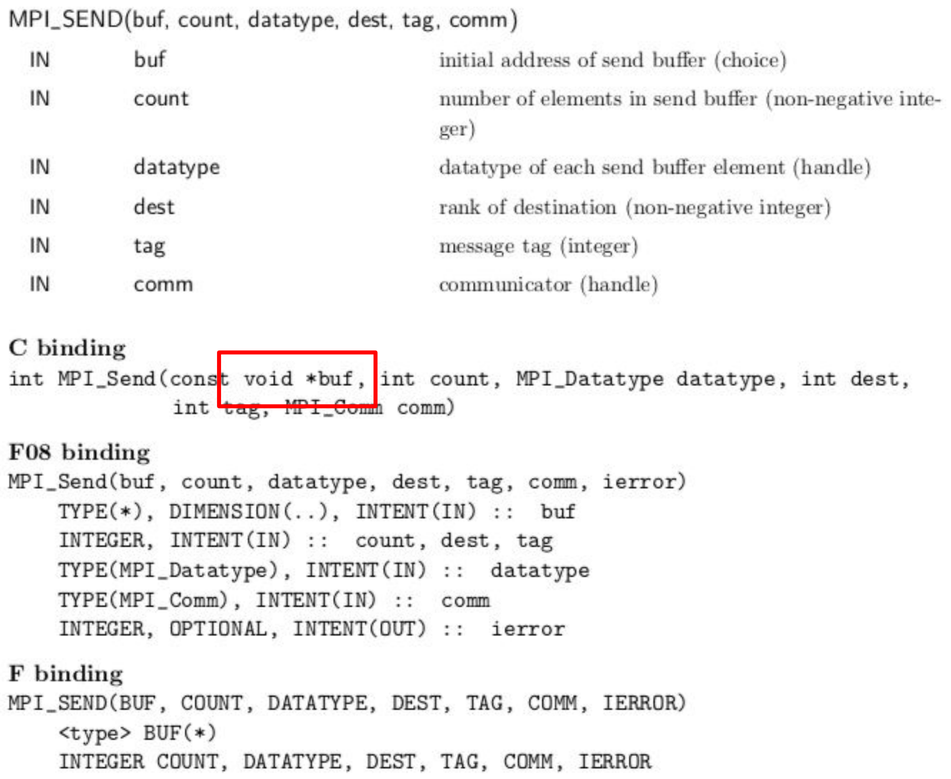
IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Pythonized



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

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

C binding

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

F08 binding

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

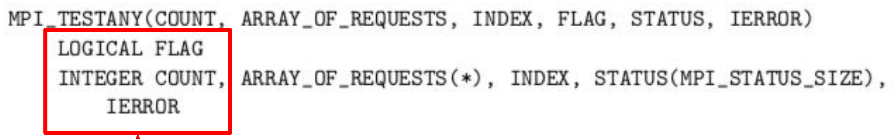
F binding

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

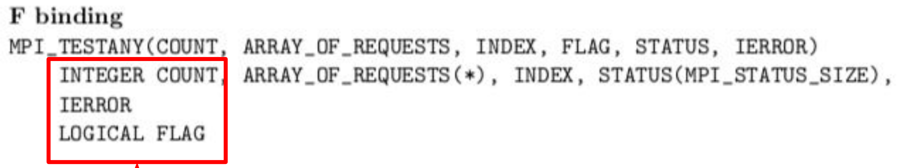
RenDiff: Fortran param listing

MPI-3.1

Pythonized



```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
        IERROR
```



F binding

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
    LOGICAL FLAG
```

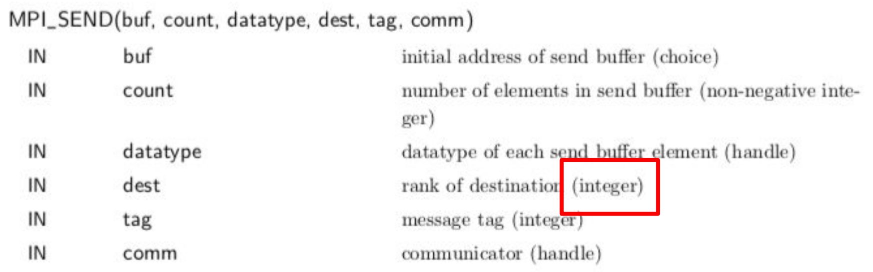
1. Listing order is different
2. Lack of indent for wrapped INTEGER line
3. But still semantically equivalent

Rendering differences compared to MPI-3.1

- Possibly contentious changes:
    - “(...blah…)” to the right in LIS sometimes different
    - Particularly in treatment of different types of integers (non-negative, positive, state, plain, not appearing in old text/only appearing in new rendering, …)

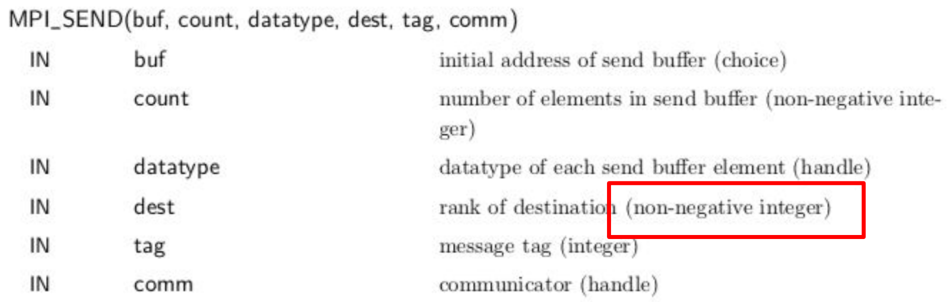
RenDiff: Differences in LIS (example 1)

MPI-3.1	Pythonized



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

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



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

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

Technically, this is a clarification beyond what MPI-3.1 stated, but it is correct.
Remember: the value of MPI_PROC_NULL is not specified.

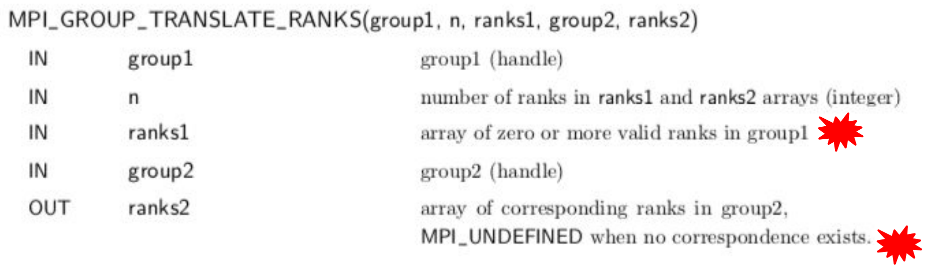
Issue:

- In terms of peer specification, “rank” is an integer (which is always >= 0)
- In terms of topology, edges are specified as “non-negative” integers (but are expressed in terms of ranks)

RenDiff: Differences in LIS (example 2)

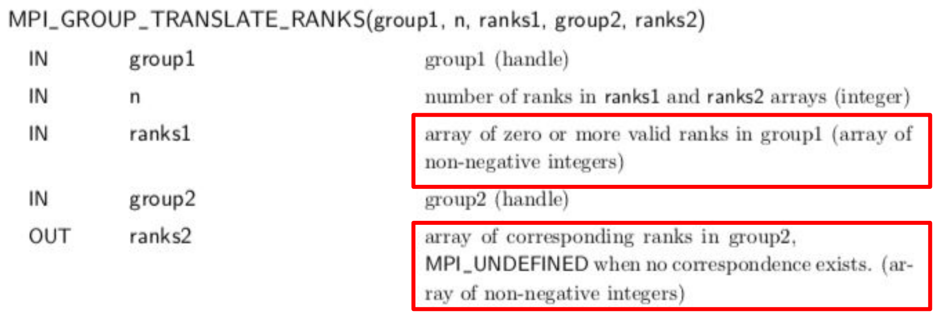
MPI-3.1

Pythonized



MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)

IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.



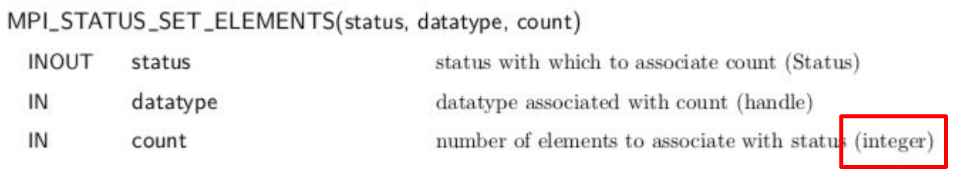
MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)

IN	group1	group1 (handle)
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)
IN	ranks1	array of zero or more valid ranks in group1 (array of non-negative integers)
IN	group2	group2 (handle)
OUT	ranks2	array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists. (array of non-negative integers)

MPI-3.1 did not specify a () clause in the LIS for these 2 parameters

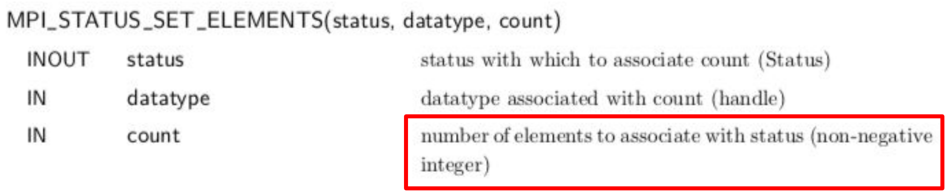
RenDiff: Differences in LIS (example 3)

MPI-3.1 | **Pythonized**



MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (integer)



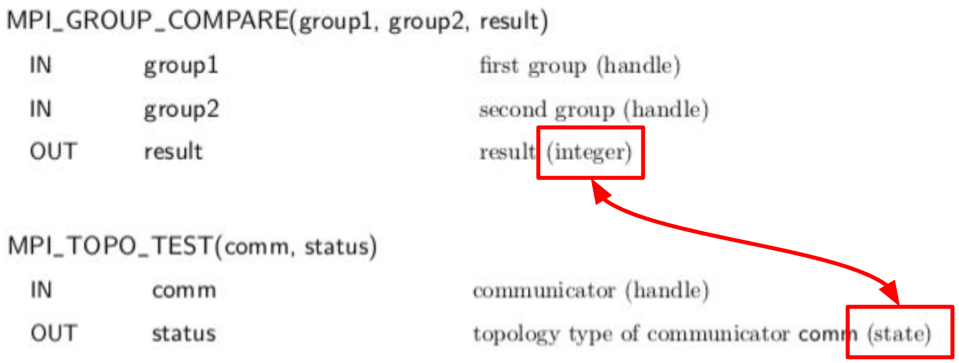
MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT	status	status with which to associate count (Status)
IN	datatype	datatype associated with count (handle)
IN	count	number of elements to associate with status (non-negative integer)

Because MPI_SEND (etc.) number of elements
is expressed as a non-negative integer

RenDiff: Differences in LIS (example 4)

MPI-3.1



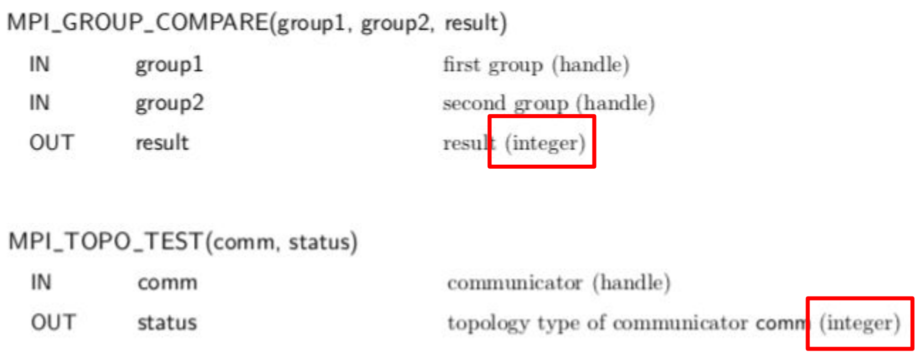
MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator comm (state)

Pythonized



MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	first group (handle)
IN	group2	second group (handle)
OUT	result	result (integer)

MPI_TOPO_TEST(comm, status)

IN	comm	communicator (handle)
OUT	status	topology type of communicator comm (integer)

Some enum outputs are marked as “integer” in MPI-3.1;
others are marked as “state”

Which should we use for Pythonization?

How do we check for correctness?

Machine-assisted comparison to MPI-3.1

Automated comparison of:

- Bindings LaTeX for v3.1.x branch
  vs.
- Rendered bindings LaTeX for v4.0.x branch

Anything that is different, let a human review:

1. Mark it as “ok”, *or*
2. Mark it as “need to go check the LaTeX / Python”

Current status

- All bindings have been Pythonized
- First correctness pass complete
- Need reviews from Chapter Committees

Remember: this is only on the `mpi-4.x` branch
We are not Pythonizing the `mpi-3.x` branch

The Plan

1. Today: hand off PDF / LaTeX to Chapter Committees
    a. Pull request for Pythonization of the v4.0.x branch
2. Get a detailed review from Chapter Committees
    a. Use the automated tool to show the differences between 3.1 and 4.0 HEAD
    b. PR the JSON results of your comparison
    c. If changes are necessary, make them yourself or give us an itemized list
3. Chapter Committees must approve differences by **21 Jan 2020**
    a. That's T-4 weeks before the Feb 2020 MPI Forum physical meeting
    b. We need time to make any final corrections/fixes before the T-2 week deadline
4. WG submits final PDF by 4 Feb 2020
    a. T-2 weeks before meeting
5. Errata vote during Feb 2020 MPI Forum physical meeting
6. Merge the pull request

All existing MPI-4 PRs must be Pythonized

Wiki with instructions for chapter authors

(including tips for those who don’t know Python)

Live demo of comparison tool for Chapter Committees

Wiki page with comparison tool instructions

Things for Chapter Committees to check

1. Did we get the translation from LaTeX to Python (to rendered LaTeX) right?
2. Did we accidentally delete anything?
    - Part of a binding
    - Other text
3. Did we accidentally add anything?
    - Part of a binding
    - Other text
4. Did we accidentally edit anything?
    - Other text