www.bsc.es

Barcelona Supercomputing Center

Centro Nacional de Supercomputación

MPI: Non-Blocking Communication, Collective Communication, Datatypes

Marc Jordà, Antonio J. Peña

Montevideo, 21-25 October 2019

What will be covered in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **More advanced topics:**
  - **Non-blocking communication**, collective communication, datatypes
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

Blocking vs. Non-blocking Communication

- `MPI_SEND/MPI_RECV` are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For “send” completion implies variable sent can be reused/modified
    - Modifications will not affect data intended for the receiver
  - For “receive” variable received can be read
- `MPI_ISEND/MPI_IRECV` are nonblocking variants
  - Routine returns immediately – completion has to be separately tested
  - These are primarily used to overlap computation and communication to improve performance

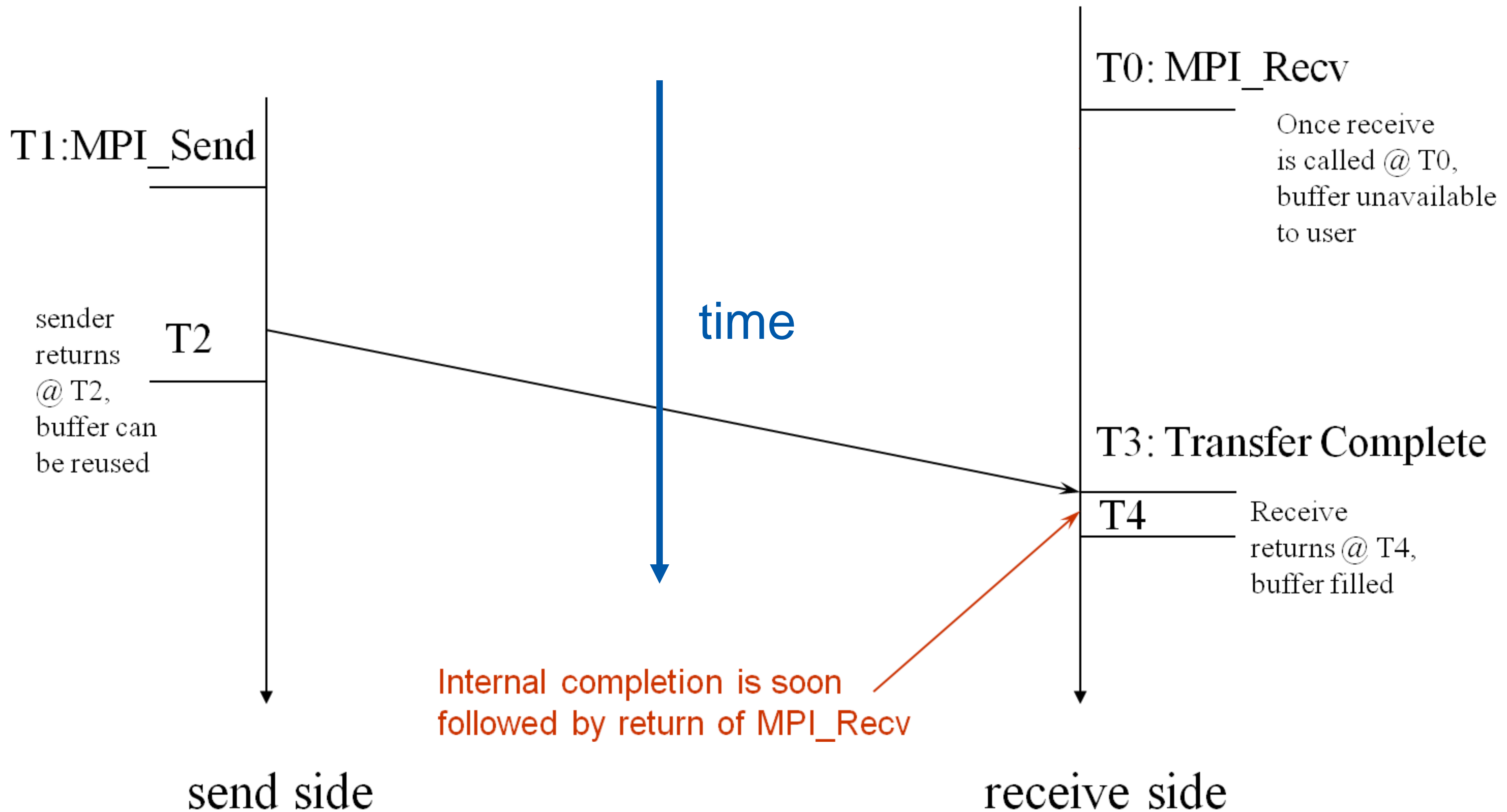
Blocking Communication

- In blocking communication
  - **`MPI_SEND`** does not return until buffer is empty (available for reuse)
  - **`MPI_RECV`** does not return until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is sent
- A process receiving data will be blocked until the receive buffer is filled
- Exact completion semantics of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to **deadlocks**:

```
if (rank == 0) {
        MPI_SEND(..to rank 1..)
        MPI_RECV(..from rank 1..)
else if (rank == 1) {
        MPI_SEND(..to rank 0..)      ← recv before send fixes it
        MPI_RECV(..from rank 0..)
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Blocking Send-Receive Diagram



Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) "request handles" that can be waited on and queried
  - `MPI_ISend(buf, count, datatype, dest, tag, comm, request)`
  - `MPI_IRecv(buf, count, datatype, src, tag, comm, request)`
  - `MPI_Wait(request, status)`

- Non-blocking operations allow overlapping computation and communication

- One can also test without waiting using **MPI_Test**
  - **MPI_Test(request, flag, status)**

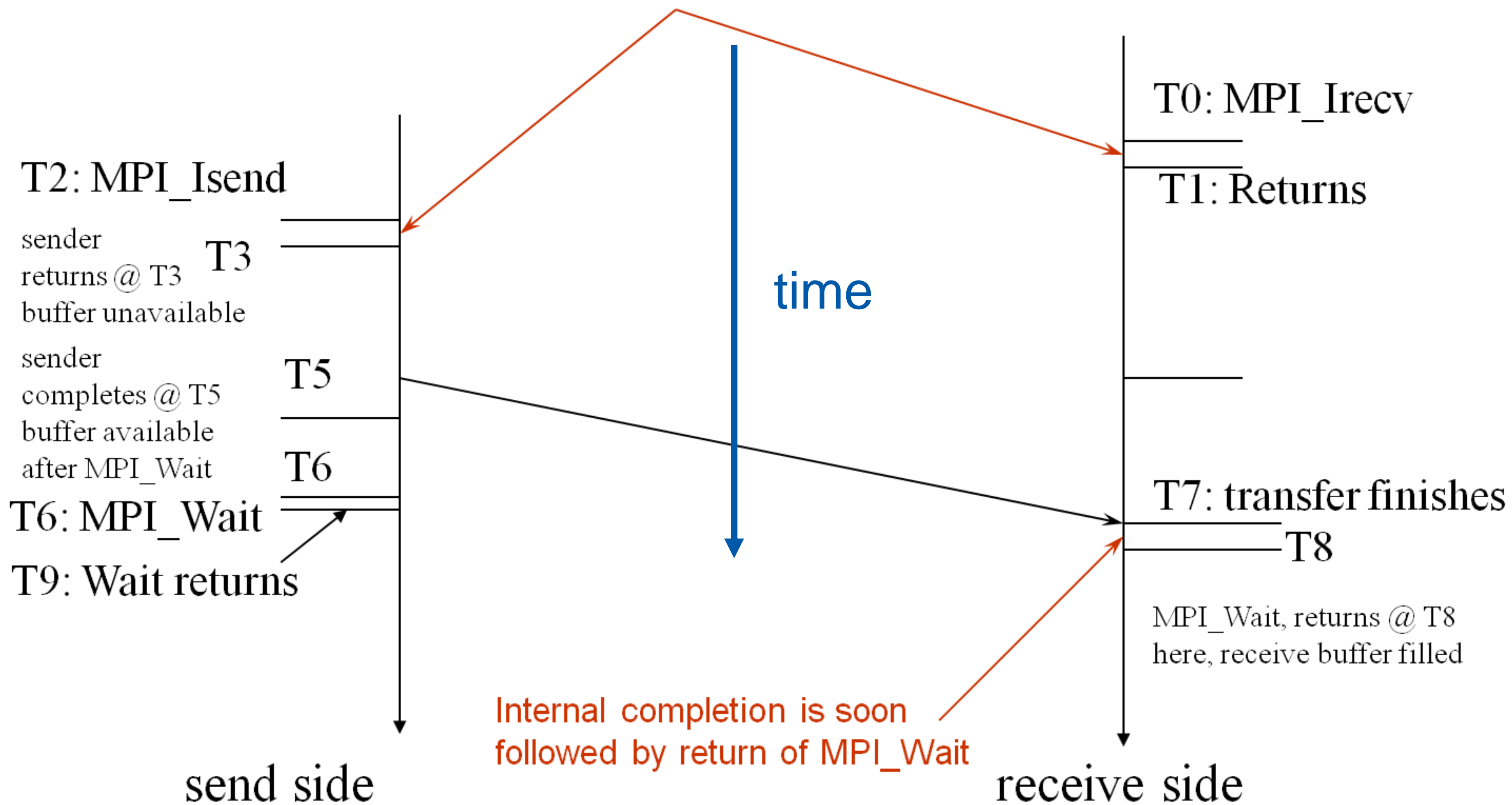
- Anywhere you use **MPI_Send** or **MPI_Recv**, you can use the pair of **MPI_ISend/MPI_Wait** or **MPI_IRecv/MPI_Wait**

Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`
- There are corresponding versions of **TEST** for each of these
  - MPI_Testall, MPI_Testany and MPI_Testsome.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Non-Blocking Send-Receive Diagram



Message Completion and Buffering

- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including MPI_ANY_SOURCE)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

```
*buf = 3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will always
             receive 3 */
```

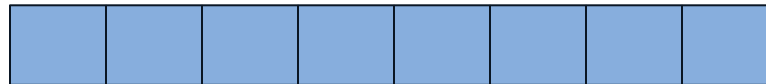
```
*buf = 3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /* Receiver may get 3, 4, or
             anything else */
MPI_Wait(…);
```

- Just because the send completes does not mean the receive completed
  - Message may be buffered by the system
  - Message may still be in transit

A Non-Blocking communication example

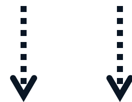
P0

P1

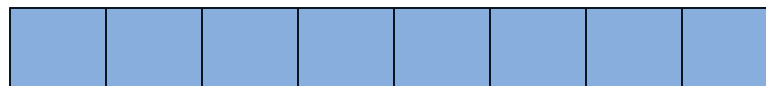


Blocking Communication

P0



P1



Non-blocking Communication

A Non-Blocking communication example

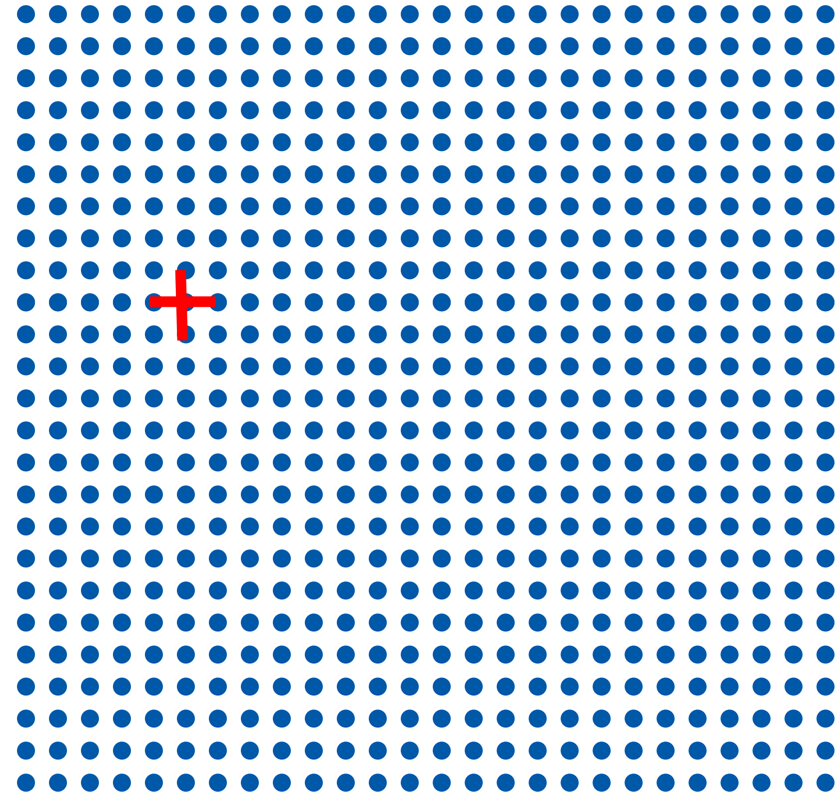
```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            data[i] = compute(i);
            MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                      &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else if (rank == 1){
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

Regular Mesh Algorithms

- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the differential equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same
- Five-point stencil is a popular approximation solution

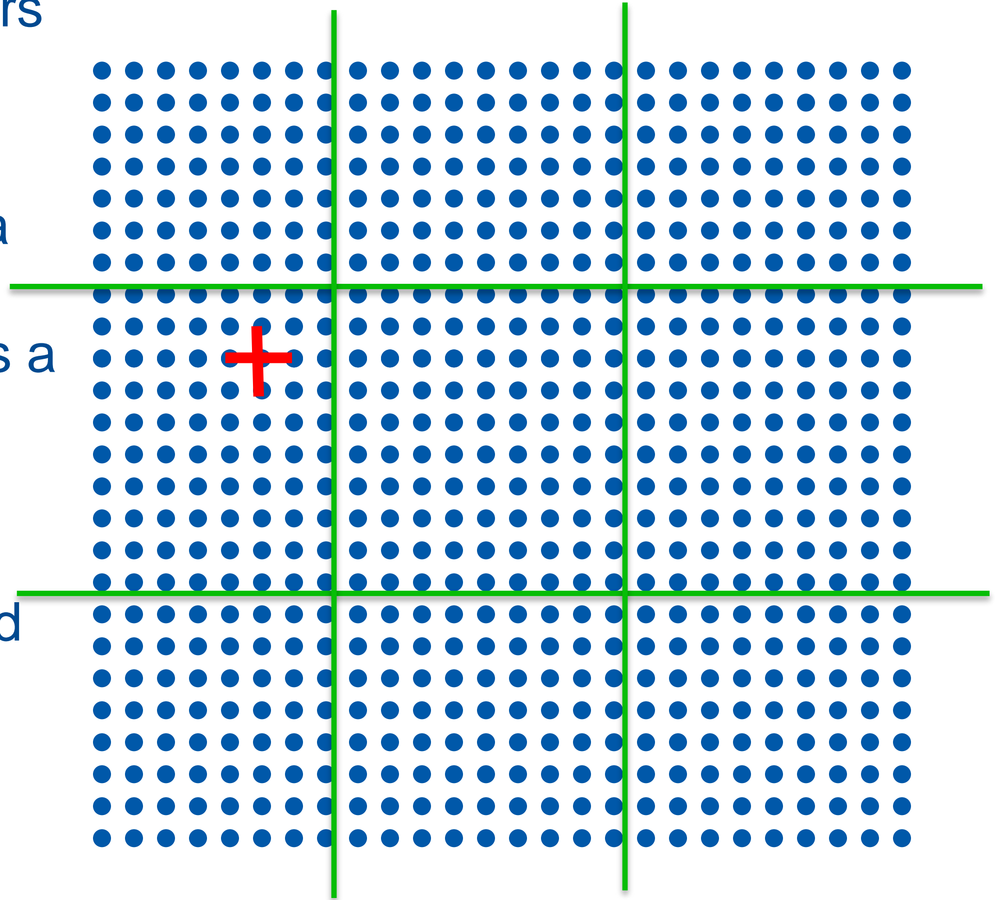
The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the 4 neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.

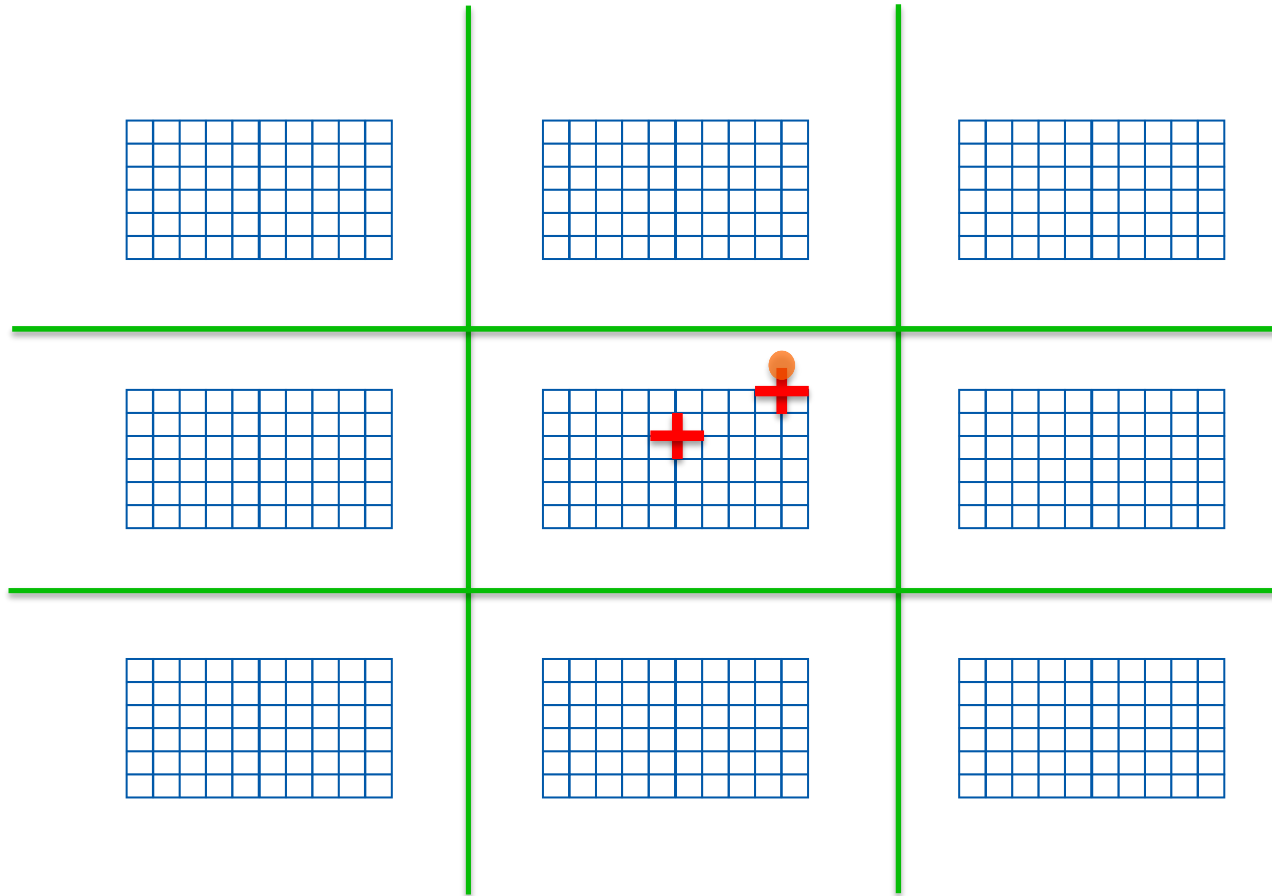


The Global Data Structure

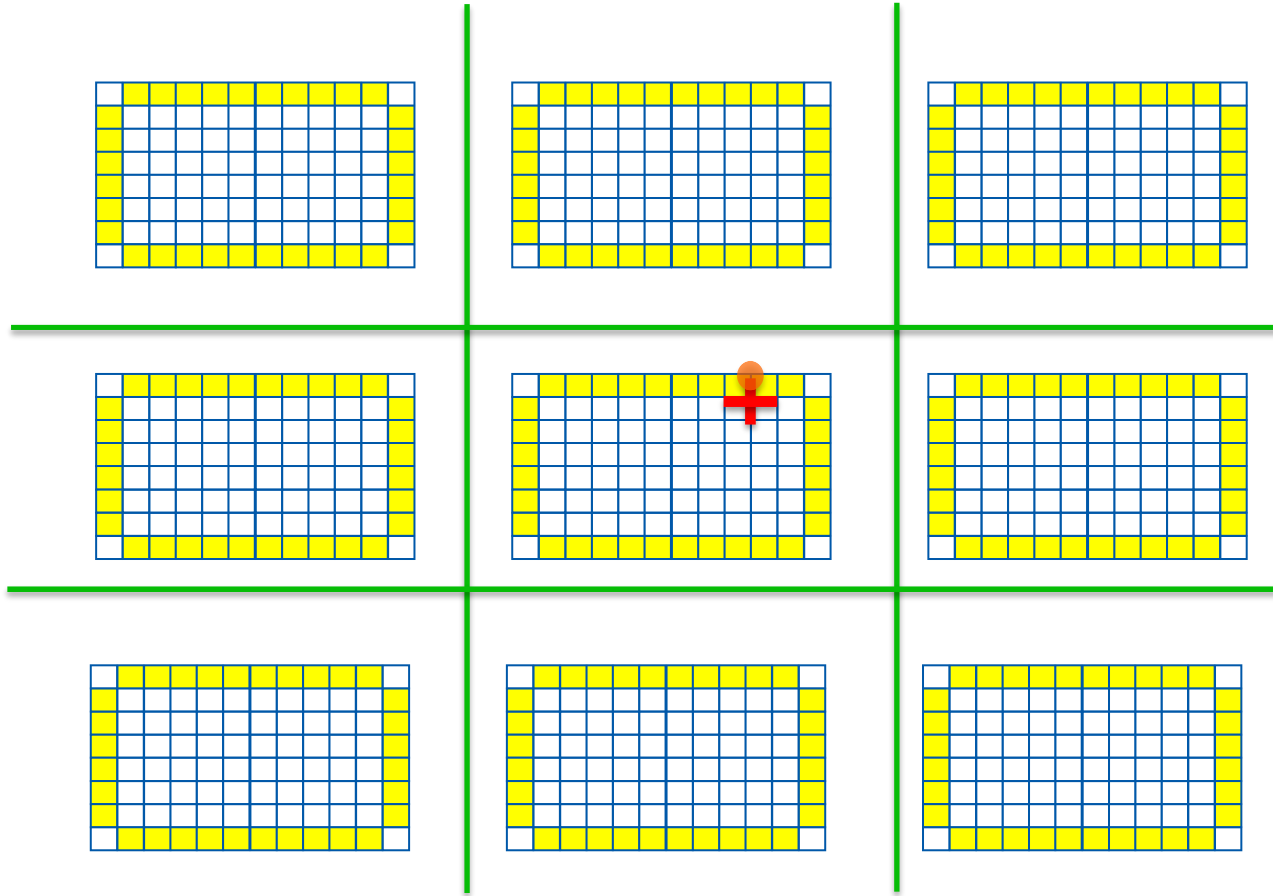
- Each circle is a mesh point
- Difference equation evaluated at each point involves the 4 neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- Decompose mesh into equal sized (work) pieces



Necessary Data Transfers



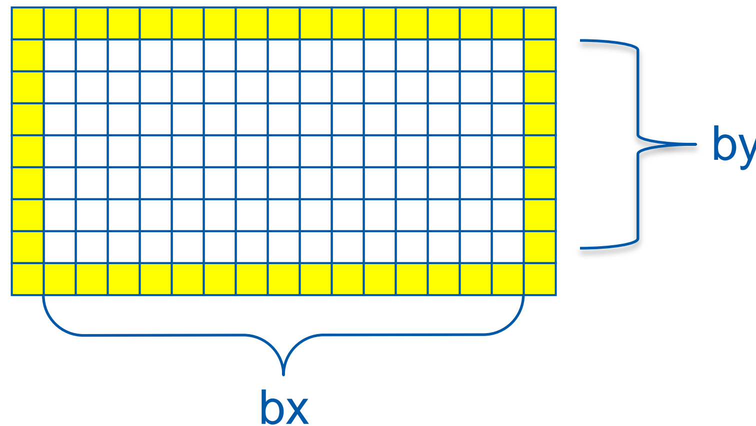
Necessary Data Transfers



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

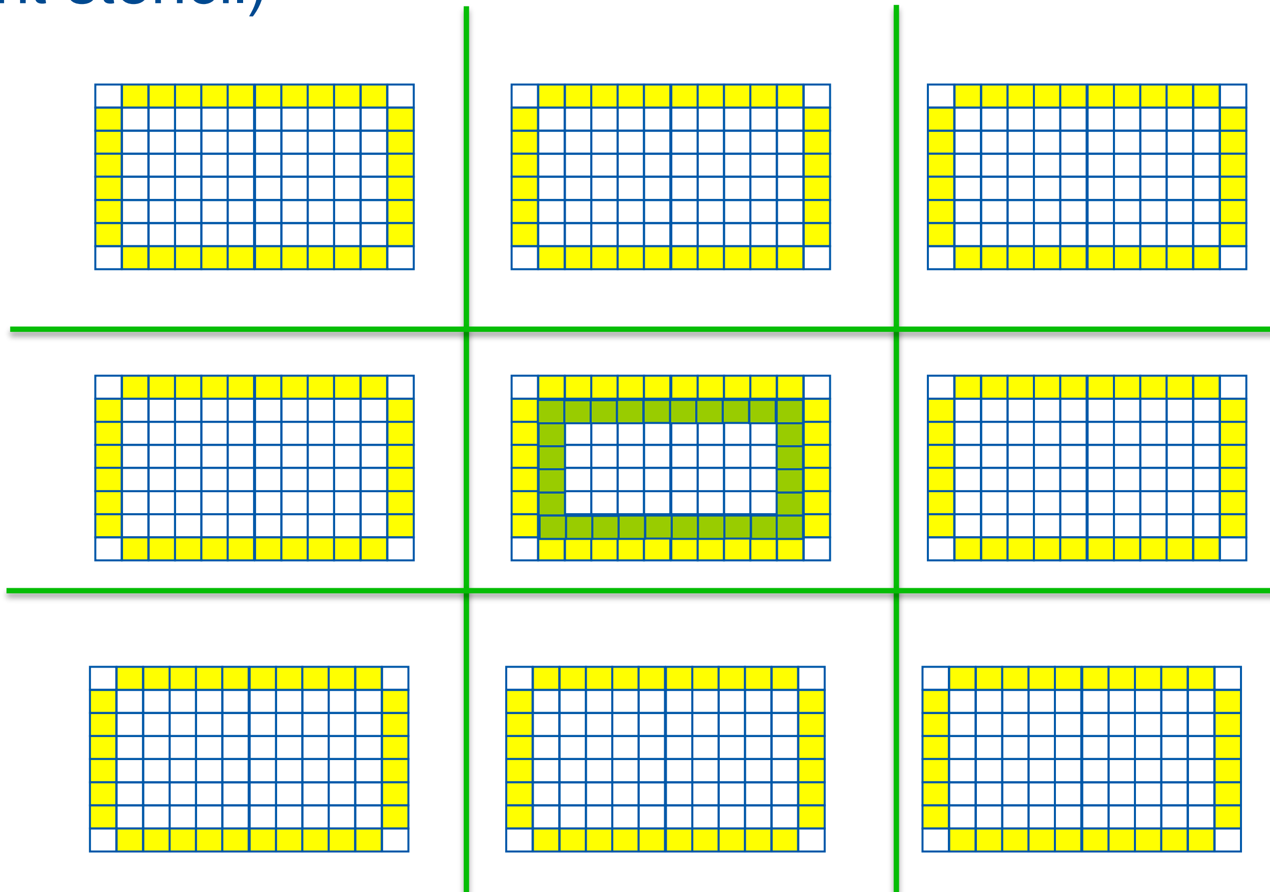
The Local Data Structure

- Each process has its local "patch" of the global array
  - "bx" and "by" are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)



Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)

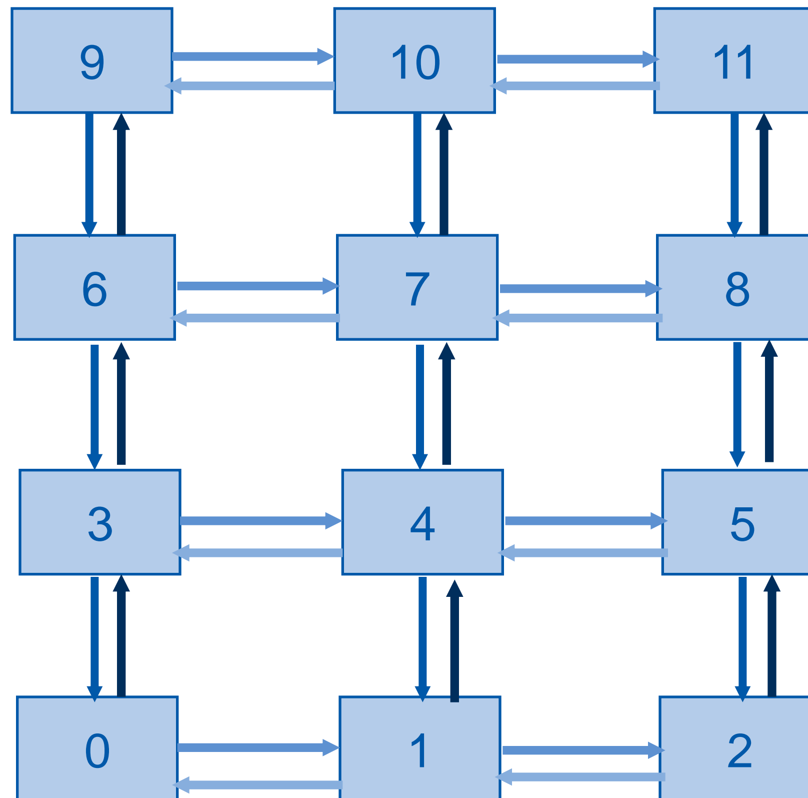


Understanding Performance: Unexpected Hot Spots

- Basic performance analysis looks at two-party exchanges
- Real applications involve many simultaneous communications
- Performance problems can arise even in common grid exchange patterns
- MPI illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable memory stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

```
for (i = 0; i < n_neighbors; i++) {
   MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
   MPI_Recv(edge, len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

- What is wrong with this code?

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has up nbr)
    MPI_Recv( … up … )
…
if (has down nbr)
    MPI_Send( … down … )
```

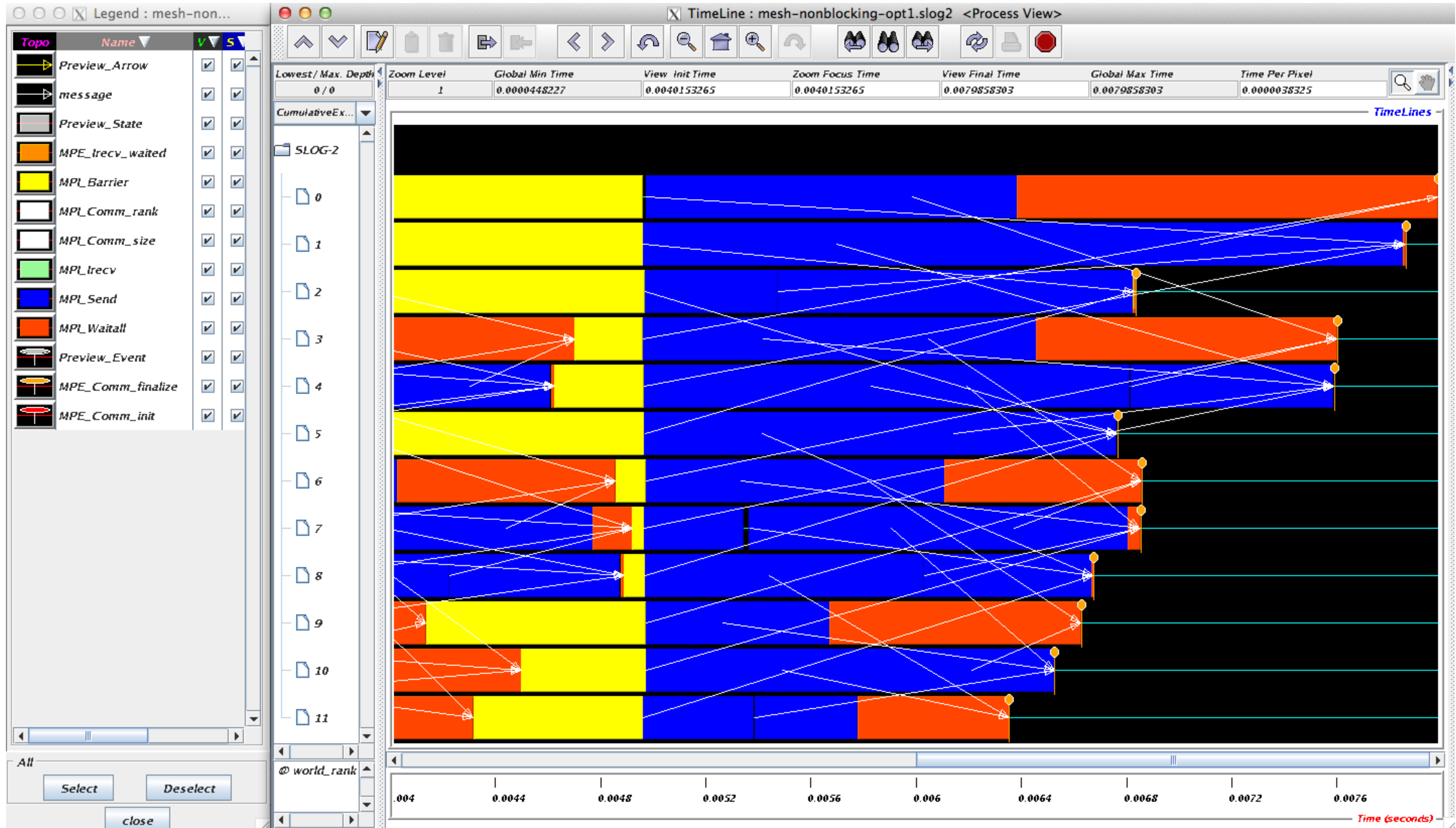
sequentializes (all except the top process block)

Fix 1: Use Irecv

```
for (i = 0; i < n_neighbors; i++) {
        MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
                        comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
        MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
MPI_Waitall(n_neighbors, requests, statuses);
```

- Does not perform well in practice. Why?

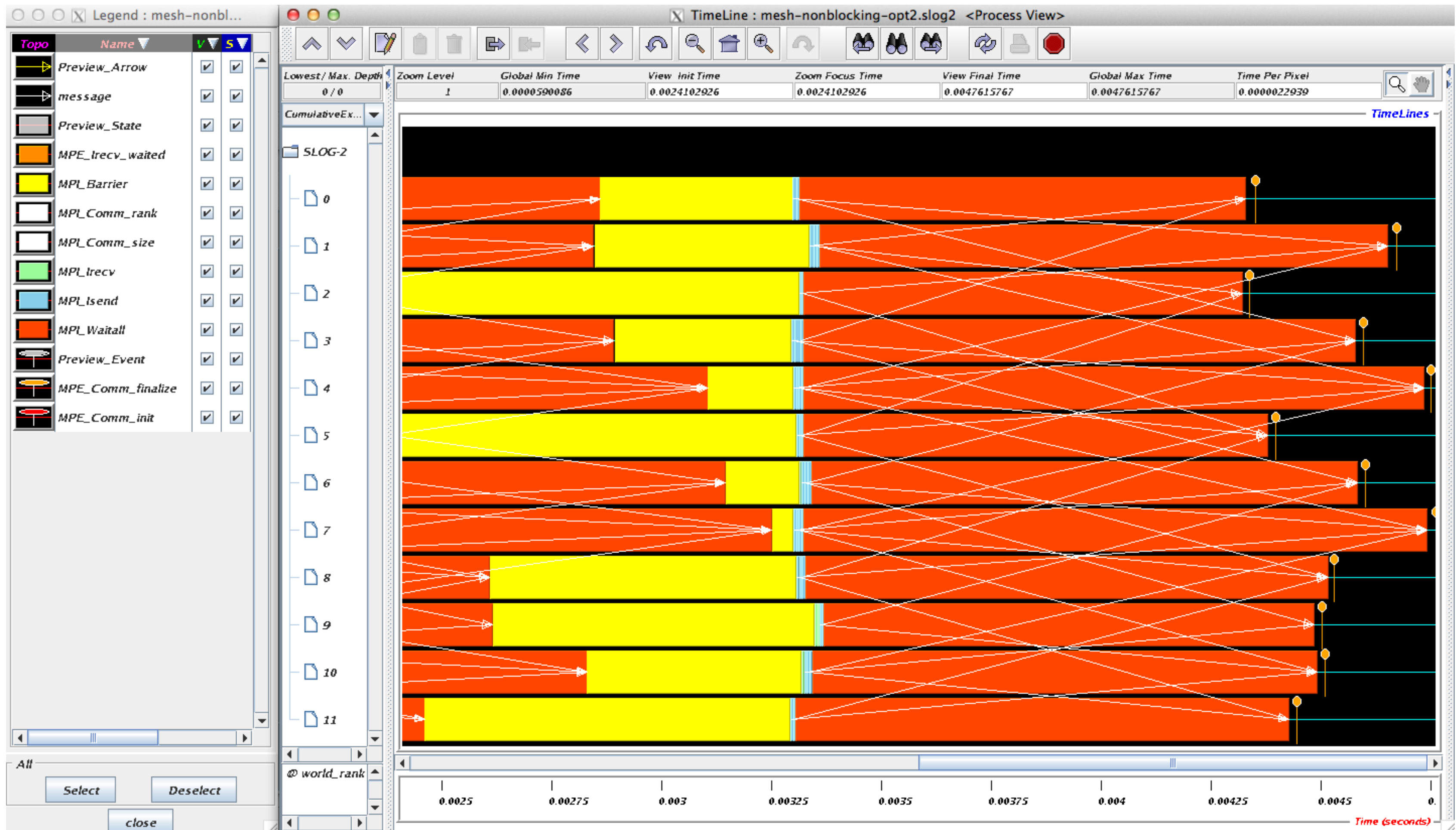
Timeline from IB Cluster



Fix 2: Use Isend and Irecv

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
              comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Isend(edge, len, MPI_DOUBLE, nbr[i], tag, comm,
              requests[n_neighbors + i]);
}
MPI_Waitall(2 * n_neighbors, requests, statuses);
```

Timeline from IB Cluster



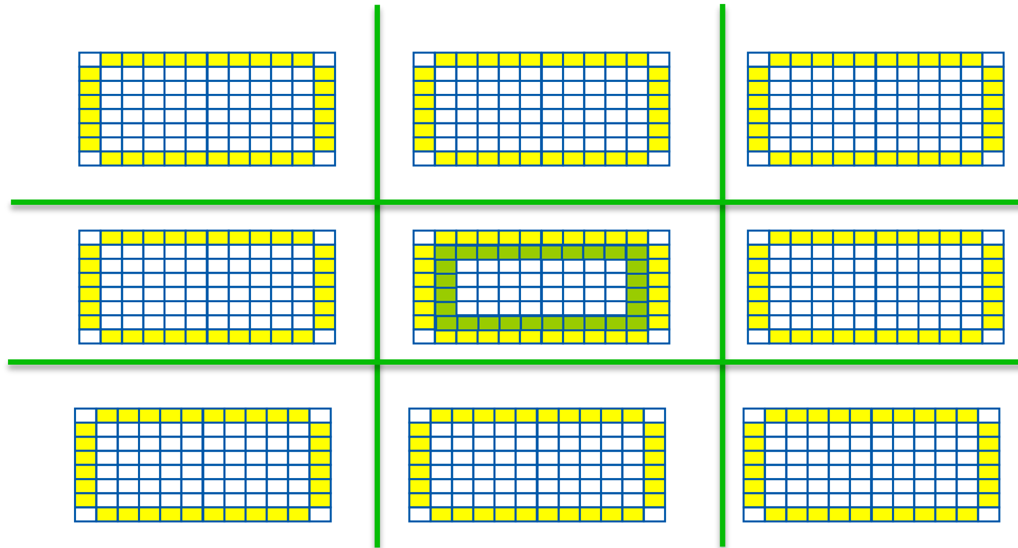
Note processes 4 and 7 are the only interior processors; these perform more communication than the other processors

Lesson: Defer Synchronization

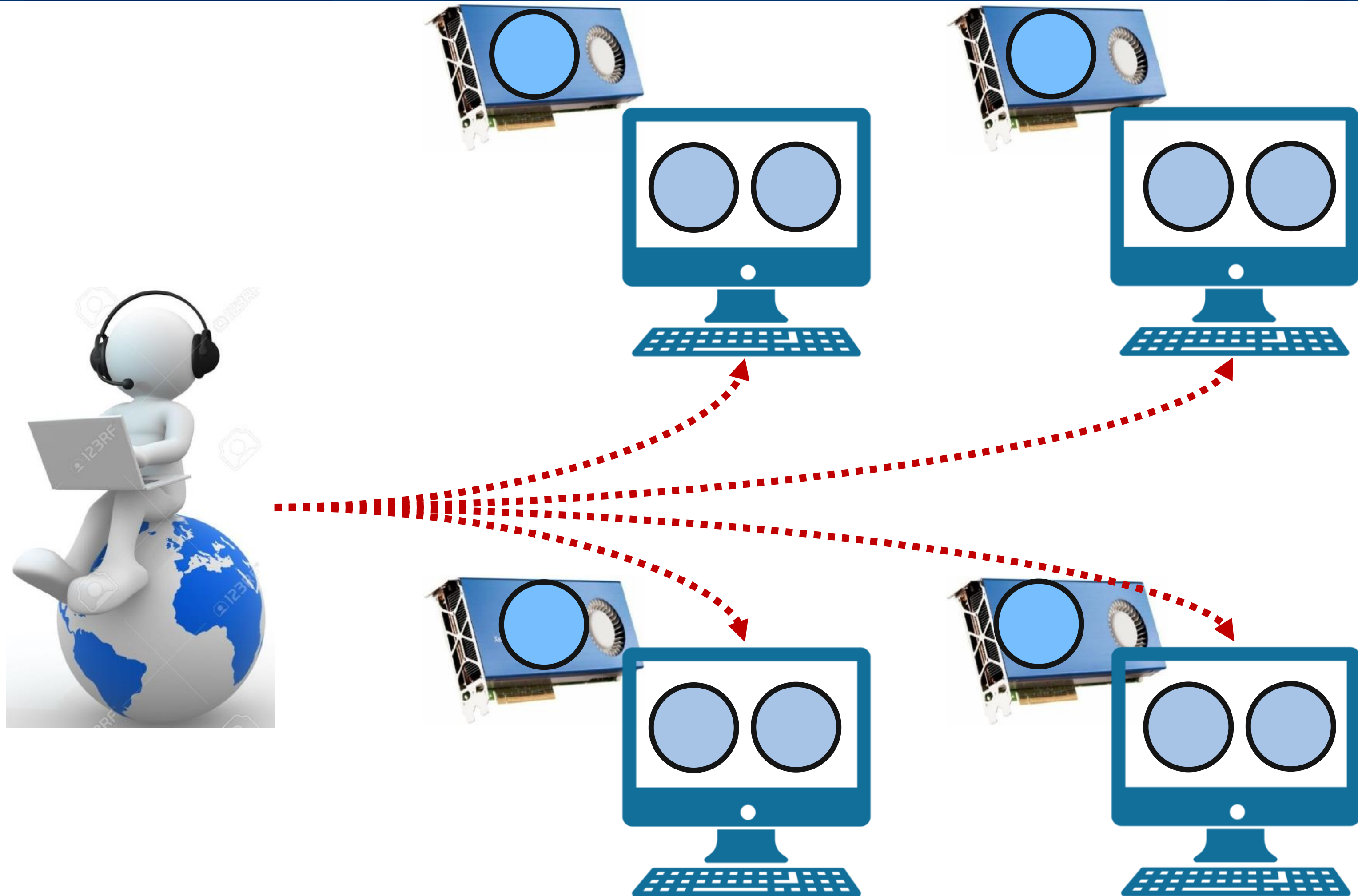
- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use non-blocking operations and `MPI_Waitall` to defer synch.
- Tools can help out with identifying performance issues
  - Tau, HPCToolkit, and Scalasca are popular profiling tools

Code Example

- *stencil_mpi_nonblocking.c*
- Non-blocking sends and receives
- Manually packing and unpacking the data
- Additional communication buffers are needed



Be careful of heterogeneity



MPI in Heterogeneous Environments

- MPI does not mandate that your program run in homogeneous environments
- But many common algorithms use a homogeneity assumption, primarily for simplicity
  - Assuming that all processors compute at the same speed will result in your algorithm running at the speed of the slowest processor

GPUs vs. Intel Xeon Phi

- GPU
  - **No MPI process on a GPU** (since there's no operating system)
  - GPU systems are typically homogeneous
    - Each MPI process has one or more CPU cores + one or more GPUs
    - All MPI processes are "identical"
- Intel Xeon Phi
  - GPU-like mode available ("offload mode")
  - Also provides a "native mode" where **you can have MPI processes running on the Intel Xeon Phi** (since it has an OS)
  - These systems can be heterogeneous
    - Some MPI processes run on the Xeon and some run on the Xeon Phi
    - Your algorithm might need to take such heterogeneity into account

What will be covered in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **More advanced topics:**
  - Non-blocking communication, **collective communication**, datatypes
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a comm.
- `MPI_BCAST` distributes data from one process (the root) to all others in a communicator
- `MPI_REDUCE` combines data from all processes in the communicator and returns it to one process
- In many numerical algorithms, `SEND/RECV` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency

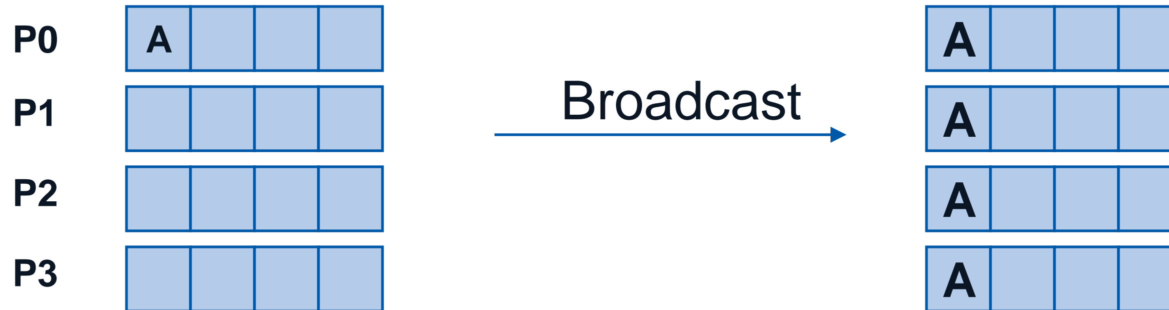
MPI Collective Communication

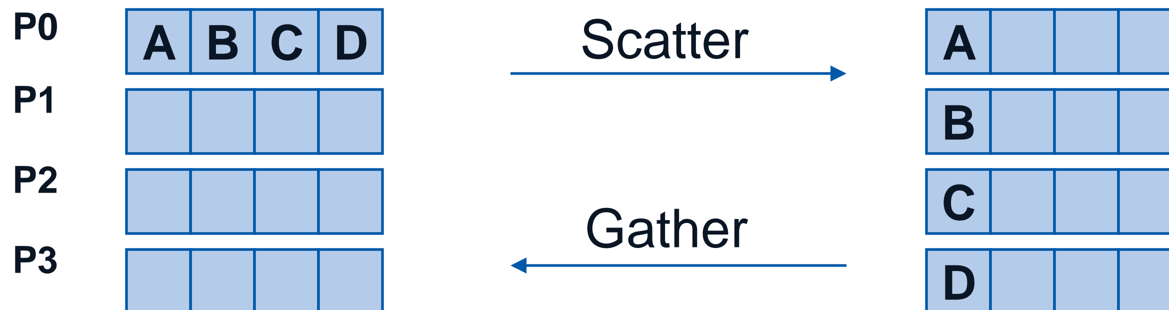
- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used
  - Different communicators deliver similar functionality
- Non-blocking collective operations in MPI-3
  - Covered later (but conceptually simple)
- Three classes of operations:
  - Synchronization
  - Data movement
  - Collective computation

Synchronization

- `MPI_Barrier(comm)`
  - Blocks until all processes in the group of the communicator `comm` call it
  - A process cannot get out of the barrier until all other processes have reached barrier

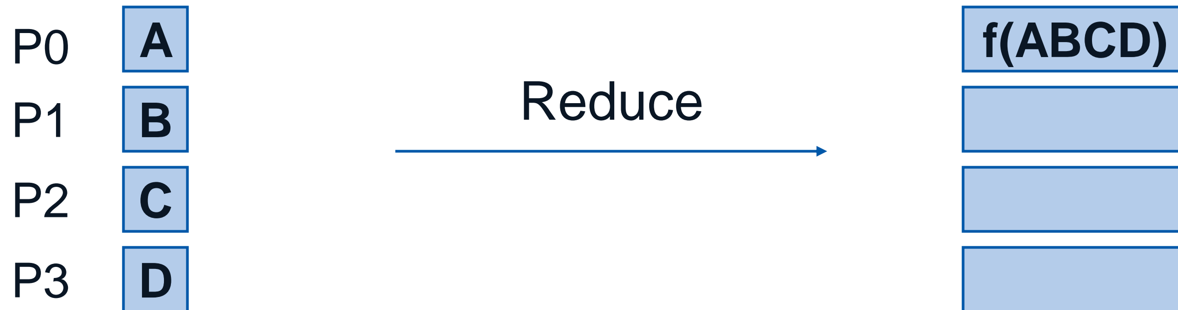
Collective Data Movement

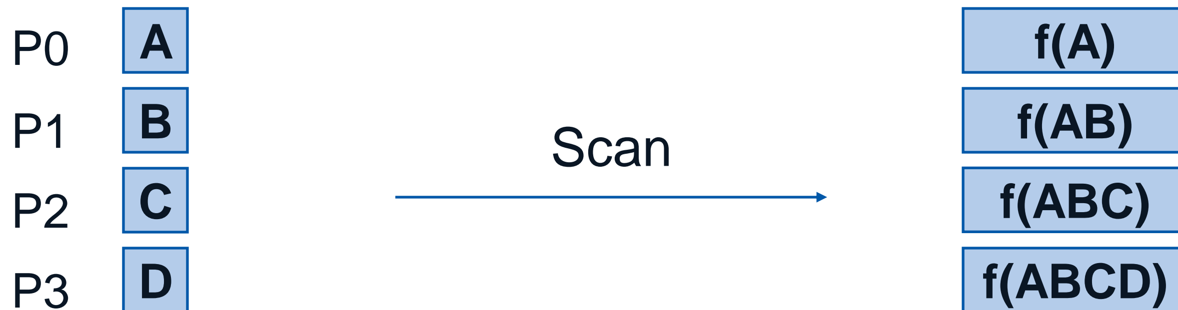




More Collective Data Movement

Collective Computation





MPI Collective Routines

- Many Routines: **`MPI_ALLGATHER`**, **`MPI_ALLGATHERV`**, **`MPI_ALLREDUCE`**, **`MPI_ALLTOALL`**, **`MPI_ALLTOALLV`**, **`MPI_BCAST`**, **`MPI_GATHER`**, **`MPI_GATHERV`**, **`MPI_REDUCE`**, **`MPI_REDUCESCATTER`**, **`MPI_SCAN`**, **`MPI_SCATTER`**, **`MPI_SCATTERV`**
- "**`All`**" versions deliver results to all participating processes
- "**`V`**" versions (i.e.: vector) allow the chunks to have <u>different size for each rank</u>
- **`MPI_ALLREDUCE`**, **`MPI_REDUCE`**, **`MPI_REDUCESCATTER`**, and **`MPI_SCAN`** take both <u>built-in</u> and <u>user-defined</u> functions

MPI Built-in Collective Computation Operations

`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise exclusive or
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location

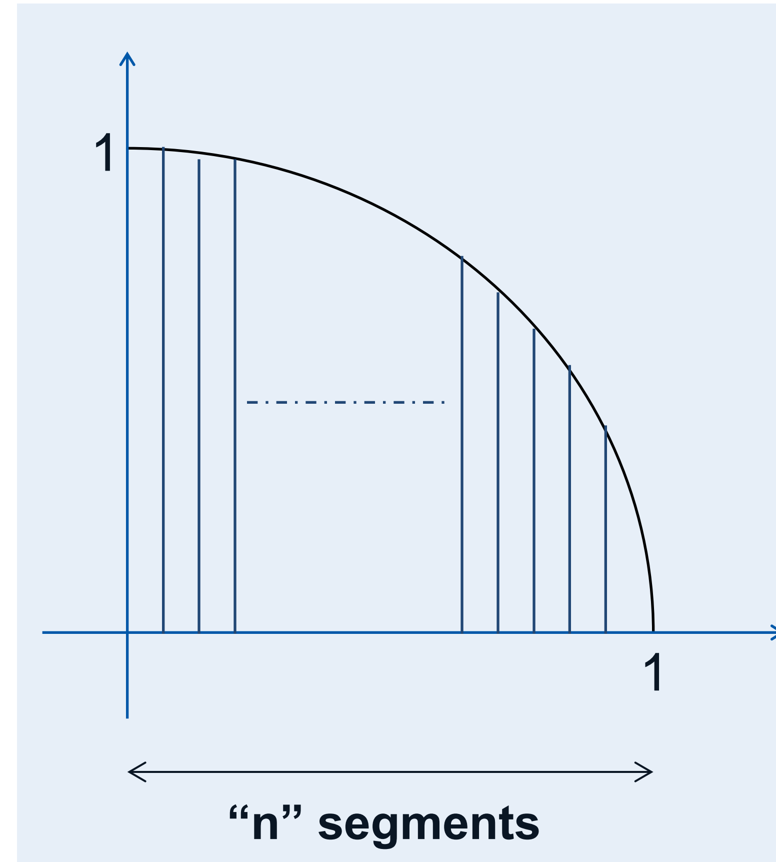
Defining your own Collective Operations

- Create your own collective computations with:
```
MPI_Op_create(user_fn, commutes, &op);
MPI_Op_free(&op);

user_fn(invec, inoutvec, len, datatype);
```
- The user function should perform:
```
inoutvec[i] = invec[i] op inoutvec[i];
```
for i from 0 to len-1
- The user function can be non-commutative, but must be associative

Example: Calculating Pi

- Calculating pi via numerical integration
  - Divide interval up into subintervals
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get pi



1. Width of each segment (w) will be 1/n
2. Distance (d(i)) of segment "i" from the origin will be "i * w"
3. Height of segment "i" will be sqrt(1 – [d(i)]^2)

Example: PI in C

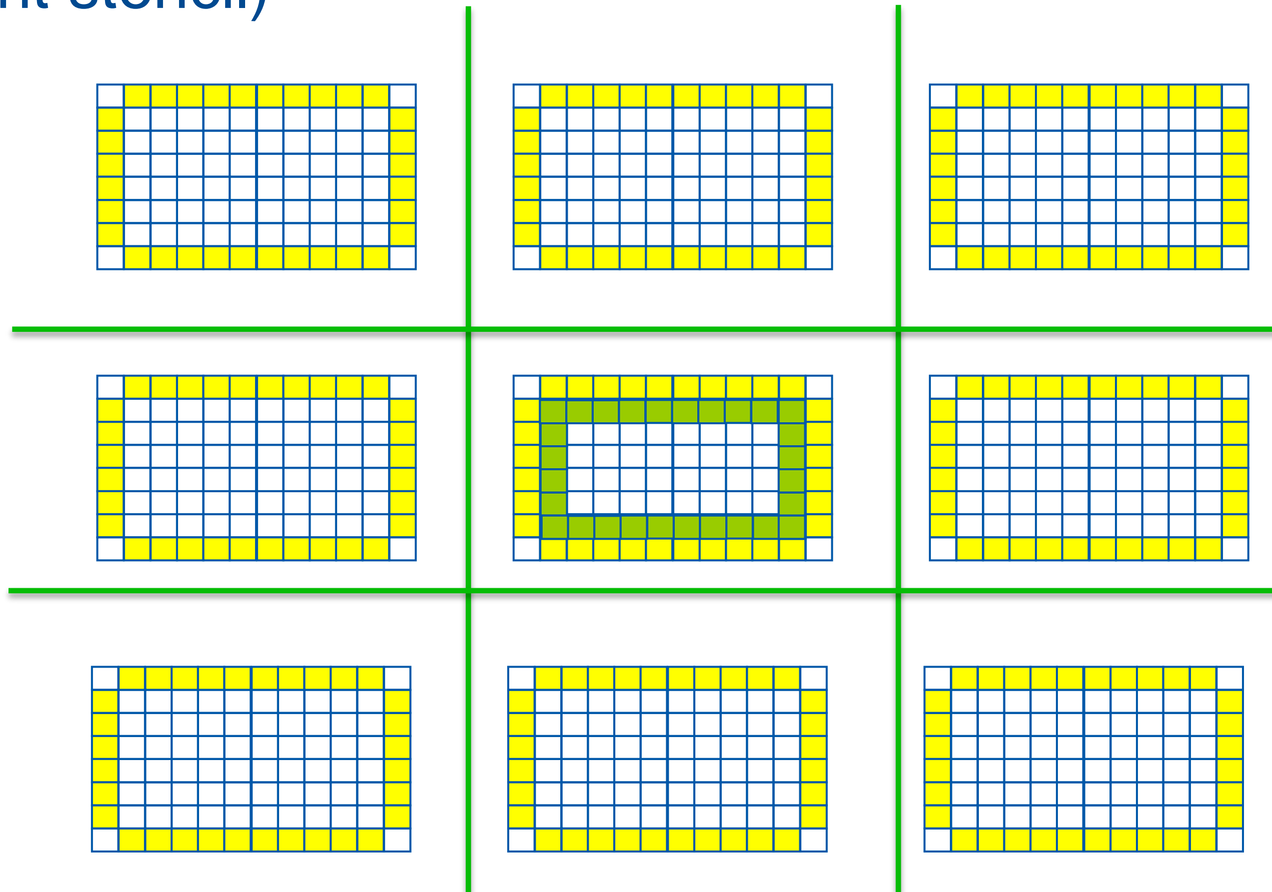
```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 - (((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi,
                fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

What will be covered in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **More advanced topics:**
  - Non-blocking communication, collective communication, **datatypes**
  - One-sided communication
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

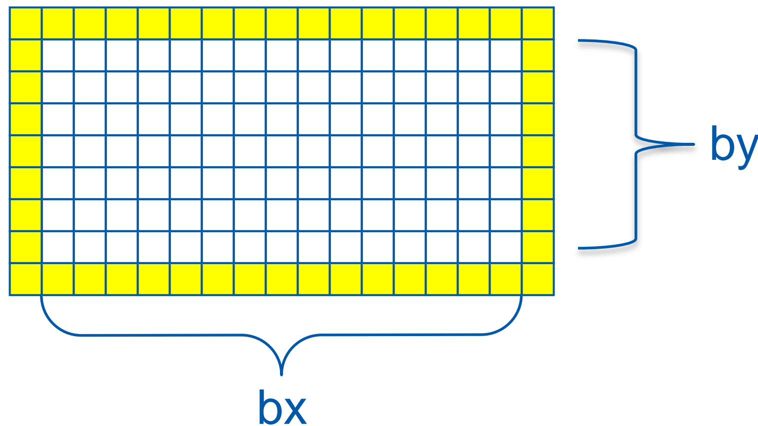
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)



The Local Data Structure

- Each process has its local "patch" of the global array
  - "bx" and "by" are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)



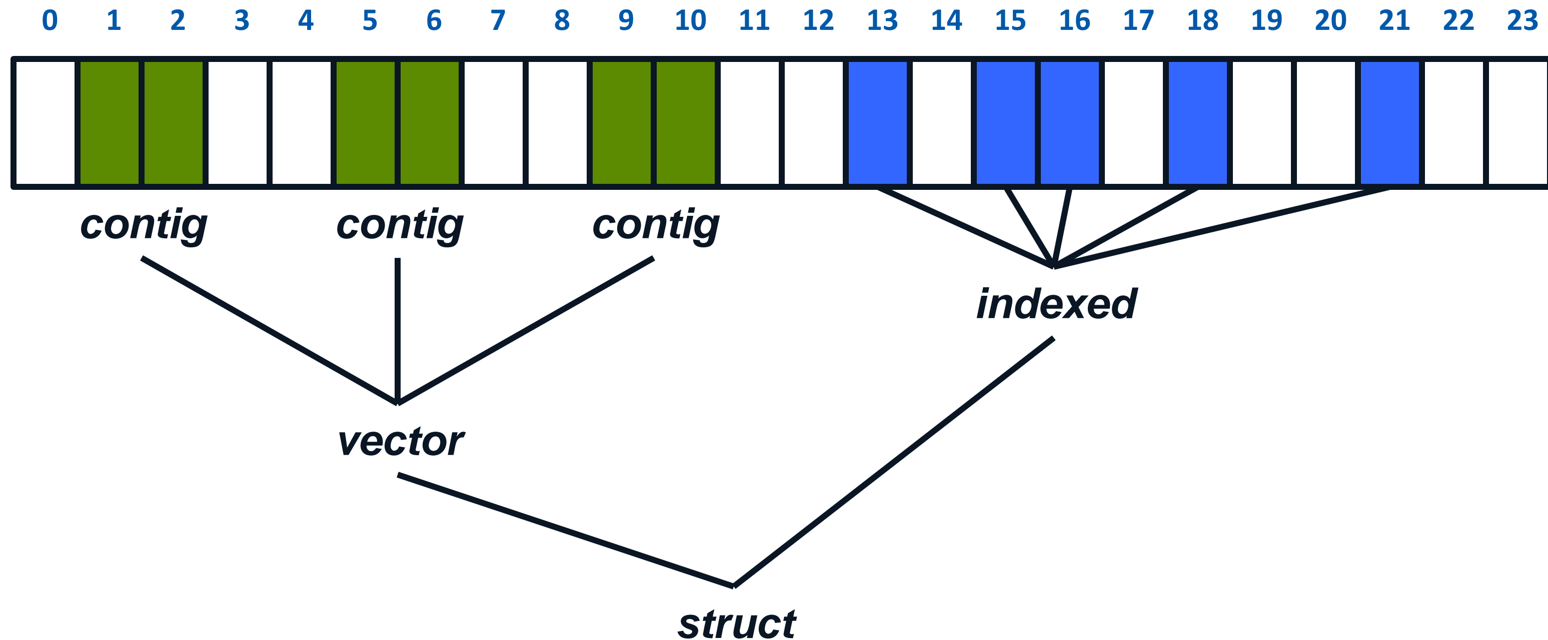
Introduction to Datatypes in MPI

- Datatypes allow to (de)serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O
- Several constructors allow arbitrary layouts
  - Recursive specification possible
  - *Declarative* specification of data-layout
    - “what” and not “how”, leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple

Simple/Predefined Datatypes

- Equivalents exist for all C, C++ and Fortran native datatypes
  - C int → MPI_INT
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T
  - Fortran integer → MPI_INTEGER
- MPI provides routines to represent more complex user-defined datatypes
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)

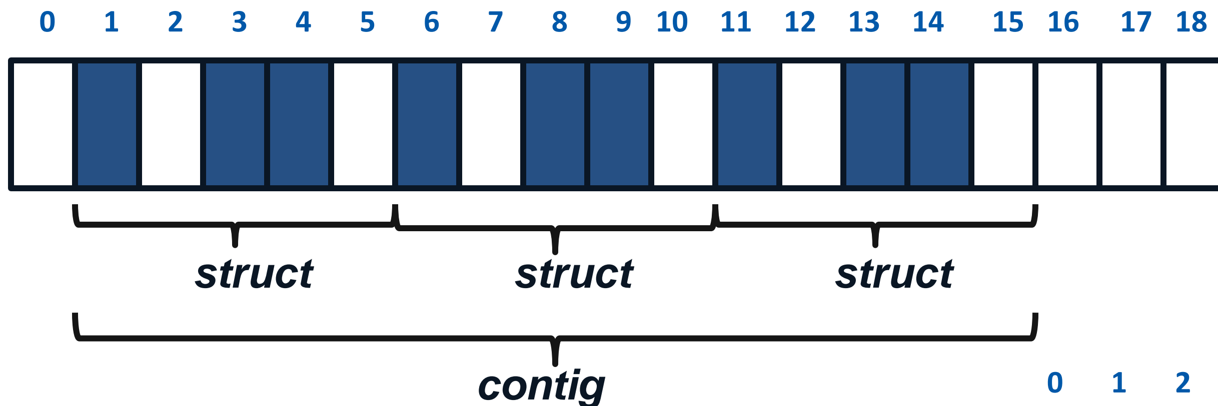
Derived Datatype Example

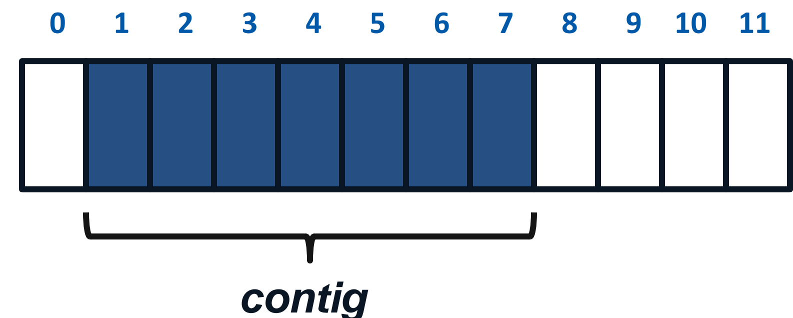


MPI_Type_contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



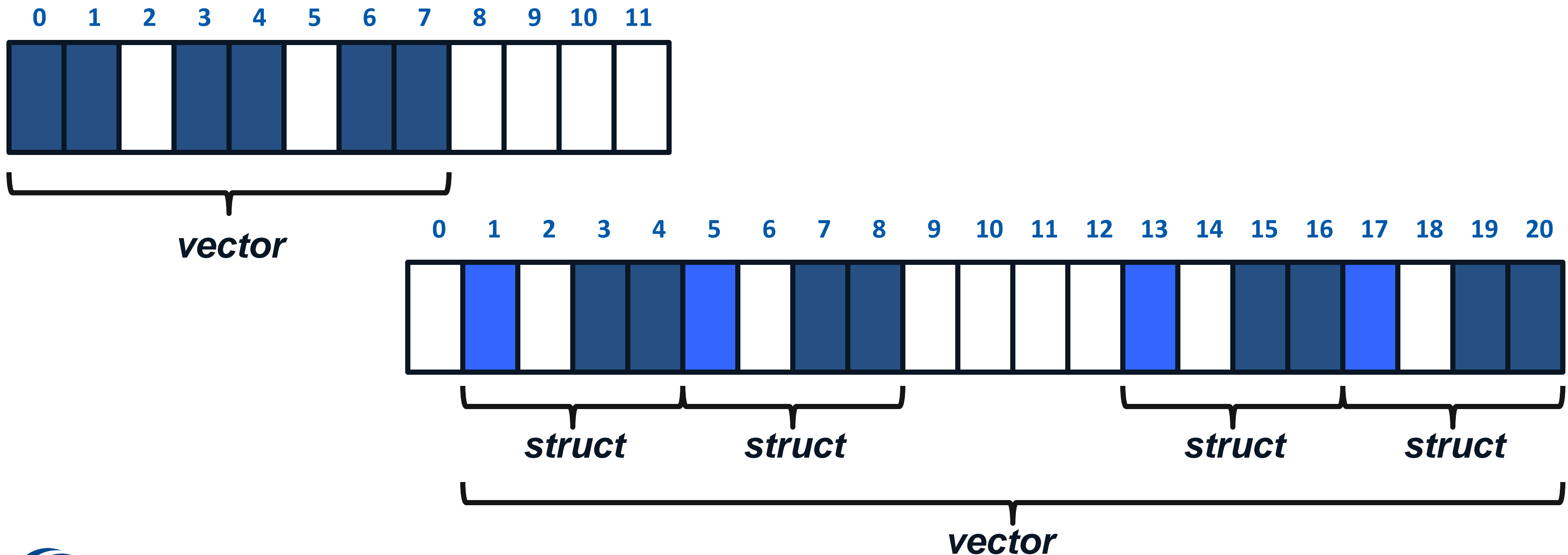


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI_Type_vector

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

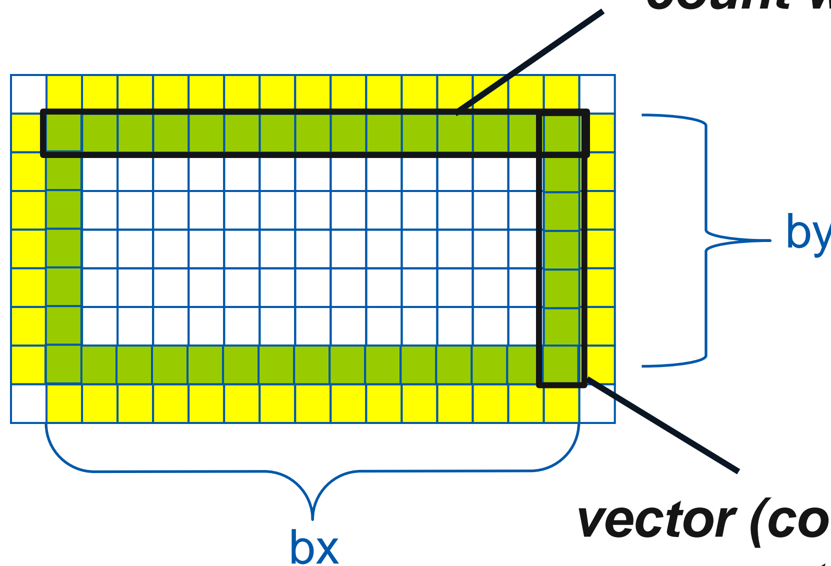
- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Use Datatype in Halo Exchange

contig (count=bx, MPI_DOUBLE, …) or
count with MPI_DOUBLE

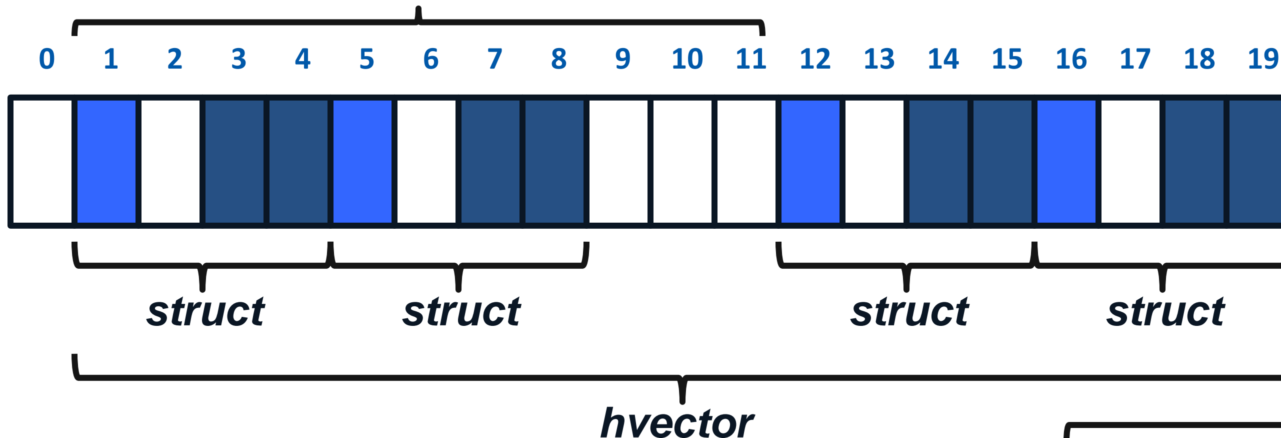


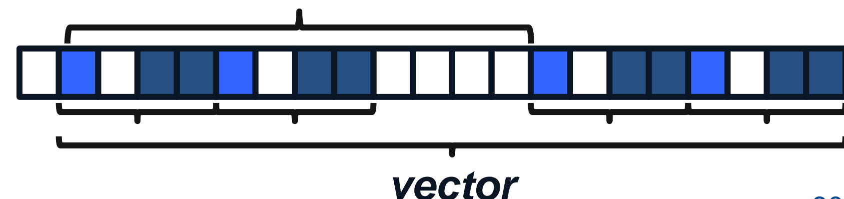
vector (count=by, blocklen=1, stride=bx+2, MPI_DOUBLE, …)

MPI_Type_create_hvector

```
MPI_Type_create_hvector(int count, int blocklen, MPI_Aint stride,
      MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Create non-unit strided vectors, stride is in bytes
- Useful for composition, e.g., vector of structs

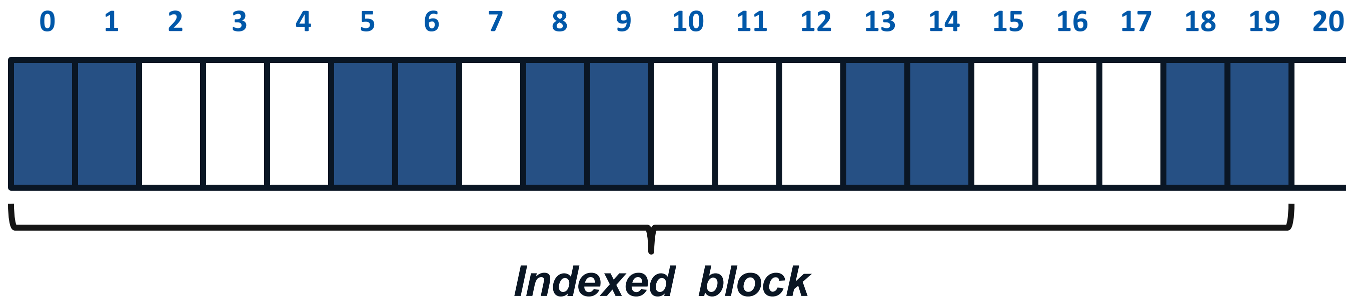




MPI_Type_create_indexed_block

```
MPI_Type_create_indexed_block(int count, int blocklen,
       int *array_of_displacements,
       MPI_Datatype oldtype, MPI_Datatype *newtype)
```

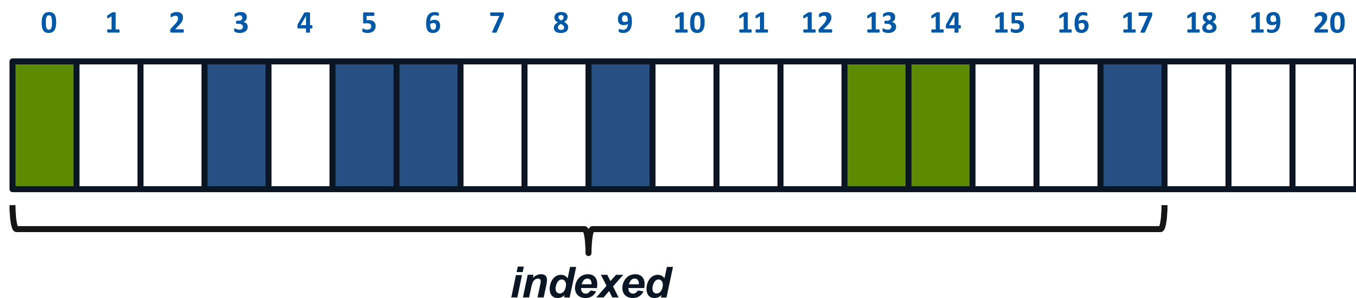
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

```
MPI_Type_indexed(int count, int* array_of_blocklens,
        int *array_of_displacements,
        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}

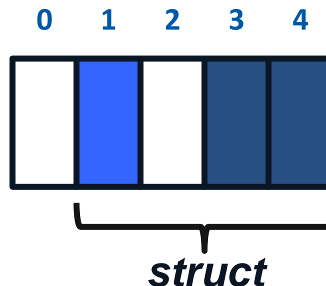


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI_Type_create_struct

```
MPI_Type_create_struct(int count,
        int *array_of_blocklens,
        int *array_of_displacements,
        MPI_Datatype *array_of_types,
        MPI_Datatype *newtype)
```

- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

```
MPI_Type_create_subarray(int ndims, int* array_of_sizes,
        int *array_of_subsizes, int *array_of_starts,
        int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Convenience function for creating datatypes for array segments
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

MPI_BOTTOM and MPI_Get_address

- Specify absolute addresses instead of offsets from buf ptr
- MPI_BOTTOM is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- MPI_Get_address
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use "&" operator in C!)
- Very important to
  - build struct datatypes
  - If data spans multiple arrays

```
int a = 4;
float b = 9.6;
MPI_Datatype struct;

MPI_Get_address(&a, &disps[0]);
MPI_Get_address(&b, &disps[1]);
...
MPI_Type_create_struct(count,
                       blocklens, disps,
                       oldtypes, &struct);
...
MPI_Recv(MPI_BOTTOM, 1, struct,…);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Commit, Free, and Dup

- Types must be committed before use
  - Only the ones that are used!
  - *MPI_Type_commit* may perform heavy optimizations (and will hopefully)

- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it

- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)

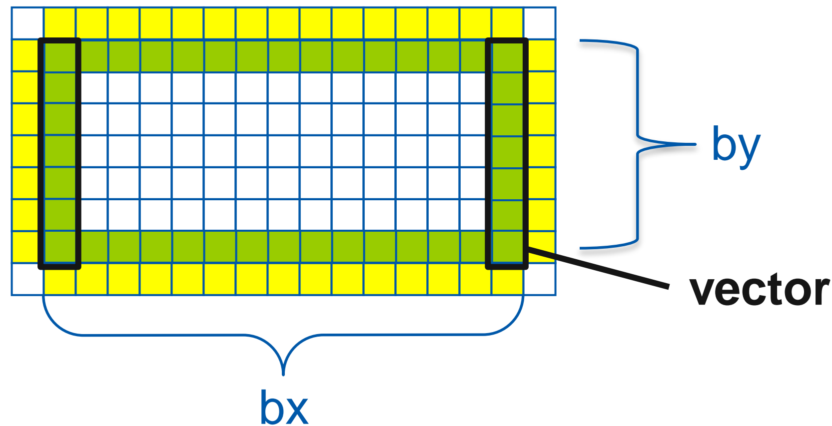
Datatype Selection Order

- Simple and effective performance model:
  - More parameters == slower
- **predefined < contig < vector < index_block < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Try datatype “compression” bottom-up

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes

Code Example

- *stencil-mpi-ddt.c*
- Non-blocking sends and receives
- Data location specified by MPI datatypes
- Manual packing of data no longer required



www.bsc.es

Barcelona Supercomputing Center

Centro Nacional de Supercomputación

Thank you!

For further information please contact

marc.jorda@bsc.es, antonio.pena@bsc.es