Advanced MPI

John Urbanic

Parallel Computing Scientist
Pittsburgh Supercomputing Center

Distinguished Service Professor
Carnegie Mellon University

Copyright 2025

MPI Advanced Features

In the MPI Basics talk we only touched upon the core MPI routines. Here we will discuss some of the advanced features that you are likely to use. We've grouped them thematically as just "looking through the index" can be intimidating.

- Datatypes
- Collective Communication
- Topology (Communicators)
- Non-Blocking Communications
- Single-sided Communications (RMA)
- Shared Memory
- Hybrid Programming
- Performance Options
- Attributes
- Dynamic Processes
- MPI-IO
- Modern Fortran (and other languages)
- Library Support
- MPI Implementations

MPI Evolution

1993: Supercomputing 93 - draft MPI standard presented.

1994: Final version of MPI-1.0 released

1995: MPI-1.1

1996: MPI-2

1997: MPI-1.2

2008: MPI-1.3, MPI-2.1

2009: MPI-2.2

2012: MPI-3.0

2021: MPI-4.0

2025: MPI-5.0

2028? MPI-6

Although there are some general themes to various releases (3.0 has a lot of “ultra-scalability” features), we won’t review routines chronologically. I’ve gathered each topic across all releases as it makes much more sense that way.

User Defined Data Types

User defined data types are often somewhere between organizationally useful, and indispensably convenient. They serve two purposes:

- Translation between formats
  - Automatic conversion of data sent from a big-endian to little-endian node
  - Not very common these days

- Noncontiguous messages
  - *Many* cases, from rows and columns of arrays to dynamically allocated lists
  - Serious performance implications

User Defined Data Types

There are two steps to using a defined data type:

1) Create the type using one of the multiple creation routines

`MPI_Type_contiguous(count…)`	Could be used in our current Laplace.
`MPI_Type_vector(count,…,stride…)`	Adds a stride. Could be used for row/column.
`MPI_Type_indexed()`	Varying strides and block sizes.
`MPI_Type_create_subarray()`	What you would guess and very flexible.
`MPI_Type_create_struct()`	Most general.
`… more`	

These routines accept existing, possibly user-defined, types. So they can be layered to build even more complex types.

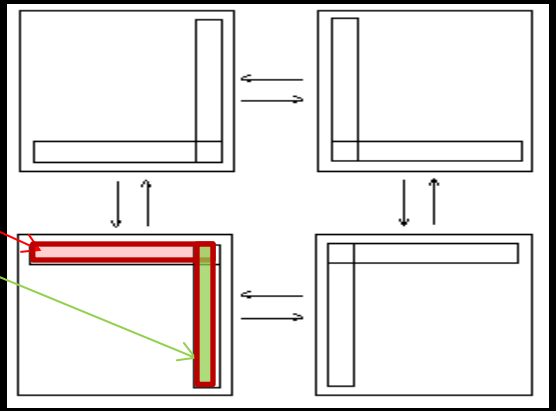
2) Commit the type

```
MPI_Type_commit()
```

Vector Type

(stencil code example)

```
MPI_Type_vector(int count,int blocklength, int stride, MPI_Datatype oldtype,MPI_Datatype *newtype)
```



```
MPI_Datatype row, column ;
MPI_Type_vector ( COLUMNS, 1, 1, MPI_DOUBLE, &row );
MPI_Type_vector ( ROWS, 1, COLUMNS, MPI_DOUBLE , &column );
MPI_Type_commit ( &row );
MPI_Type_commit ( &column );
.
.
//Send top row to up neighbor
MPI_Send(Temperature[1,1], 1, row, dest, tag, MPI_COMM_WORLD);
.
//Send last column to right hand neighbor (in the pictured layout)
MPI_Send(Temperature[1,COLUMNS], 1, column, dest, tag, MPI_COMM_WORLD);
```

Note that we are only sending 1 of each type, and that column and row sends look the same except for their type. We also could have used the MPI_Type_contiguous() routine to define the row type: MPI_Type_contiguous(ROWS, MPI_DOUBLE, &ROW)

3D Halo Codes

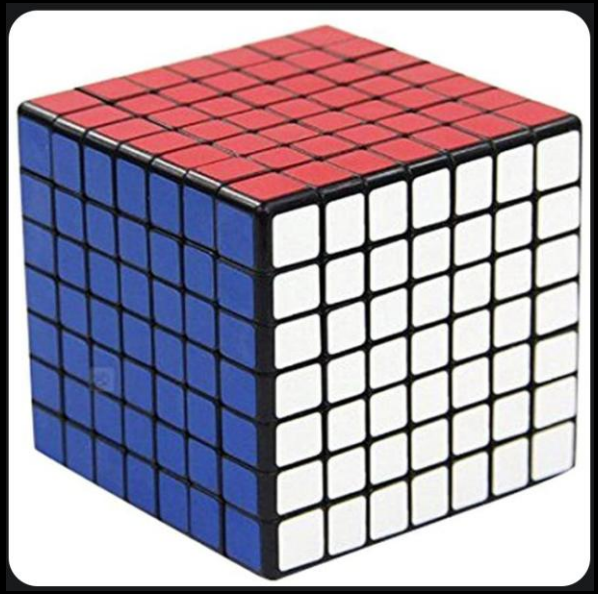
To send whole face, you could use

- Contiguous on one face to send 49 elements
- Vector on another to send 49 elements with a stride of 7
- Vector on another face with a stride of 7x7=49

- Opposing faces use same type to send.
- Matching faces receives use same type.

But for Halo codes we want to exclude the ghost zones. So for the first face you need to have block lengths and count of 5, with the same stride of 7, but make sure you start on the first "real" grid point.

You can also use the MPI_Type_create_subarray routine to define these. It is usually simpler and more symmetric for these cases as you explicitly state the start points, dimensions and sizes and leave the stride calculations as implicit. You can get by with 3 different types of message (for the X, Y and Z directions) that cover all the sends and receives.



There is a nice walkthrough example at:

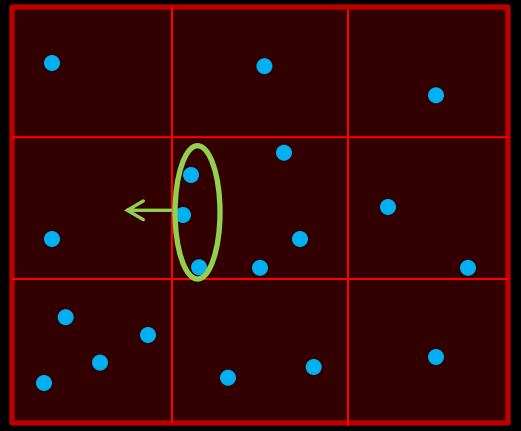
https://stackoverflow.com/questions/28182076/number-of-subarray-data-types-for-exchanging-2d-halos-in-3d-process-decompositio

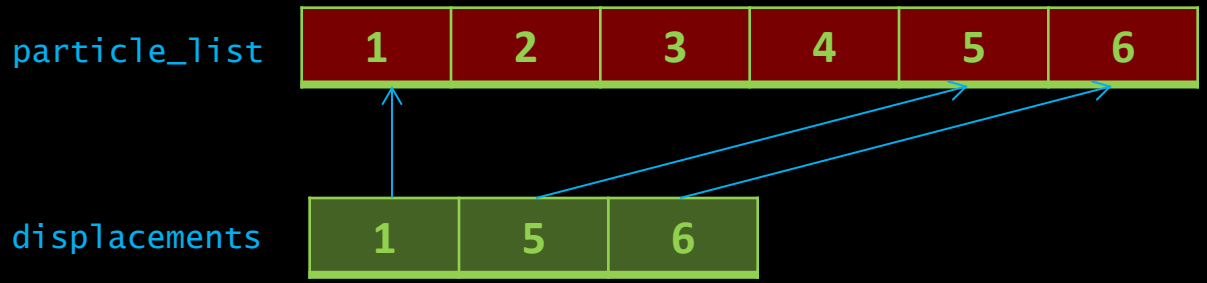
Indexed Type

(MD code example)

```
MPI_Type_create_indexed_block(int count, int blocklength,int displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_Datatype particle_type, buffer_type;
.
//When we create particle type it groups whatever the elements of
//particle_list really are: array of floats, C struct, …
.
//find particles on border and put their particle_list[]
//indices into displacements[]
.
.

MPI_Type_create_indexed_block(num_to_send, 1, displacements[],
                              particle_type, &buffer_type);

MPI_Type_commit (&buffer_type);

MPI_Send(particle_list, 1, buffer_type, left, tag, comm);

//And repeat every time!
MPI_Type_free(&buffer_type);
```





Indexed Type

(Receive Side)

Static size buffer:

```
MPI_Recv(incoming_particles, MAX_INCOMNG, particle_type, right, tag, comm, &status);
MPI_Get_count(&status, particle_type, &how_many);
```

Note that although we sent buffer_type, we are receiving particle_type.

Dynamically allocated buffer:

```
MPI_Probe(right, tag, comm, &status);
MPI_Get_count(&status, particle_type, &how_many);
MPI_Type_get_extent(particle_type, &lower_bound, &extent);
incoming_particles = (Particle *) malloc( how_many * extent);
MPI_Recv(incoming_particles, number, particle_type, right, tag, comm, &status);
```

Extent is the size, accounting for any padding. We will see this in more detail with structures.

Dynamic Memory

hIndexed Type

```
MPI_Type_create_hindexed_block(int count, int blocklength, int displacements[],MPI_Datatype oldtype, MPI_Datatype *newtype)
```

What if we want to send items from a linked list? This is often a bunch of individually allocated items scattered around memory. Our displacements are no longer nice element offsets in some array. We need to specify actual memory addresses.

```
MPI_Aint displacements[MAX_LIST_SIZE];
.
.
while (not_at_end_of_list){
            if (send_this_item){
                        MPI_Get_address(item_to_send, &displacements[number_to_send];
                        number_to_send++;
            }
            .
            .
}

MPI_Type_create_hindexed_block(number_to_send, 1, displacements, item_type, &items_message_type);

MPI_Type_commit (&items_message_type);

MPI_Send(MPI_BOTTOM, 1, items_message_type, dest, tag, comm);

MPI_Type_free(&items_message_type);
```

MPI_Aint is the MPI address type.

We get the memory address of each item we are going to send.

MPI_BOTTOM is used whenever we are using absolute addresses. It is usually just set to 0 for any sane architecture.

Hand Packing

(Refusing to use datatypes)

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)
```

We can assemble the buffer ourselves if we want. No new data type will be required.

```
position = 0;

MPI_Pack_size(MAX_SEND_COUNT, item_type, comm, &buffsize);

send_buffer = malloc(buffsize);
…
…
while (not_at_end_of_list){
            if (send_this_item){
                        MPI_Pack(item_to_send, 1, item_type, send_buffer, buffsize, &position, comm);
            }
            …
            …

}

MPI_Send(send_buffer, position, MPI_PACKED, dest, tag, comm);
```

Position is both input and output. It gets incremented as the buffer grows.

Hand Unpacking

```
int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

```
MPI_Recv(recv_buffer, MAX_SEND_COUNT, MPI_PACKED, source, tag, comm, &status);

MPI_Get_count(&status, MPI_PACKED, &total);

position = 0;

while (position < total){
            MPI_Unpack(recv_buffer, total, &position, &newitem, 1, item_type, comm);
            }
            …
            …
}
```

Position is input and output. It gets updated as the message is unpacked.

We could also receive these wholesale:

```
MPI_Recv(recv_buffer, MAX_SEND_COUNT, item_type, source, tag, comm, &status);

MPI_Get_count(&status, item_type, &total);
```

Hand Packing

And we can pack any combination of types that we wish:

```
struct {
    int    values[10];
    char   name[STRING_SIZE];
    double variance;
    int    total;
} to_send;

position =0;

MPI_Pack(&to_send.values, 10, MPI_INT, sendbuf, buffersize, &position, comm);
MPI_Pack(&to_send.name, STRING_SIZE, MPI_CHAR, sendbuf, buffersize, &position, comm);
MPI_Pack(&to_send.variance, 1, MPI_DOUBLE, sendbuf, buffersize, &position, comm);
MPI_Pack(&to_send.total, 10, MPI_INT, sendbuf, buffersize, &position, comm);

MPI_Send(sendbuf, position, MPI_PACKED, dest, tag, comm);
```

Position is both input and output. It gets incremented as the buffer grows.

Note that we have to pay attention to this structure on the corresponding `MPI_Unpack`.

Defining Structs

```
MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype)
MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
MPI_Type_create_resized(MPI_Datatype oldtype,MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)
```

We can define structs as actual MPI datatypes with `MPI_Type_create_struct` but it requires some attention to detail because of packing issues. Due to alignment issues of all modern processors, structures will often be padded in implementation dependent ways. A structure like:

```
struct {
      char x;
      int y;
      double z;
} example;
```

This is why "extents" and "lower bounds" pop up in datatype oriented routines.

will rarely occupy sizeof(char)+sizeof(int)+sizeof(double) bytes of memory. Thus the correct way to define a struct in MPI is to:

1) Get the relative memory displacements of each member with `MPI_Get_address`.
2) Create a structure with `MPI_Type_create_struct` using these displacements.
3) Check the extent of the new datatype against the C size (using sizeof) with `MPI_Type_get_extent`.
4) If necessary, adjust the size to match using `MPI_Type_create_resized`.
5) Finally, commit the type with `MPI_Type_commit`.

The alternative, which only trades off portability to heterogeneous systems (an increasingly minor concern), is to simply use MPI_BYTE:

```
MPI_Send( &example_array[index], sizeof(example), MPI_BYTE, dest, tag, comm);

MPI_Recv( &receive_array[index], sizeof(example), MPI_BYTE, source, tag, comm, &status);
```

Performance Implications of Datatypes

You may have the impression that datatypes are primarily a syntactical convenience. While this is undoubtedly true in many cases, the other motivation for them is very much performance. As we have seen elsewhere in MPI, memory copies are expensive. By defining a datatype, you give the system a chance to just grab the data straight from memory and send it out. Assembling everything into one buffer first will always require a copy.

This might not always be possible, and sometimes hand packing data is sensible. But ask yourself if defining the location of the items to be sent might enable the system to avoid this extra copy. Especially note that many HPC system deal with strided memory access particularly well.

You may have also noticed that some of our examples redefine the datatype every iteration. That might seem like a lot of overhead. But, if that defines a pattern that allows the system to avoid an extra memory copy, it can be well worth it.

Collective Communications

The Broadcast and reduce operations have more complex analogs that are very useful in a variety of algorithms. Just like Bcast and Reduce, the benefit of using these routines, instead of point-to-point messages, is not only syntactical convenience, but also much better efficiency.

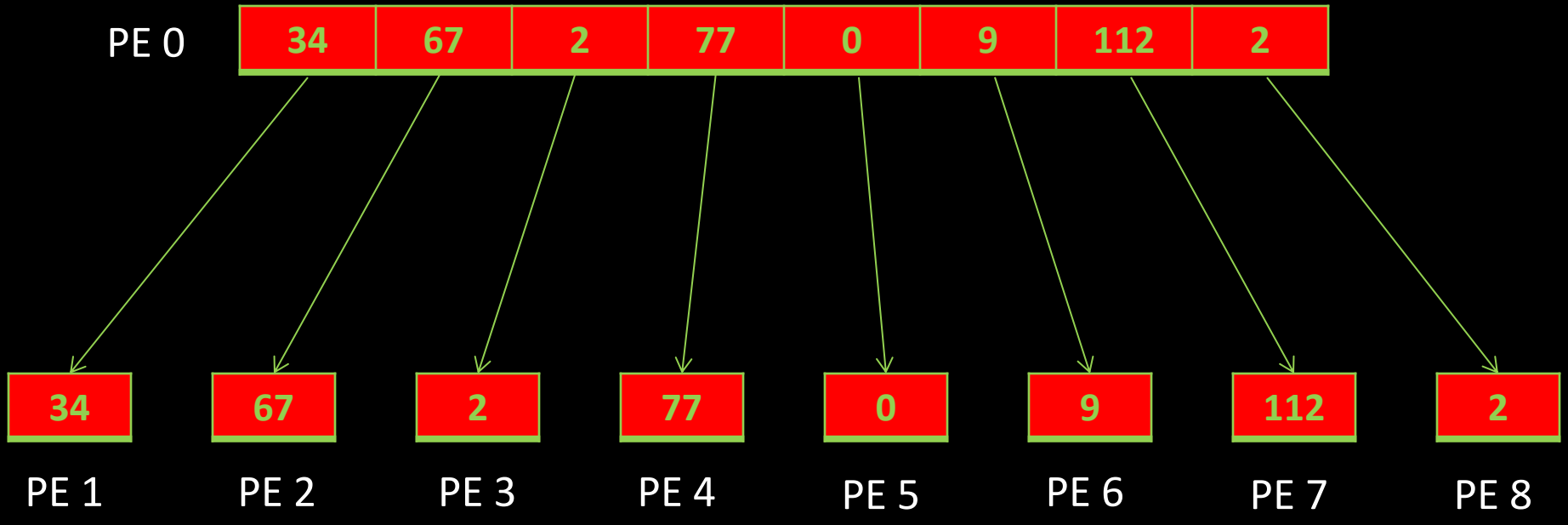
Keep in mind that these are called “collective” because every PE in the communicator *must* call these routines at the same time.

Let’s take a look at the basic idea, and then we’ll see the wide variety of related routines.

Scatter

One PE has some data that should be distributed differently to other PEs. Every PE in the communicator participates in this call:

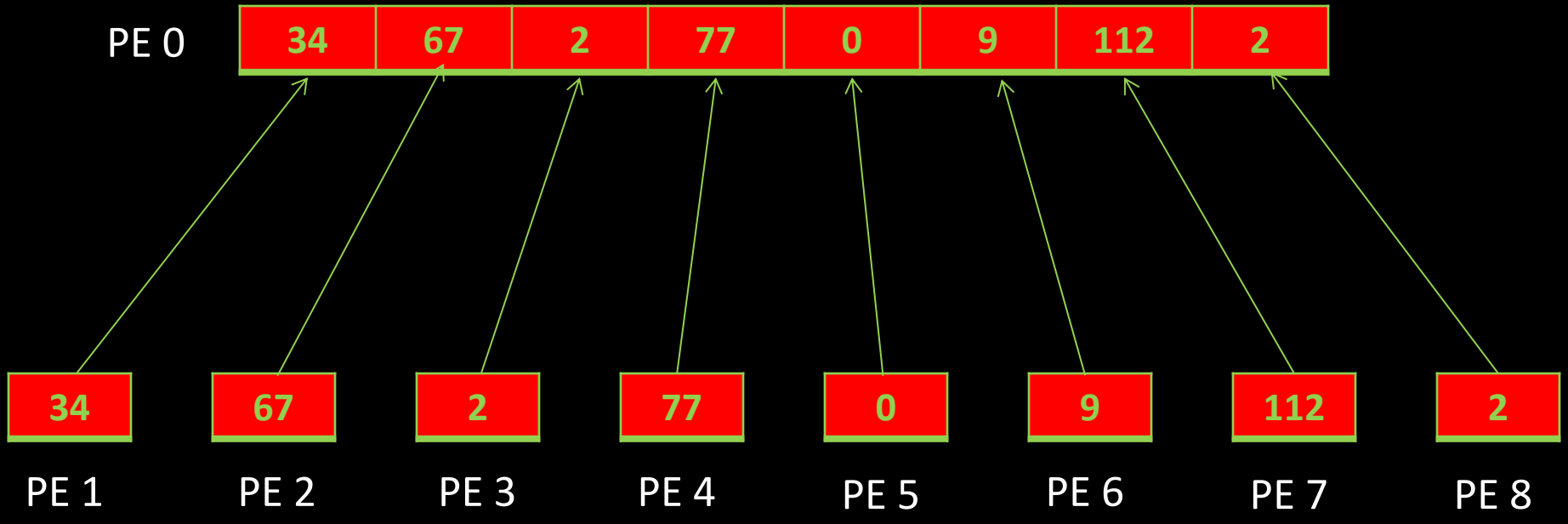
```
MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count,
            MPI_Datatype recv_datatype, int root, MPI_Comm communicator)
```



Gather

The exact opposite. The other PEs have some data that you wish to collect on one PE.

```
MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count,
           MPI_Datatype recv_datatype, int root, MPI_Comm communicator)
```



Scatter / Gather

There are several significantly different variations of these routines. When you need them, they are very helpful:

`MPI_Scatterv()`	Count can vary for each PE (uses an array for send_counts)
`MPI_Gatherv()`	“”

The “All” versions mean results are returned to all processors. No root parameter.

`MPI_Allgather()`	Like Gather, but all processes receive result
`MPI_Allgatherv()`	Allgather with varying count
`MPI_Alltoall()`	Each process sends distinct data to each receiver. Perfect for transpose or FFT.
`MPI_Alltoallv()`	Count can vary
`MPI_Alltoallw()`	Count, displacement and even datatype can vary (crazy flexible)

These are not as scalable and the MPI-3 standards group even considered deprecating vector collectives. Don’t use them for “exa-scale” codes.

Advanced Reduce

Reduce has some advanced variations and capabilities as well.

- Reduce can operate on arrays of data

```
float A[10], B[10];
.
MPI_Reduce(A, B, 10, MPI_FLOAT, MPI_MAX, 0, comm)
```

PE 0 will have a max from different PEs in B. Each will be the maximum in that element position.

- User defined reduction operations

```
int MPI_Op_create(MPI_User_function *user_fn, int does_operator_commute, MPI_Op *op)
```

You can define you own reduction operations when necessary.

- Location of min and max

You can determine the PE of the min and max for each element using operators MPI_MINLOC and MPI_MAXLOC. However, your input and output arrays need to be “MPI pair” datatypes to hold the PE number. Read the man pages…

More Advanced Reduce

- Allreduce

```
MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

  Every PE gets result, no root.

- Reduce local

```
MPI_Reduce_local(const void *inbuf, void *inoutbuf, int count, MPI_Datatype datatype, MPI_Op op)
```

  Apply any reduce operator to local data. inoutbuff = inbuf (op) inoutbuff element by element.

- MPI_Scan, MPI_Reduce_scatter

  There are still more variations of limited general usefulness, but just what you need for certain operations. Scan is good for certain load-balancing operations, while Reduce_scatter is exactly what you need for a distributed array/vector multiply. We can't cover them all, but what you should take away is that *if you find yourself writing a critical communication pattern, take a quick look at the MPI function index to make sure you are not overlooking the obvious.*

Communicators & Topologies

Communicators make life much easier in MPI. They enable three important capabilities:

- Groups (for algorithms)
- Topologies (for performance and decomposition)
- Contexts (for libraries)

Although I've probably just described what each of these mean to some degree, these are really best explained by example.

Communicators and Groups

The most basic way to think of communicators is as a way to break PEs into separate groups. You will most likely first encounter this when you want to have some PEs participate in an action while others do not. This immediately makes collective operations problematic. When using the COMM_WORLD communicator, all PE's have to participate in every collective. So, we will want to make our own sub-group.

Indeed, groups are one way MPI operates when creating sets of PEs. First we create a group, then we manipulate it until it represents the PEs that we want, then we create a communicator representing that group.

This following example illustrates how a group consisting of all but the 0 process of the World group is created. Then a communicator (comm_workers) is formed for that new group. The new communicator is used in a collective call just for the workers. Then all processes execute a collective call in the MPI_COMM_WORLD context.

Communicators to enable control flow

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

  int my_PE, send_data=1, recv_data1, recv_data2;
  MPI_Group group_world, group_workers;
  MPI_Comm comm_workers;
  int ranks[] = {0};  //Ranks to exclude, just 0 here

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_PE);

  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Group_excl(group_world, 1, ranks, &group_workers);
  MPI_Comm_create(MPI_COMM_WORLD, group_worker, &comm_workers);

  //Reduce just for workers
  if(my_PE != 0){
    MPI_Reduce(&send_data, &recv_data1, 1, MPI_INT, MPI_SUM, 0, comm_workers);
  }

  MPI_Reduce(&send_data, &recv_data2, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  //Proper clean up
  if (comm_workers != MPI_COMM_NULL) MPI_Comm_free(&comm_workers); //comm_workers is NULL on PE 0
  MPI_Group_free(&group_world);
  MPI_Group_free(&group_workers);

  MPI_Finalize();

}
```

recv_data1 = number of workers PE (PEs-1) on PE 1
recv_data2 = number of world PEs on PE 0
Everything else is unknown

Group Commands

There are enough group manipulation routines that it is usually very direct to craft the group you require.

```
MPI_Group_compare
MPI_Group_difference
MPI_Group_excl
MPI_Group_incl
MPI_Group_intersection
MPI_Group_range_excl
MPI_Group_range_incl
MPI_Group_rank
MPI_Group_size
MPI_Group_translate_ranks
MPI_Group_union
```

Communicator Creation Commands

It is also possible to work directly with communicators. This is what many of our topology oriented routines do. Here are a few of the basic versions:

```
MPI_Comm_create          (Just used)
MPI_Comm_group           (Just used)
MPI_Comm_dup             (Duplicates)
MPI_Comm_split
```

Comm_split is pretty powerful. It allows us to split a communicator into many fragments based upon a “color”. “key” can be used to determine ordering in the new communicator.

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

We could replace the three MPI comm/group routines in the previous example with this “groupless” code:

```
if (myPE = 0) then color=0;
else color =1;
MPI_Comm_split(MPI_COMM_WORLD,  color, 0, & comm_workers);
```

The inconsequential difference would be that Comm_create returns a NULL communicator to the excluded PE0, whereas PE 0 gets its own, isolated, comm_workers communicator with Comm_split.

Topologies

Communicators can be tools to make our data decomposition easier.

- Most MPI codes have a data decomposition that reflects the physical layout of the model.
- Often there is a locality relationship to the PE's, such as "nearest neighbor".
- If this logical topology can be properly mapped to the physical network and PE layout, it can greatly aid performance.
- At Petascale exploiting this is almost a necessity.

MPI's topology routines will help us with these issues. It has two general approaches to the subject:

- Cartesian topologies
- Graph relationships

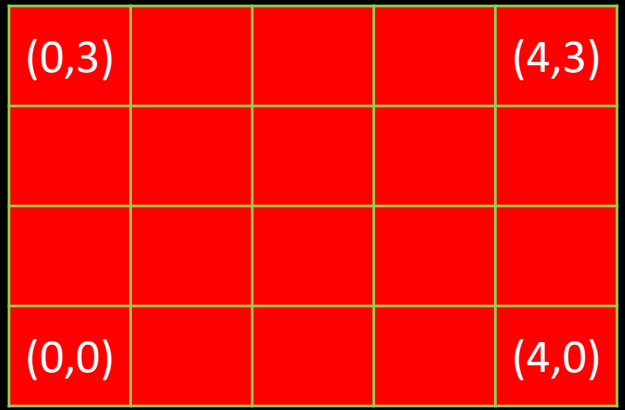
We will look at them both.

Cartesian Topologies

The first thing to do is to create a communicator that captures our topology.

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], int periods[], int reorder, MPI_Comm *comm_cart)
```

```
int dims[2] = { 5, 4 };
int periods = { 0, 0 };
MPI_Cart_create(comm_old, 2, dims[], periods[], 1, &comm_2D);
```



Use reorder = 1 = true so that MPI can reorder the PEs in the most optimal relationship.

Excess processes get a MPI_COMM_NULL. If we ran the above on 24 PEs, 4 would get that. Too few processes is an error.

The new communicator now contains this relationship and we can use it in various ways going forward.

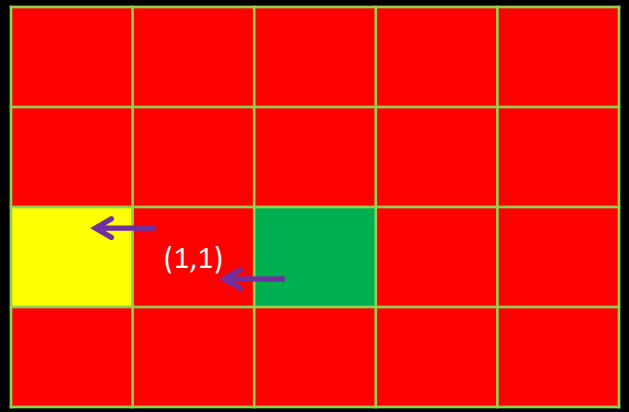
Cartesian Topologies Routines

How do we use this? Perhaps the most important thing in a cartesian layout is to know who your immediate neighbors are.

```
MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```

For example, suppose that the PE that is in the (1,1) location for this new communicator wants to know where to send and receive for a left shift of data. It would do this:

```
MPI_Cart_shift(comm_2D, 0, -1, &rank_source, &rank_dest);
```



PEs on the edge will get an MPI_PROC_NULL, unless there are periodic conditions, and then they would get the appropriate opposite PE.

Useful Cartesian Topologies Routines

There are a few other useful routines to manage yourself within a Cartesian topology:

```
MPI_Dims_create(int nnodes, int ndims, int dims[])
```
This will do the calculation of how to spread nnodes around some number of (perhaps constrained) dimensions.

```
MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])
```
Give it the comm and maxdims and it will return the full dims and the calling PEs coords.

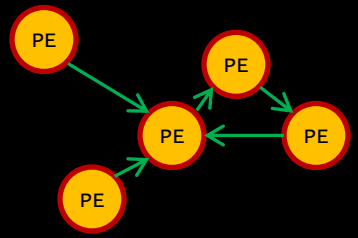
```
MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```
Give it any rank in the communicator and it will return the coords.

```
MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
```
Can return any hyperplane of the communicator as a new communicator. For our 2D example, we could set remain_dims = { 1, 0} and each PE would get a communicator for the row it is in.

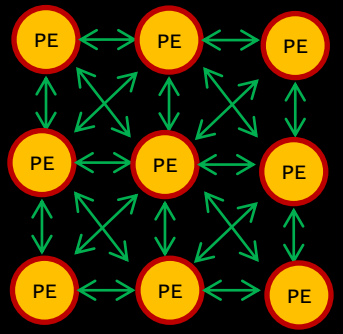
These routines (and others) overlap in capability, so there are usually several ways to accomplish any ultimate communication pattern. We could do almost anything by setting up colors just so and using MPI_Comm_split and some math, but these often fit the problem better.

Graph Topologies

To capture topologies that aren't strictly Cartesian, MPI has a graph API as well. You might think it is mostly applicable to crazy, convoluted graph relationships.



And, it could be. But, there are a lot of FEA or dynamic load balanced data decompositions that don't always look like simple Cartesian relationships. Even something as simple as our "5 point stencil" 2D example can easily become a "9 point stencil", which requires diagonal neighbors.

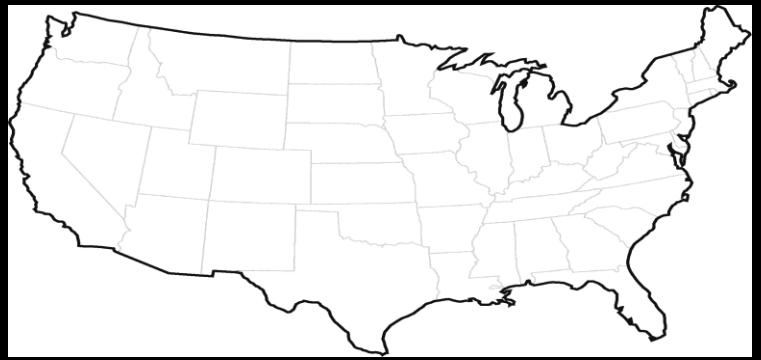


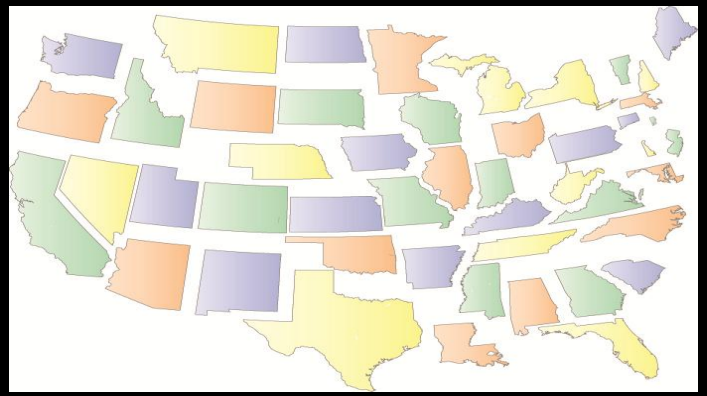
How often will you need these?

The ability to deal with irregular domain decompositions is a major strength of MPI, and one main reasons it hasn't been supplanted by many other parallel programming contenders.

And irregular problems domains also lead to issues of load-balancing - a topic we have avoided thus far with our more regular example problems. Most interesting and realistic problems aren't quite so cartesian.

Let's consider our simple weather modeling problem.





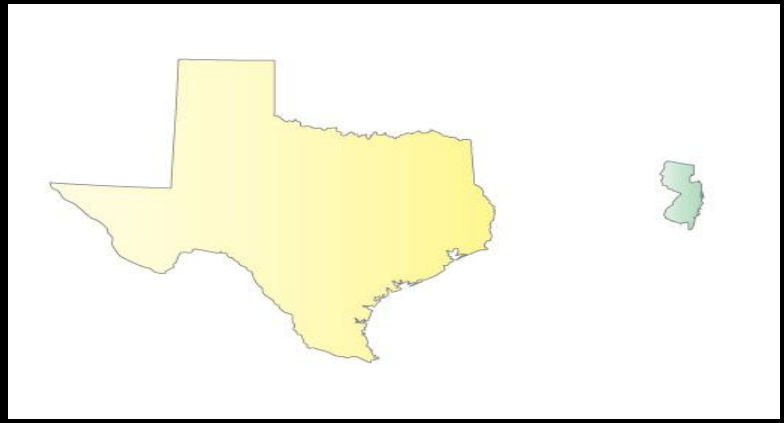
Load Balancing

Obviously, we should be thoughtful and make sure that each PE has a reasonably equal amount of work for each time step.

Less obvious is how to accomplish this. Our equal sized patches are not likely to be cartesian.

Also, we probably want any scheme to be able to accommodate any number of PEs at runtime.

And it might even be desirable to have a scheme that can change during the run itself. Picture a hurricane model, where you probably want to have a lot more elements around the hurricane than far away where less activity is happening.

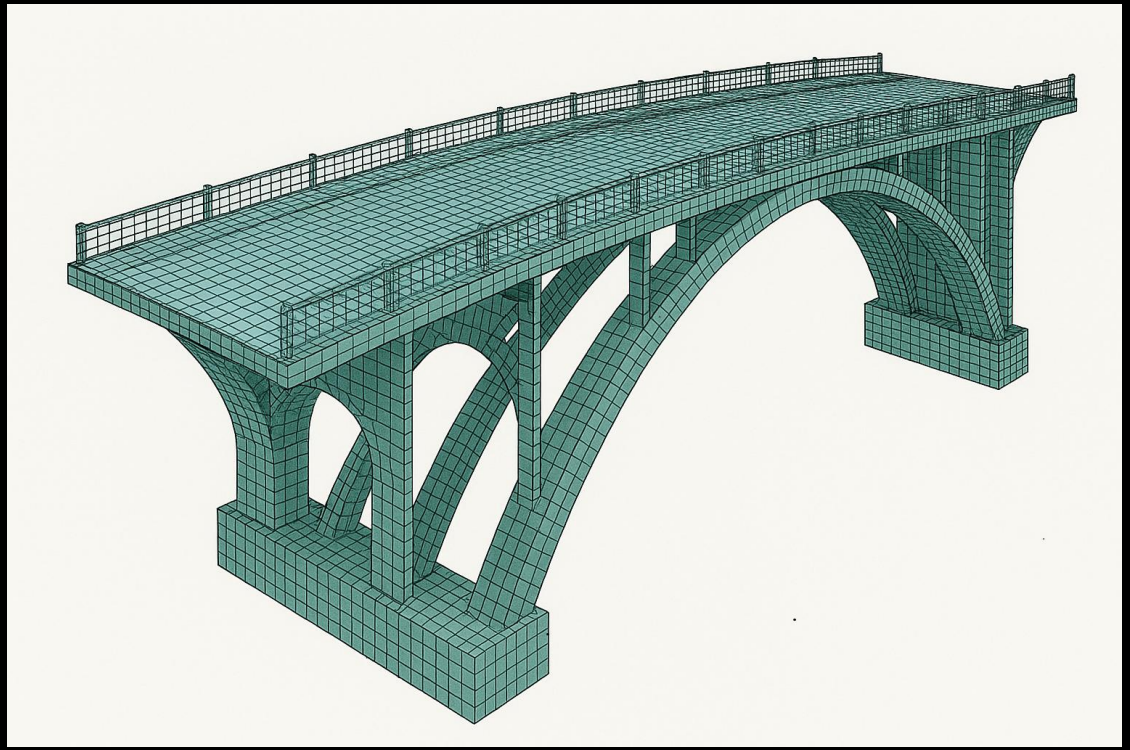


Now realize that that hurricane it going to move. Now we require *Adaptive Mesh Refinement*.

And this requires a simultaneously evolving domain decomposition.

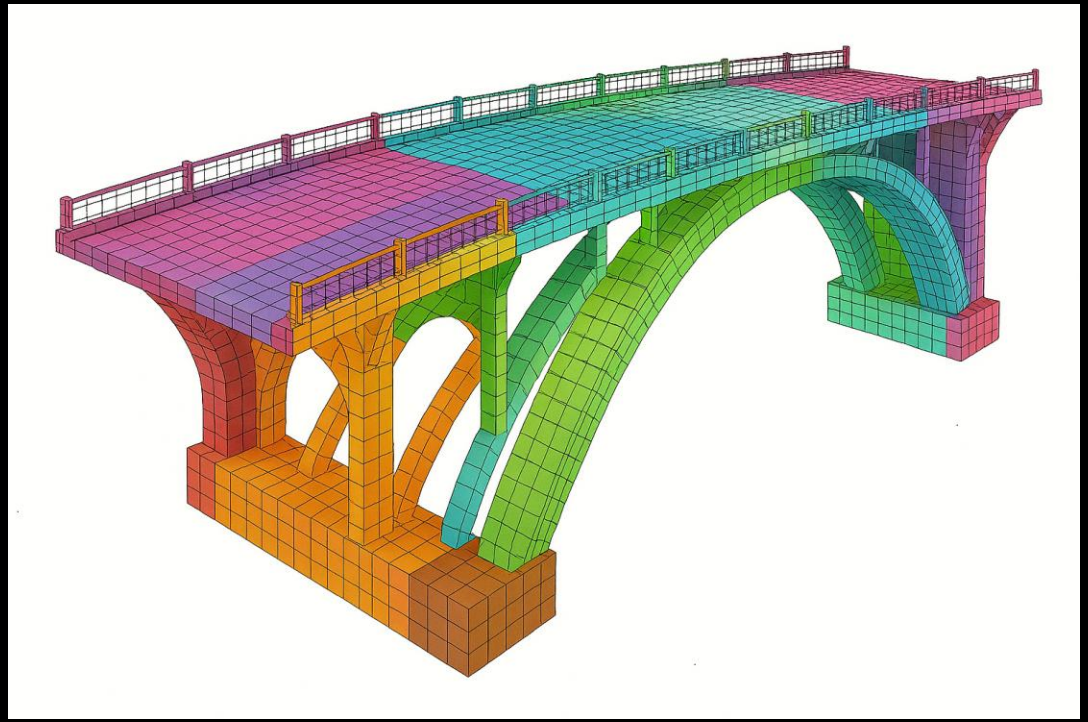
3D Irregular Meshes

Of course the real world is 3D, at least, and we usually find ourselves simulating something more like this.



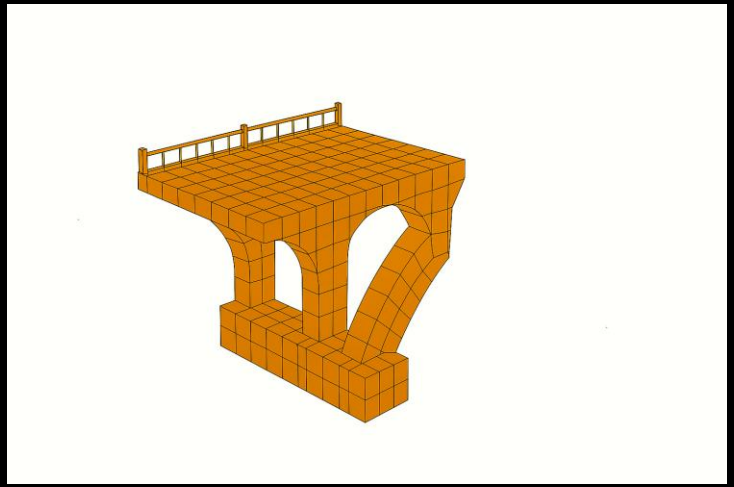
Decomposing 3D Irregular Meshes

A domain decomposition might look something like this.



Halo Zone Technique

Our ghost/halo zone technique is still required to communicate between these PEs. Now picture all the communicating surfaces for one of them.



I hope you can see how our communicators will make this bookkeeping much easier (early MPI codes had to do it all manually). Even so, there are libraries that are very helpful in helping to decompose meshes for you. Even AMR problems - of which this is not one.

Graph Topologies

MPI has three different interfaces to construct graphs.

- The original MPI-1 general graph topology. Not scalable, so we will treat it as deprecated.
- Adjacent graph interface
- General graph interface

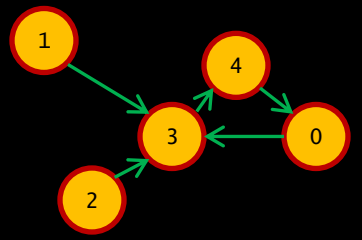
These last two avoid specifying the complete graph at any given PE. This is generally what you want, and always what you need for a seriously scalable code.

Adjacent Graph Interface

This keeps all creation local, and hence is the preferred method. Use whenever you know who your neighbors should be.

```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,  int sources[],      int sourceweights[],
                                                  int outdegree, int destinations[], int destweights[],
                                                  MPI_Info info, int reorder,        MPI_Comm *comm_dist_graph)
```

PE	Indegree	Sources	Outdegree	Dest
0	1	4	1	3
1	0		1	3
2	0		1	3
3	3	0,1,2	1	4
4	1	3	1	0



Weights might be useful to the implementation. info might provide further hints to the system. Allowing reordering will allow the implementation to chose a better (NP-hard!) mapping.

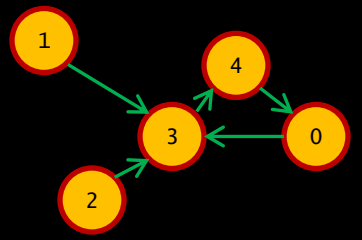
General Graph Interface

More complex and requires communication. But it does allow any PE to specify any edge. Use if you must.

```
MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[], int degrees[], int destinations[],
                      int weights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

This could have countless variations. Here is one.

PE	Number	Sources	Degree (dest per source)	Destinations
0	3	0,1,2	1,1,1	3,3,3
1	0			
2	0			
3	0			
4	2	3,4	1,1	4,0



It so happens that PEs 0 and 4 have the complete graph definition here.

Graph Query Interfaces

There routines allow you to find out who your neighbors are.

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted)
```

Give it the comm and maxdims and it will return the number of incoming and outgoing edges. Also lets you know if weights are supplied.

```
MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree,  int sources[],      int sourceweights[],
                                        int maxoutdegree, int destinations[], int destweights[])
```

Give it a communicator and the counts from above, and it will return sources, destinations and weights.

But so far, none of these routines actually accomplish any communication. We can use all of our existing send and receive routines. But, we also have some routines that really leverage these new communicators…

Neighborhood Collectives

This is the super scalable payoff for our “neighborhood aware” communicators. We have operations which perform collective communications within our close neighborhood. The two most important are:

```
MPI_Neighbor_allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                       void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Each PE i gathers data items from each PE j if an edge (j,i) exists in the topology graph, and each PE i sends the same data items to all PEs j where an edge (i,j) exists. The send buffer is sent to each neighboring PE and the l-th block in the receive buffer is received from the l-th neighbor. This could be though of as a neighborhood Bcast, from the sending PEs perspective.

```
MPI_Neighbor_alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Each PE i receives data items from each PE j if an edge (j,i) exists in the topology graph or Cartesian topology. Similarly, each PE i sends data items to all processes j where an edge (i,j) exists. This call is more general than MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. The k-th block in send buffer is sent to the k-th neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.

The order of the send and receive buffers for graph topologies is the order of PEs returned by the neighbor query routine.

For Cartesian topologies the order is the order of dimensions. Buffers for boundary PROC_NULL PEs must still exist, but aren’t read or written.

More Neighborhood Collectives

There are vector version of these collectives, to allow varying counts:

```
MPI_Neighbor_allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                        void *recvbuf, const int recvcounts[], const int displs[],
                        MPI_Datatype recvtype, MPI_Comm comm)

MPI_Neighbor_alltoallv( void *sendbuf, int sendcounts[], int sdispls[], MPI_Datatype sendtype,
                        void *recvbuf, int recvcounts[], int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)
```

The extra flexible w version allows us to mix different types. This can be useful to allow an efficient in-place transfer (think about exchanging colums/row datatypes in this one call).

```
MPI_Neighbor_alltoallw( void *sendbuf, int sendcounts[], MPI_Aint sdispls[], MPI_Datatype sendtypes[],
                        void *recvbuf, int recvcounts[], MPI_Aint rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

and although we haven't delved into non-blocking routines in a serious fashion yet, this is a good place to mention that we will have non-blocking versions of all of these as well.

```
MPI_Ineighbor_allgather()
MPI_Ineighbor_allgatherv()
MPI_Ineighbor_alltoall()
MPI_Ineighbor_alltoallv()
MPI_Ineighbor_alltoallw()
```

One neat communicator feature

```
MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm * newcomm)
```

MPI added in a non-collective communicator creator. Up until then all communicator routines had been collective. This generally hasn't been a burdensome limitation, but there are some nifty applications for a communicator that can be created independently by subsets of an existing communicator:

- We can load balance in flexible new ways where groups can redefine themselves. A bunch of idle PEs can aggregate without requiring the super-pool to synchronize with them.

- We can route around a failed PE. Previously we would need that PE to participate in creating a new comm, or else the others were permanently locked out of any collective operations.

Communicators Summary

While simple communicators offer some organizational convenience to point-to-point communications (send and receive), we could always substitute our own bookkeeping.

However, all collective operations (bcast, reduce, barrier, gather, scatter and all their variants) are completely dependent on their communicator. If we can't define a proper communicator, we can't use them in anything but very simple patterns.

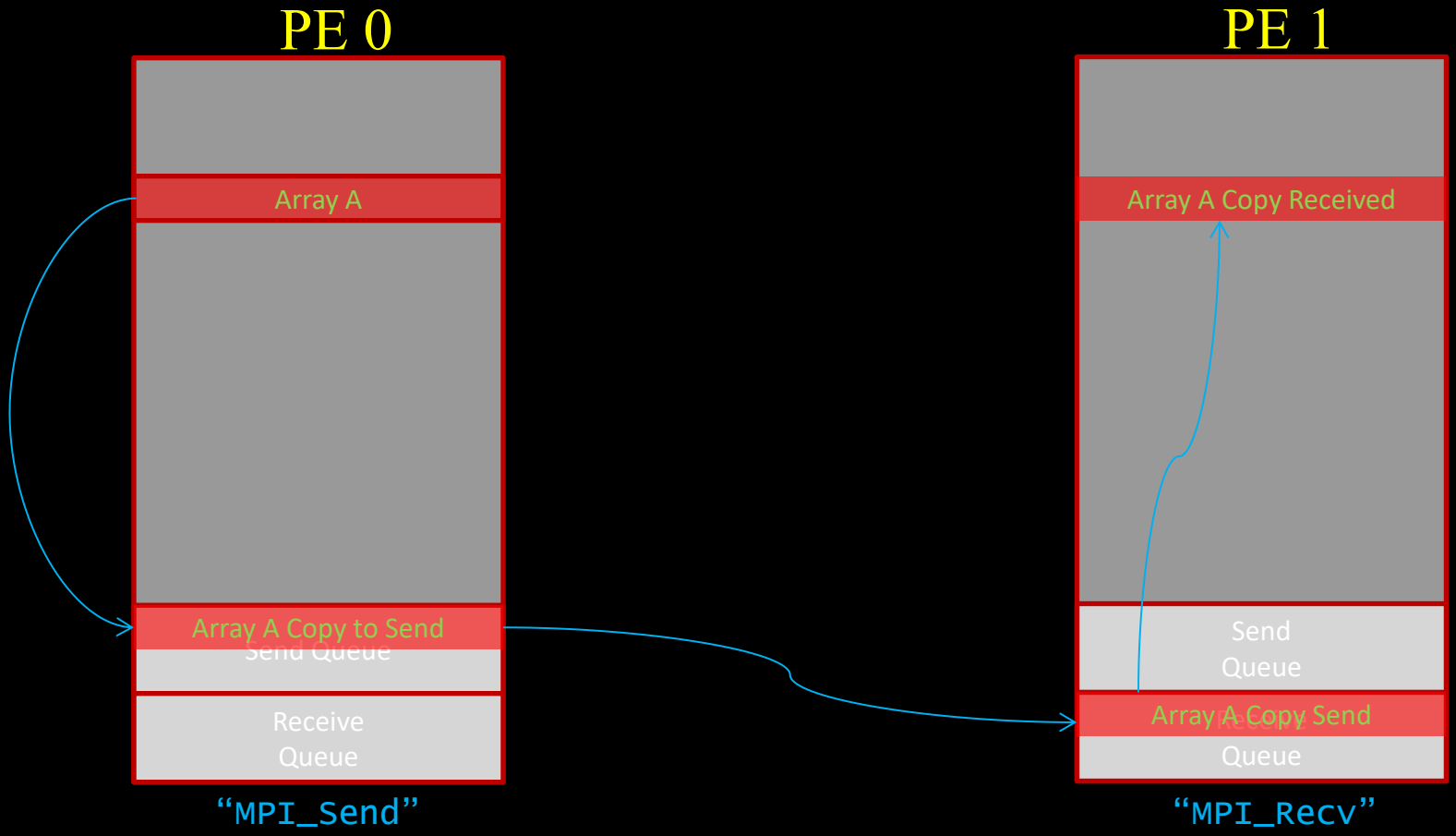
And as our problems grow more complex, we can fold a lot of the organization into the communicators themselves. So much so that a complex operation like ghost zone communications in a complex FEA mesh can be combined into a single call.

MPI Communication Hierarchy

MPI has a very graceful ladder of communication optimizations that start with our basic blocking routines and allow us to incrementally optimize sections to use multi-million core platforms with ideal efficiency.

1. Blocking Message Passing
2. Non-blocking Message Passing
3. RMA Active Target
4. RMA Passive Target
5. RMA Scalable Synchronization

Default MPI Messaging



Non-Blocking Messaging

We didn't really give non-blocking messaging it's fair shake in the MPI Basics talk. Non-blocking sends and receives can make sending massive numbers of messages, or large messages, possible without deadlock or buffer overruns.

Imagine how this piece of code behaves as the message size becomes very large:

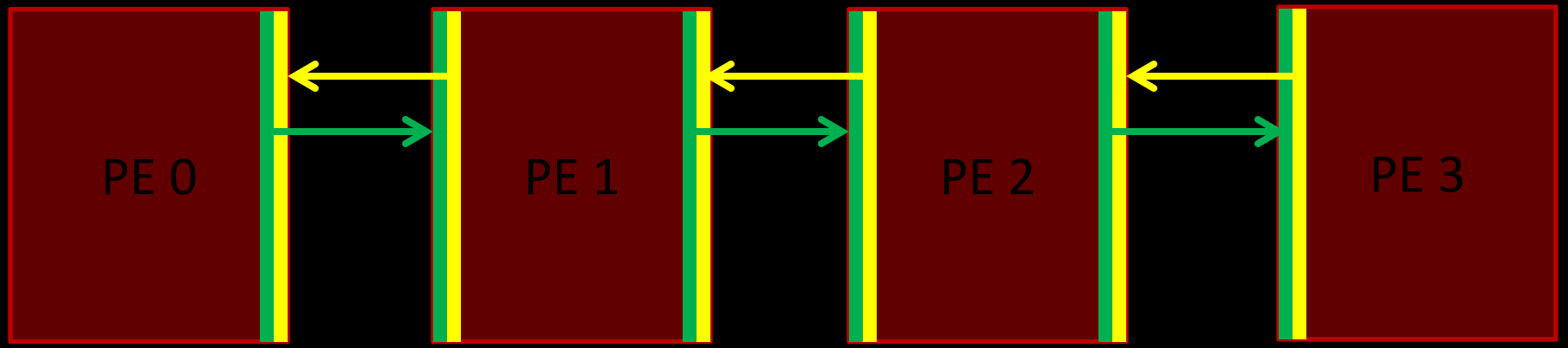
```
if(rank==0{
          MPI_Send(x to process 1)
          MPI_Recv(y from process 1)
}
if(rank==1){
           MPI_Send(y to process 0);
           MPI_Recv(x from process 0);
}
```

- At some point, the Send() will block until the other PE can absorb it. Both sends end up blocking each other in a deadlock. MPI_Ssend ("don't return until receive has started") will catch these immediately.

- Also very important is the fact that *MPI_Isend()* does not need to make an extra copy to protect the integrity of any not-quite-sent data. This can be a huge win.

Not just a theoretical consideration

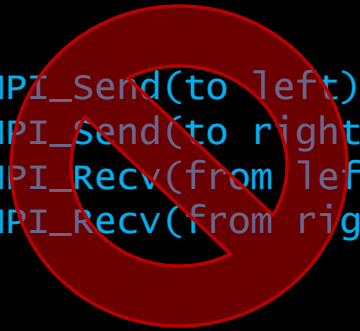
The Laplace solver used for our Hybrid Challenge competition is usually parallelized with a send/receive pattern like this:



The classic “ghost zone” data exchange.

Two Blocking Methods

There are two similar ways of coding this that we might try:



```
MPI_Send(to left)
MPI_Send(to right)
MPI_Recv(from left)
MPI_Recv(from right)
```

```
MPI_Send(to left)
MPI_Recv(from right)
MPI_Send(to right)
MPI_Recv(from left)
```

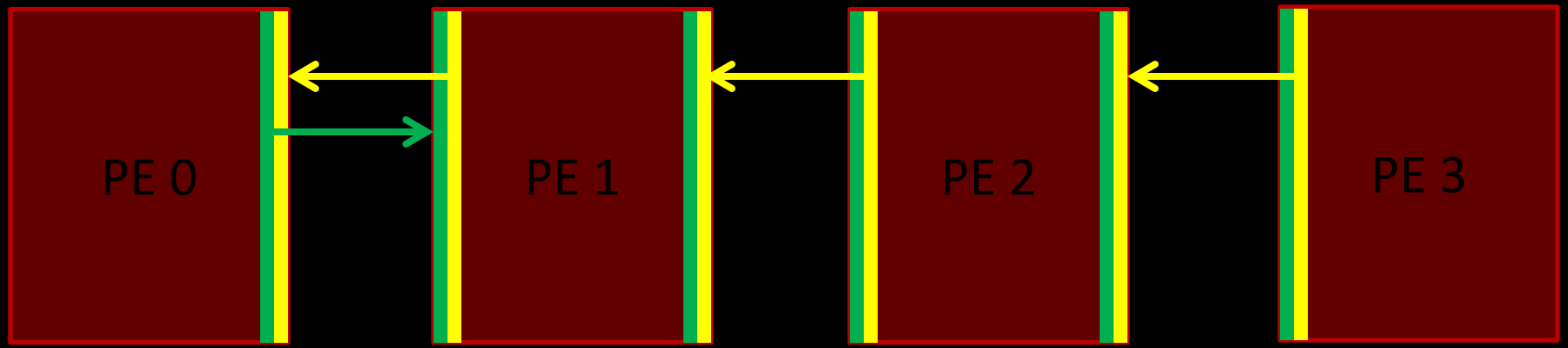
On Blue Waters they both work OK solving the beginning 1000x1000 problem. But when we scale up to the full competition size (10000x10000) one of them hangs. Where?

PE's 1-3 are blocking sending to the left, and PE 0 is blocking on the send to the right.

Hung…

PE's 1-3 are blocking sending to the left, and PE 0 is blocking on the send to the right.

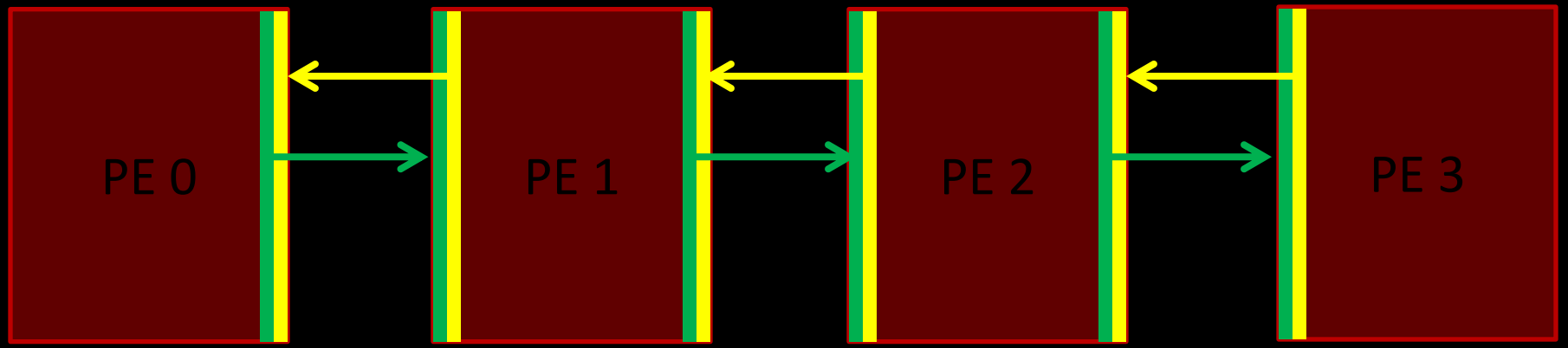
```
MPI_Send(to left)
MPI_Send(to right)
MPI_Recv(from left)
MPI_Recv(from right)
```



Is our other solution truly the answer? *Note that using MPI_Ssend() here would have caught this problem right away!*

Cascading Messages

At least the second solution doesn't' hang. But it does results in a sequential process here that we don't really want.



We can do better, and without any complication.

```
MPI_Send(to left)
MPI_Recv(from right)
MPI_Send(to right)
MPI_Recv(from left)
```

MPI_Sendrecv

For this particular situation, MPI has a ready answer

```
MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest,   int sendtag,
             void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag,
             MPI_Comm comm, MPI_Status *status)
```

This does what you'd hope, and solves our problem for these kind of data shifts, but we can avoid these problems in general with a different approach.

Basic Non-Blocking Routines

As we said before, it is no big deal to modify your typical blocking algorithm to do this. Add an "I" to the routine name, a "request" flag parameter, and check to see when operations have completed before using the data.

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,MPI_Comm comm, MPI_Request *request)

MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Also:

```
MPI_Waitall()
MPI_Waitany()
MPI_Waitsome()
MPI_Testall()
MPI_Testany()
MPI_Testsome()
```

And the useful variations:

```
MPI_Send_Init()
MPI_Recv_Init()
MPI_Start()
MPI_StartAll()
```

If you have unchanging communication patterns, you should really use these.

Non-Blocking Basics

Here is our “shifter” exercise done with non-blocking routines.

```
#include "mpi.h"

main(int argc, char **argv){

  int size, rank, send, value_sent, value_recieved, tag=5;
  MPI_Request reqs[2];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  send = rank+1;
  if (rank == size-1) send = 0;

  value_sent = rank+100;

  MPI_Irecv(&value_recieved, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &reqs[0]);
  MPI_Isend(&value_sent, 1, MPI_INT, send, tag, MPI_COMM_WORLD, &reqs[1]);

  /* We _could_ do things here while we wait and poll with MPI_Test() */

  MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

  printf("PE: %d, Value %d\n",rank,value_recieved);

  MPI_Finalize();

}
```

Send_init and Recv_init as used by a *Summer Boot Camp Hybrid Challenge* winner

```
call MPI_Send_Init(temperature(1,columns), rows, MPI_DOUBLE_PRECISION, right, 1r, MPI_COMM_WORLD, request(1), ierr)
call MPI_Recv_Init(temperature_last(1,0), rows, MPI_DOUBLE_PRECISION, left, 1r, MPI_COMM_WORLD, request(2), ierr)
// 8 of these as winning solution did a 2D (left, right, up, down) decomposition on 10,000 x 10,000 size problem
      .
      .

do while ( dt_global > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
            do i=1,rows
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(i,j-1) )
            enddo
         enddo
      .
      .
         call MPI_StartAll(8,request,statuses)

         dt=0.0
      .
         do j=1,columns
            do i=1,rows
               dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
               temperature_last(i,j) = temperature(i,j)
            enddo
         enddo
      .
      .
         call MPI_WaitAll(8,request,statuses,ierr)
      .
      .
enddo
```

Allow communications to overlap with the temperature_last update and maximum delta search.

Make sure all is complete before using this data in the next iteration.

Mixing Blocking and Non-Blocking Messaging

You can mix and match blocking and non-blocking send/recv routines:

```
if(rank==0{
          MPI_Isend(x to process 1)
          MPI_Recv(y from process 1)
}
if(rank==1){
           MPI_Isend(y to process 0);
           MPI_Recv(x from process 0);
}
```

This solves our deadlock (and improves our performance: no extra buffer copy). Irecv() might save us another buffer copy. You can *not* mix blocking and non-blocking collectives.

Other Non-Blocking Routines

Almost every applicable MPI routine has a non-blocking version. Just add “I” to the routine name and add the “request” parameter. They operate pretty much as one would expect.

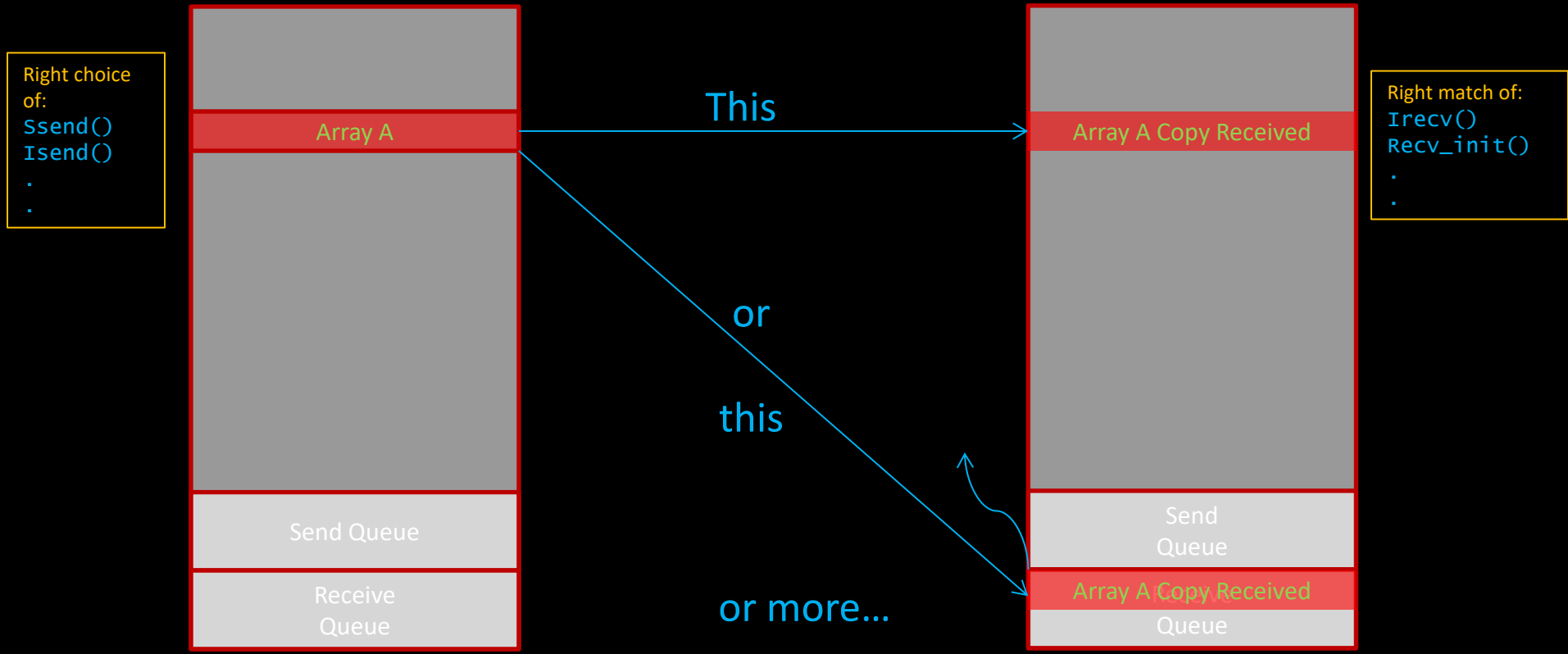
There is one that catches special attention, the non-blocking barrier:

```
MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

What use could a non-blocking barrier possibly have? The most common, and quite useful, is to let the other PEs know “I’m ready to move on (call `MPI_Ibarrier`), but while I am waiting I am going to go ahead and do something useful. I’ll keep checking (`MPI_Test`) to see when the rest of you are ready to move on too”.

Optimized MPI Messaging

We now have the tools to control much of the buffering and synchronization of our cooperative message passing. *If both ends do the right thing*.



Single Sided Communications

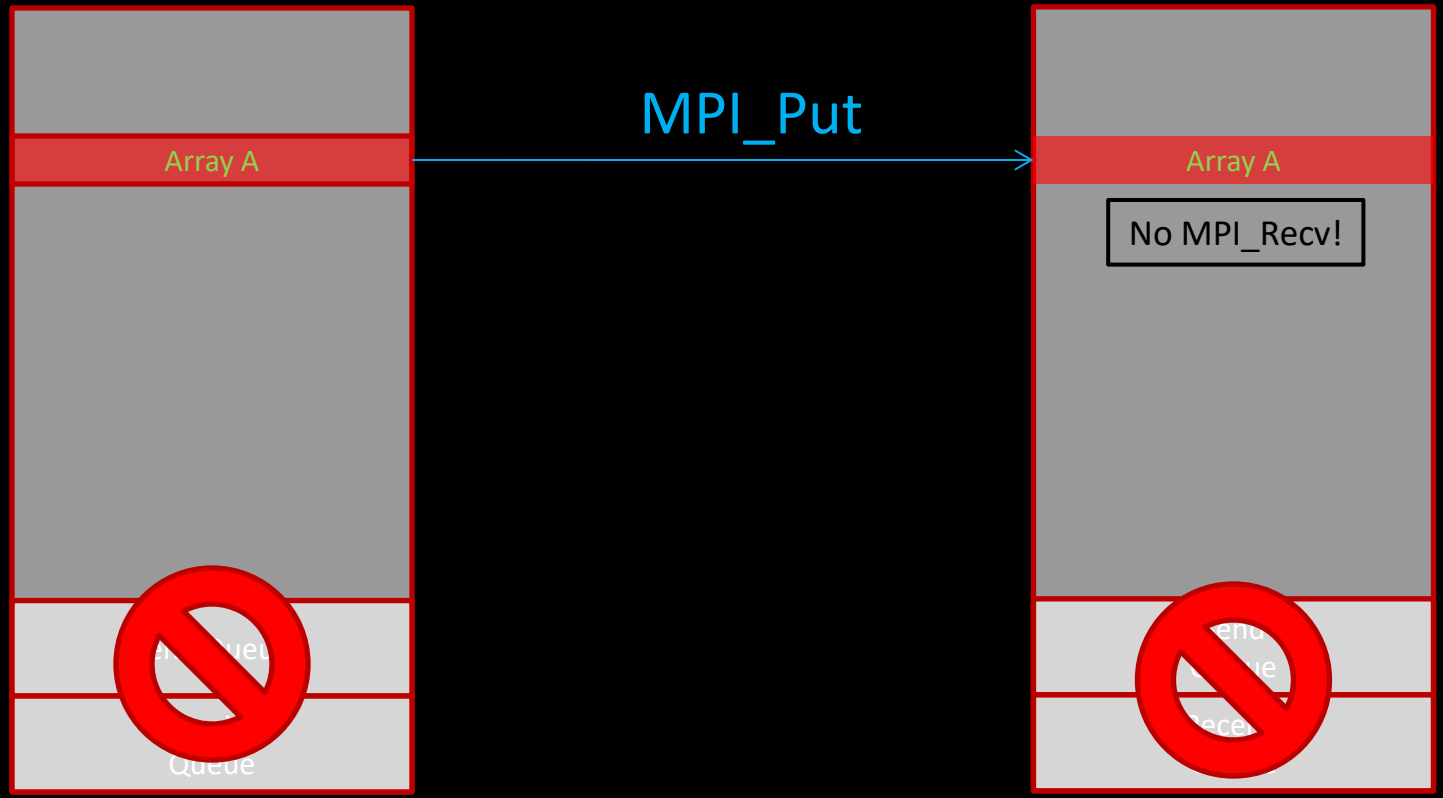
What is the advantage of single-sided communications?

- Performance. Most of the time we can envision our problem as simply a distribution of data that needs to get from here to there during the communication phase. All the ordering, buffering and synchronization that message *passing* implies is just overhead.

- Algorithmic. What if I have a global counter somewhere that gets sporadically updated by random PEs? Perhaps it tracks overflows in the data on all the PEs. How to I do this with cooperative message passing? Do I have to dedicate a PE to poll for random messages? Shouldn't this be simpler?

The directness of Put

Indeed, singled-sided communications has the ultimately direct capability of just cramming data into another PE at will. This has the previously mentioned advantages. Let's consider some of our obligations when using this method.



Remote Memory Access

MPI refers to these types of operations as Remote Memory Access (RMA). It requires three processes:

1. Define data windows
2. Move Data
3. Know when it has finished moving

We'll examine each of these, including their important variations. But first, let's just look at them in action.

Finding Pi with Single-sided Communication

Adapted from the excellent Argonne MPI Pages (which you should peruse)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int n, myid, numprocs, i;
   double PI25 = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x;
   MPI_WIN nwin, piwin;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   //Real window only on PE 0
   if (myid == 0) {
     MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(&pi,sizeof(double), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }
   //But Win_create is collective operation so other PEs must still participate
   else {
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }

   while (1) {
       if (myid == 0) {
           printf("Enter the number of intervals: (0 quits) ");
           scanf("%d",&n);
                   pi = 0.0;
       }

       MPI_Win_fence(0, nwin);
       if (myid != 0)
            MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
       MPI_Win_fence(0, nwin);

       h   = 1.0 / (double) n;
       sum = 0.0;
       for (i = myid + 1; i <= n; i += numprocs) {
           x = h * ((double)i - 0.5);
           sum += (4.0 / (1.0 + x*x));
       }
       mypi = h * sum;

       MPI_Win_fence(0, piwin);
       MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM, piwin);
       MPI_Win_fence(0, piwin);

       if (myid == 0)
           printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25));

   }

   MPI_Win_free(&nwin);
   MPI_Win_free(&piwin);
   MPI_Finalize();

}
```

Creating Windows

With RMA we must carefully define the target regions for remote accesses. This is a collective operation.

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

```
MPI_Win_free(MPI_Win *win)
```

This is the area that we are permitting RMA to access on our own local PE. We will be using displacements to refer to locations within the windows and we define those here.

If you need to allocate memory for the window after it has been created, you also have these new routines, which we won't get into:

```
MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
MPI_Win_detach(MPI_Win win, const void *base)
```

Moving Data

The routines used to move the data are pretty basic:

```
MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
                           int target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
                           int target_count, MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
                           int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

For accumulate the operations can only be the predefined ones, and the basic *components* of the datatypes must be all the same predefined types (vector of MPI_INT fine).

There are versions of these routines that return request flags as well.

```
MPI_Rput(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
           int target_count, MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
MPI_Rget(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
           int target_count, MPI_Datatype target_datatype, MPI_Win win, MPI_Request *request)
MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
                           int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Request *request)
```

MPI_Win_fence

The most basic method for managing when it is safe to access data in the window is the fence routine:

```
int MPI_Win_fence(int assert, MPI_Win w
```

*Stricter than necessary, but an excellent guide until you understand the intricacies.

Ex: If there are no put operations on the window in the fenced region, you could have both gets (from other PEs) and local loads/reads.

It is collective over all the PEs in the window and completes any operations since the las
data is updated both to and from the window.

The easy rule* is to:

- use `MPI_Win_fence` to separate the RMA and local access (computation) sections of code.
- do not overlap RMA operations (access the same section of any window) within one of these fenced regions.

The available `assert` options allow for possibly substantial optimization.

MPI_MODE_NOSTORE - the local window was not updated by local stores (or local get or receive calls) since last synchronization.
MPI_MODE_NOPUT - the local window will not be updated by put or accumulate calls after the fence call, until the ensuing fence.
MPI_MODE_NOPRECEDE - the fence does not complete any sequence of locally issued RMA calls. Must be given by all PEs in the group.
MPI_MODE_NOSUCCEED - the fence does not start any sequence of locally issued RMA calls. Must be given by all PEs in the group.

Separate RMA and local access

MPI_Win_fence clearly separates these two regions.

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int n, myid, numprocs, i;
   double PI25 = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x;
   MPI_WIN nwin, piwin;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   //Real window only on PE 0
   if (myid == 0) {
     MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(&pi,sizeof(double), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }
   //But Win_create is collective operation so other PEs must still participate
   else {
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }

   while (1) {
       if (myid == 0) {
           printf("Enter the number of intervals: (0 quits) ");
           scanf("%d",&n);
                    pi = 0.0;
       }

       MPI_Win_fence(0, nwin);
       if (myid != 0)
            MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
       MPI_Win_fence(0, nwin);

       h   = 1.0 / (double) n;
       sum = 0.0;
       for (i = myid + 1; i <= n; i += numprocs) {
           x = h * ((double)i - 0.5);
           sum += (4.0 / (1.0 + x*x));
       }
       mypi = h * sum;

       MPI_Win_fence(0, piwin);
       MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM, piwin);
       MPI_Win_fence(0, piwin);

       if (myid == 0)
           printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25));

   }

   MPI_Win_free(&nwin);
   MPI_Win_free(&piwin);
   MPI_Finalize();

}
```

Passive Target RMA

RMA as we've used it thus far is not truly "single-sided" as the target PEs have to participate in the MPI_Win_fence calls. They are collective, and the originating PEs can't use any of the transferred data until they make this call.

MPI has another mode that eliminates this requirement, and the target PE is truly passive. This starts to look a lot like a true SMP, where one core can just access any memory location unbeknownst to the other cores.

The main refinement here is that instead of synchronizing all the windows in a collectively defined fenced region, we will target a specific window on a single PE for a period controlled by locks. The rules we used to separate local access and to not overlap operations apply to these locked regions the same way.

MPI_Win_lock

Our more targeted approach specifies a target PE:

```
MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_Win_unlock(int rank, MPI_Win win)
```

The lock types can be:

```
MPI_LOCK_SHARED – allow other operations on this window.
MPI_LOCK_EXCLUSIVE – do not.
```

Assert is an optimization flag. We will default to 0.

Use these appropriately. Shared may be fine with non-overlapping regions of the window, or even accumulate calls to the same region.

Note that a PE should usually lock its *own* window when accessing it.

MPI_Win_allocatae

MPI allows implementations to demand that *passive* memory windows are allocated in specific ways. MPI has a number of memory allocation routines, but the most straightforward for this purpose is simply:

```
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
                 void *baseptr, MPI_Win *win)
```

size and disp are in bytes. info is for implementation hints and NULL is always valid.

Fortran 2008 has a different recommended procedure. Older Fortran has some other specific concerns, which are easily dealt with, but not on this slide.

Passive Counter

We can now implement that randomly updated counter that we mentioned before. It is as simple as:

```
…
value = 1;
MPI_Win_lock( MPI_LOcK_EXCLUSIVE, rank, 0, win );
MPI_Accumulate( &value, 1, MPI_INT, rank, 0, 1, MPI_INT, MPI_SUM, win );
MPI_Win_unlock( rank, win );
…
```

Simple enough. What if we want to leverage this code to create a mutex lock. That can be quite useful when trying to share global resources amongst the PEs.

We will use a classic mutex technique where we just read/add a counter that rests at 0. If it is 1, then we know we just got the resource and later we can free it by decrementing.

If it is not 1, then we decrement it back and try again later.

Passive Mutex

```
…
value = 1;
MPI_Win_lock( MPI_LOcK_EXCLUSIVE, rank, 0, win );
MPI_Accumulate( &value, 1, MPI_INT, rank, 0, 1, MPI_INT, MPI_SUM, win );
MPI_Get( &counter, 1, MPI_INT, rank, 0, 1, MPI_INT, win );
MPI_Win_unlock( rank, win );
…
```

It is actually quite difficult to do this with the tools we have, and this situation is familiar to anyone doing multi-threaded coding. Here, as there, the answer to is have nice atomic operations to facilitate all these kinds of things.

MPI_Fetch_and_op

MPI provides the very targeted atomic operation:

```
MPI_Fetch_and_op(void *origin_addr, void *result_addr, MPI_Datatype datatype,
                 int target_rank, MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

which is efficient and makes our mutex trivial, and the more general:

```
MPI_Get_accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
                   void *result_addr, int result_count, MPI_Datatype result_datatype,
                   int target_rank, MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win
```

There is also an atomic operation that is quite useful in distributed data structures like lists.

```
MPI_Compare_and_swap(void *origin_addr, const void *compare_addr, void *result_addr,
                   MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)
```

This function compares one element of type datatype in the compare buffer compare_addr with the buffer at offset target_disp in the target window specified by target_rank and win and replaces the value at the target with the value in the origin buffer origin_addr if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer result_addr.

A few more useful synch routines

There are a number of related synchronization routines that allow for finer control (i.e. fancier than our rule of thumb). Note the subtle distinctions. Most of the useful ones are:

`MPI_Win_lock_all(int assert, MPI_Win win)`

Starts an RMA access epoch to all processes in win, with a lock type of MPI_Lock_shared. During the epoch, the calling process can access the window memory on all processes in win by using RMA operations. A window locked with MPI_Win_lock_all must be unlocked with MPI_Win_unlock_all. This routine is not collective, the all refers to a lock on all members of the group of the window.

`MPI_Win_flush(int rank, MPI_Win win)`

MPI_Win_flush completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.

`MPI_Win_sync(MPI_Win win)`

The call MPI_Win_sync synchronizes the private and public window copies of win. For the purposes of synchronizing the private and public window, MPI_Win_sync has the effect of ending and reopening an access and exposure epoch on the window (note that it does not actually end an epoch or complete any pending MPI RMA operations).

`MPI_Win_flush_all(MPI_Win win)`

All RMA operations issued by the calling process to any target on the specified window prior to this call and in the specified window will have completed both at the origin and at the target when this call returns.

Scalable Synchronization RMA

There is one last approach that is targeted at very large scalability. It assumes an active target (like our fence approach). But the routines that replace fence are not collective. They are called only on the origin and target PEs. Thus the synchronization can be confined to neighborhoods and be as scalable as the application.

These routines work with groups. We can use all the usual group management routines to define our groups, and we probably start with `MPI_Win_get_group(MPI_Win win, MPI_Group *group)` to get a group from our window – which will be the same as the group of the communicator used to create the window.

Scalable Sync RMA Routines

The idea here is that each end defines the period where it is either exposing or targeting a window.

The target PE indicates when its windows may be accessed with:

```
MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
…
…
MPI_Win_wait(MPI_Win win)
```

the originating PE uses:

```
MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
…
…
MPI_Win_complete(MPI_Win win)
```

Like fence, there are some asserts to aid with optimization (`MPI_MODE_NOCHECK`,`MPI_MODE_NOSTORE`,`MPI_MODE_NOPUT`).

Within these regions we use put, get, etc. the same as we would elsewhere.

MPI Communication Hierarchy

Let's review all of the mechanisms available. Ranked by a general scalability and efficiency.

1. Blocking Message Passing
2. Non-blocking Message Passing
3. RMA Active Target ("fence")
4. RMA Passive Target ("Win_lock")
5. RMA Scalable Synchronization ("Win_post")

It is important to realize that you can start with your preferred level and often achieve near-optimal results. You can choose the more sophisticated options piecemeal as you go.

Hybrid Programming

(Most “complex” version: MPI_THREAD_MULTIPLE)

```
#include <mpi.h>
#include <omp.h>

//Last thread of PE 0 sends its number to PE 1

main(int argc, char* argv[]){

   int provided, myPE, thread, last_thread, data=0, tag=0;
   MPI_Status status;

   MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
   MPI_Comm_rank(MPI_COMM_WORLD, &myPE);

   #pragma omp parallel firstprivate(thread, data, tag, status)
   {
        thread = omp_get_thread_num();
        last_thread = omp_get_num_threads()-1;

        if ( thread==last_thread && myPE==0 )
             MPI_Send(&thread, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
        else if ( thread==last_thread && myPE==1 )
             MPI_Recv(&data, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);

        printf("PE %d, Thread %d, Data %d\n", myPE, thread, data);
   }

   MPI_Finalize();
}
```

```
% export OMP_NUM_THREADS=4
% ibrun -np 3 a.out
PE 0, Thread 0, Data 0
PE 1, Thread 0, Data 0
PE 2, Thread 0, Data 0
PE 2, Thread 3, Data 0
PE 0, Thread 3, Data 0
PE 1, Thread 3, Data 3
PE 0, Thread 2, Data 0
PE 2, Thread 2, Data 0
PE 1, Thread 2, Data 0
PE 0, Thread 1, Data 0
PE 1, Thread 1, Data 0
PE 2, Thread 1, Data 0
```

Output for 4 threads run on 3 PEs

MPI-IO

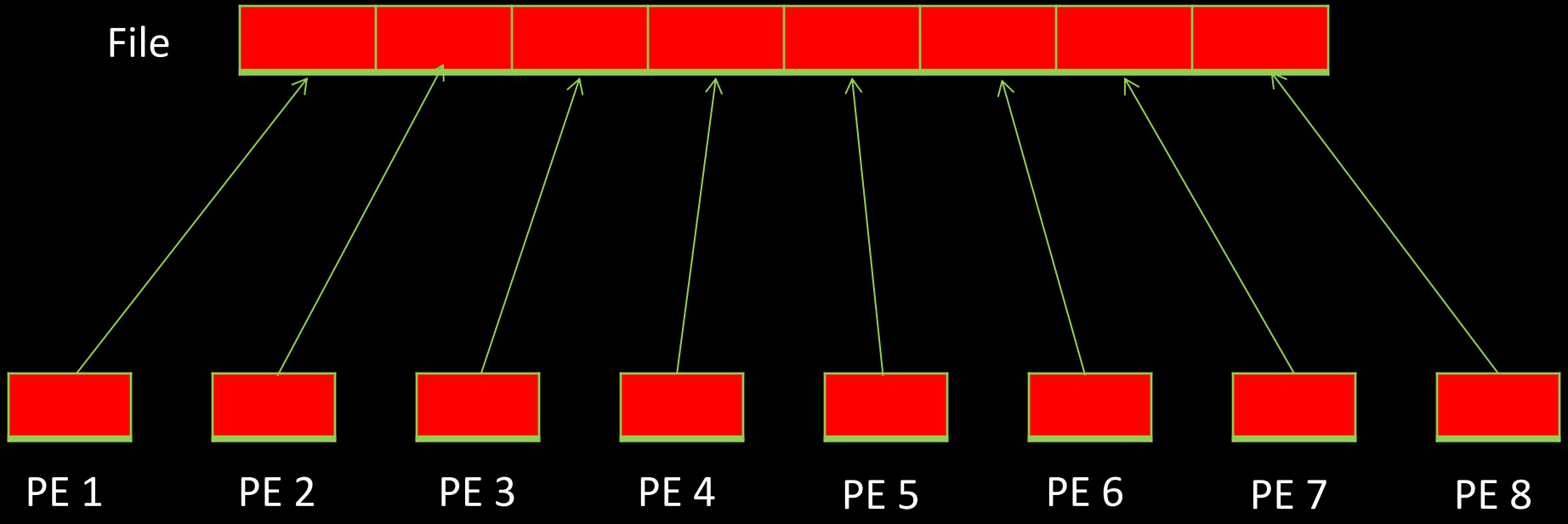
MPI-2 Introduced routines, usually referred to as “MPI_IO”, building on the MPI concepts of:

- Collective operations (to combine IO)
- Datatypes (new “distributed” datatypes to describe data scattered over PEs)
  - Skip unneeded data (ex. ghost zones)
  - Regroup data (subarrays)
- Non-blocking operations (to enable asynchronous/background IO)
- Portable file formats (based on standard MPI datatypes)

A great implementation could leverage all of these to do smart buffering and coalescing and get great performance. Unfortunately, this is very hard to do.

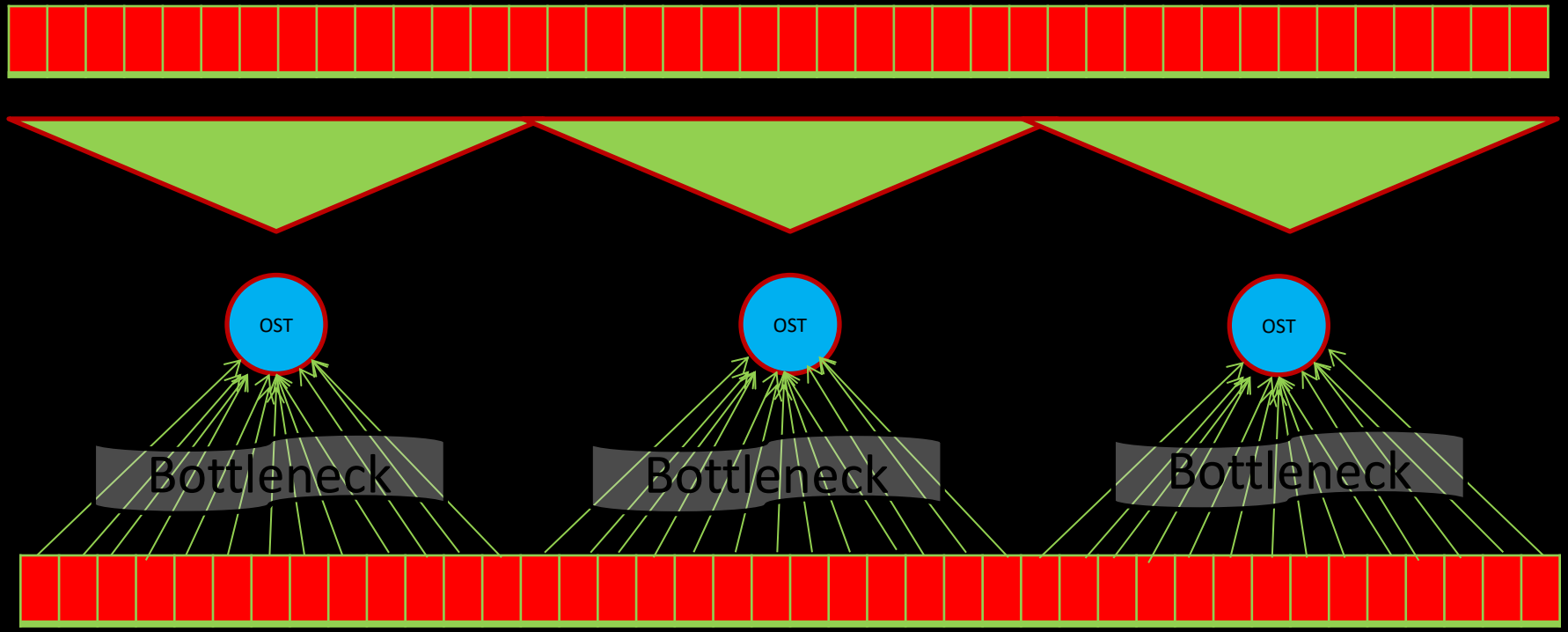
Parallel IO

A baseline IO operation for an parallel code might be something like the below for checkpointing a very regular data structure. A more difficult case might be with every PE doing irregular and random data reads and writes. But even this case is sufficient to demonstrate a typical IO bottleneck.



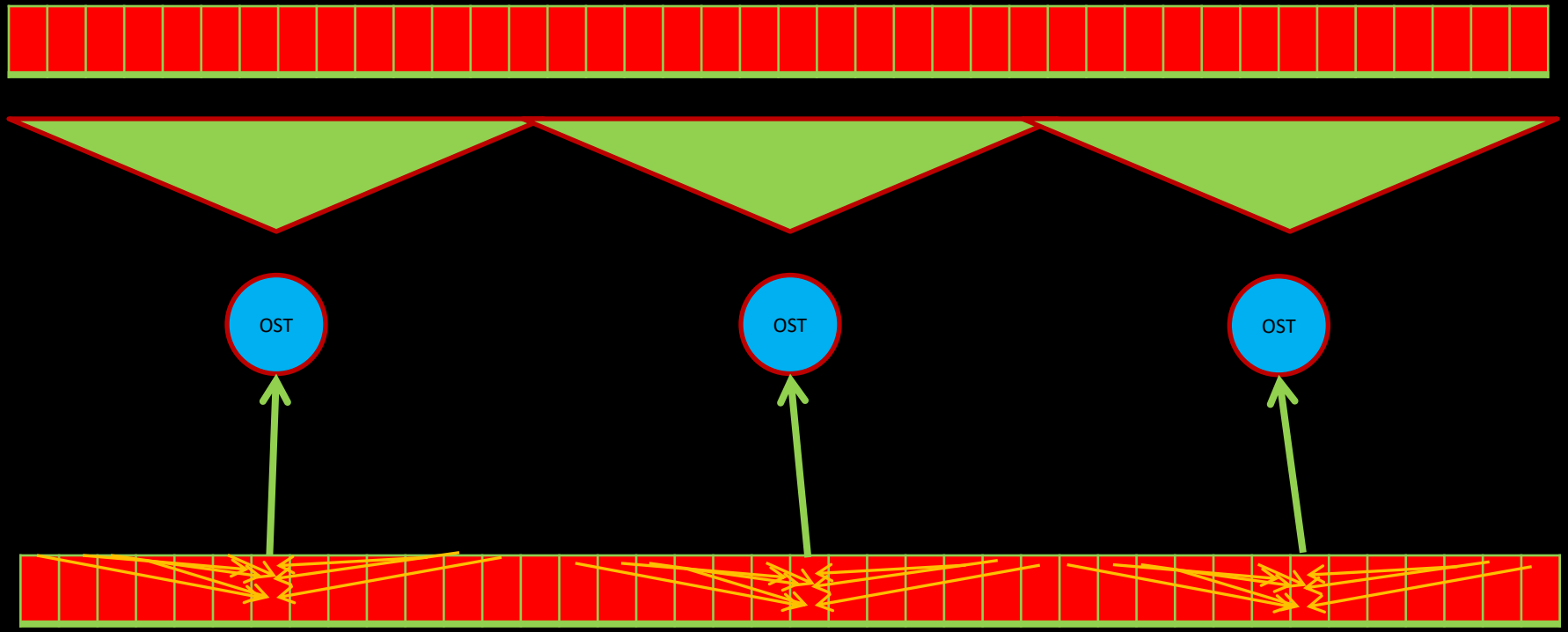
Parallel Disk System

A physical disk system looks something like the below (a simplified Lustre):



Optimized Parallel IO

Often, given the specific bandwidths and other limits, the optimal solution will look like:



MPI IO Answer

In practice, MPI-IO is often not able to reflect these realities. Fortunately, only a few parameters are usually required to comprehend common IO configurations. In particular, the number of PEs to IO channel ("OST") or likewise the optimal number of simultaneous reads/writes.

Also fortunately, this adaption is usually easily added into MPI codes as:

- IO is often already isolated in a few sections of code
- MPI is natural at handling the internal communications
- POSIX IO works fine at the output end

We do not want to diminish the potential for other issues. Writing irregular data that can dump and restart on varying numbers of PEs can take some thought. Just don't commit to an MPI-IO abstraction to handle everything unless you can afford to back off if performance issues arise.

Fortran 90

I hope any Fortran programmers out there are using modern Fortran, and Fortran 2003 and 2008 bring a lot to the table. Using mpi_f08 adds some benefits (can leave off error parameter, for one). However, there is one obviously useful F90 syntactical nicety that has a subtle problem:

```
real, dimension (ROWS,COLUMNS) :: temperature
…
call MPI_Isend(temperature(1,1:COLUMS), COLUMNS, MPI_REAL,…
```

This seems like a perfectly reasonable way to send a row without getting involved with MPI datatypes. However what is happening is that Fortran is allocating a contiguous buffer that it sends to the MPI routine. So far, so good. But, as far as the compiler can tell, that buffer can be reclaimed right after that call. It does not know that it shouldn't disturb this until some request flag is set later. End result: possibly sending deallocated memory.

MPI-3 allows one to test if the compiler is savvy about this (MPI_SUBARRAYS_SUPPORTED = .TRUE. or using ASYNCHRONOUS variables). Stick with MPI datatypes if you are unsure, although the current compiler situation is continuously improving.

C++

- MPI committee deprecated the C++ binding in MPI 2.2. It was removed in MPI 3.0
- Poor overall integration in the C++ environment (especially the STL)
- Alternative: Boost.MPI
  - world.send(1, 1, 4055); (rank,tag,data)
  - Possible performance penalty for anything variable-length: std::string, std::vector, std::map, lists, polymorphic types. These can't be MPI datatypes at all, so Boost has to serialize them.
  - No streams
- A shame as a lot of MPI routine parameters can be either inferred by the compiler (type/size of sent/received data) or defaulted (tag = 0, comm = MPI_COMM_WORLD, status = MPI_STATUS_IGNORE). But, Boost does hide some of this pain.

Python

Python lacks effective threading, so multi-processing is the only real option for scalability. The most common MPI bindings do provide for some elegant coding. Here is our "finding Pi" program. Notice that the routines return the data, instead of using buffer parameters.

```
from mpi4py import MPI
from numpy  import arange
from math   import sqrt

comm = MPI.COMM_WORLD
my_pe_num = comm.Get_rank()

intervals = 0
if my_pe_num == 0:
    intervals = int(input("How many intervals? "))

intervals = comm.bcast(intervals, root=0)

mypi  = 0
h     = 2/intervals
start = ( my_pe_num   *2  / comm.Get_size()) -1
end   = ((my_pe_num+1)*2  / comm.Get_size()) -1

for x in arange(start, end, h):
    mypi = mypi + h * 2* sqrt(1-x*x)

pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if my_pe_num == 0:
    print("Pi is approximately %f" % pi)
    print("Error is %f" % (pi-3.14159265358979323846));
```

Python with NumPy

As with many things Python, you really want to use the NumPy libraries. In which case you can gain some performance back (keep your expectations reasonable) with the Upper Case, NumPy routines, which use in-place buffers, like "normal" MPI.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
my_pe_num = comm.Get_rank()

.
.
.

    # send bottom real row down
    if my_PE_num != NPES-1:
        comm.Send( temperature[ ROWS , 1:COLUMNS+1 ], dest=my_PE_num+1, tag=DOWN)

    # receive the bottom row from above into our top ghost row
    if my_PE_num != 0:
        comm.Recv( temperature_last[ 0 , 1:COLUMNS+1 ], source=my_PE_num-1, tag=DOWN)

    .
    .
    .
```

Other Languages

Bindings are available for at least:

- Perl
- Julia
- R
- Ruby
- Java
- .NET/C#
- Rust

Attributes

The MPI committee has decided the algorithmic utility of a small key store database is useful enough that they have added such a capability to three classes of objects:

- Communicators
- Datatypes
- RMA Windows

For each of these there are routines to:

- Create Key
- Free Key
- Set Attribute
- Get Attribute
- Delete Attribute

One can then store and retrieve various values. These capabilities are particularly useful for writing libraries. There are also some useful pre-defined attributes which you may notice in the documentation.

Dynamic Process Management

This paradigm can seem tempting, but as most MPP's require you to run on a specific number of PE's, adding tasks can often cause load balance problems – if it is even permitted. For optimized scientific codes, this is rarely a useful approach.

It can be nice for dynamic work farming algorithms, but these have also often been implemented in MPI 1.x styles with fairly simple queue mechanisms.

Note that MPI 6 may change this situation a bit.

Dynamic Process Management

MPI_Comm_spawn creates a new group of tasks and returns an intercommunicator:

```
MPI_Comm_spawn(command, argv, numprocs, info, root, comm, intercomm, errcodes)
```

- Tries to start *numprocs* process running *command*, passing them command-line arguments *argv*.
- The operation is collective over *comm*.
- Spawnees are in remote group of *intercomm*.
- Errors are reported on a per-process basis in *errcodes*.
- *info* used to optionally specify hostname, archname, wdir, path, file.

MPI_Wtime: trivial but awesome

This is about as simple as a routine gets, but it brings some much needed portability as a useful high-resolution timer. You will find yourself using it often.

```
double start_time, total_time;
start_time = MPI_Wtime();
.
....  stuff being timed  ...
.
total_time = MPI_Wtime() – start_time;
```

MPI_Wtick() will return the timer resolution. This is usually microseconds or better on HPC platforms.

MPI 4

MPI 4 (released in 2021) introduced some additional features. The major ones were:

Large Counts: routine (*MPI_Send_c*) that replaced int with *MPI_Count* to accommodate Exascale size problems.

Sessions: Process sets to let libraries initialize MPI independently (*MPI_Session_init*) of the application (no global *MPI_COMM_WORD*).

Partitioned Communication: (*MPI_Psend_init*) aimed at multithreaded and GPU senders where each thread contributes part of a buffer).

Persistant Collectives: (*MPI_Bcast_init*) to enable the same kind of convenience we saw with *MPI_Send_init* earlier.

There were also many other minor additions, as well as an interim 4.1 update in 2023, with its own improvements.

MPI 5

MPI 5 (released in 2025) was a different kind of release. It focused on standardizing the ABI, so that different MPI versions – mainly the MPICH (MPICH, Intel MPI, MVAPICH, Cray) and Open MPI families - could be binary compatible.

This is less about performance than about allowing other tools (like debuggers and profilers) and language bindings to work effectively.

MPI 6

MPI 6 has already begun serious discussion. The notable improvements will be

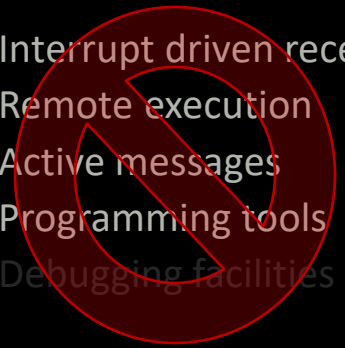
- Partitioned communication
- Hybrid/accelerator support including GPU bindings
- Dynamic resource management via Sessions
- Fault tolerance
- Revamped I/O and RMA

MPI's Widespread Adoption

The widespread adoption of MPI is not an accident. The committee has done a great job of resisting trendy features that can't be universally supported.

The core philosophy is that "the standard does not specify operations that require more operating system support than is currently standard."

- Interrupt driven receives
- Remote execution
- Active messages
- Programming tools
- Debugging facilities



Which features to use?

As of MPI 5 there are over 450 routines in the library. How do you know where to start?

1. Design your algorithm using the send/recv/isend/irecv family. In the uncommon event that you find real algorithmic limitations with the capability there, then look deeper into more advanced options.

2. After you have turned your algorithm into working code, you might find that one-sided communication is a useful optimization. You can migrate your code to that level without a complete revision.

3. If you are ready to target 100K+ processors, then MPI 4 or 5 has useful capabilities. And you won't find it intimidating at that point.

Of course this is just a general guide. You may find a more advanced feature central to your needs, or you may wish to forego any unnecessary refactoring and just target a final optimized version. The key concept is that for the vast majority of codes, *once you have your data decomposition, the communication can be incrementally optimized*. And once you have experience with the MPI-1 level routines, you will find the additional routines to be familiar variations on known themes.

Pop Quiz

Here is a good test of MPI understanding. What is the result of this code on 8 PEs?

```
if (rank % 2 == 0) // Even processes
{
  MPI_Allreduce(&rank, &evensum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  if (rank == 0) printf("evensum = %d\n", evensum);
}
else // Odd processes
{
  MPI_Allreduce(&rank, &oddsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  if (rank == 1) printf("oddsum = %d\n", oddsum);
}
```

A) evensum=16 oddsum=12

B) evensum=12 oddsum=16

C) evensum=28 oddsum=28

D) evensum=6 oddsum=22

From the 2015 International HPC Summer School