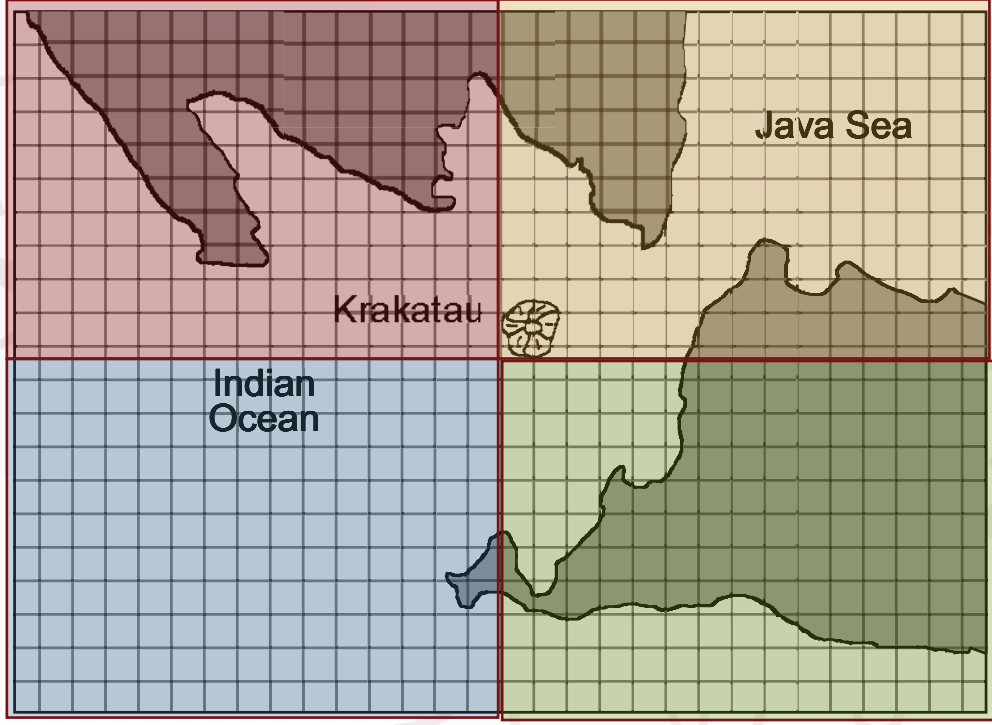


Computing on distributed data

Troels Haugbølle
haugboel@nbi.ku.dk

March 14, 2022

UNIVERSITY OF COPENHAGEN

Overview

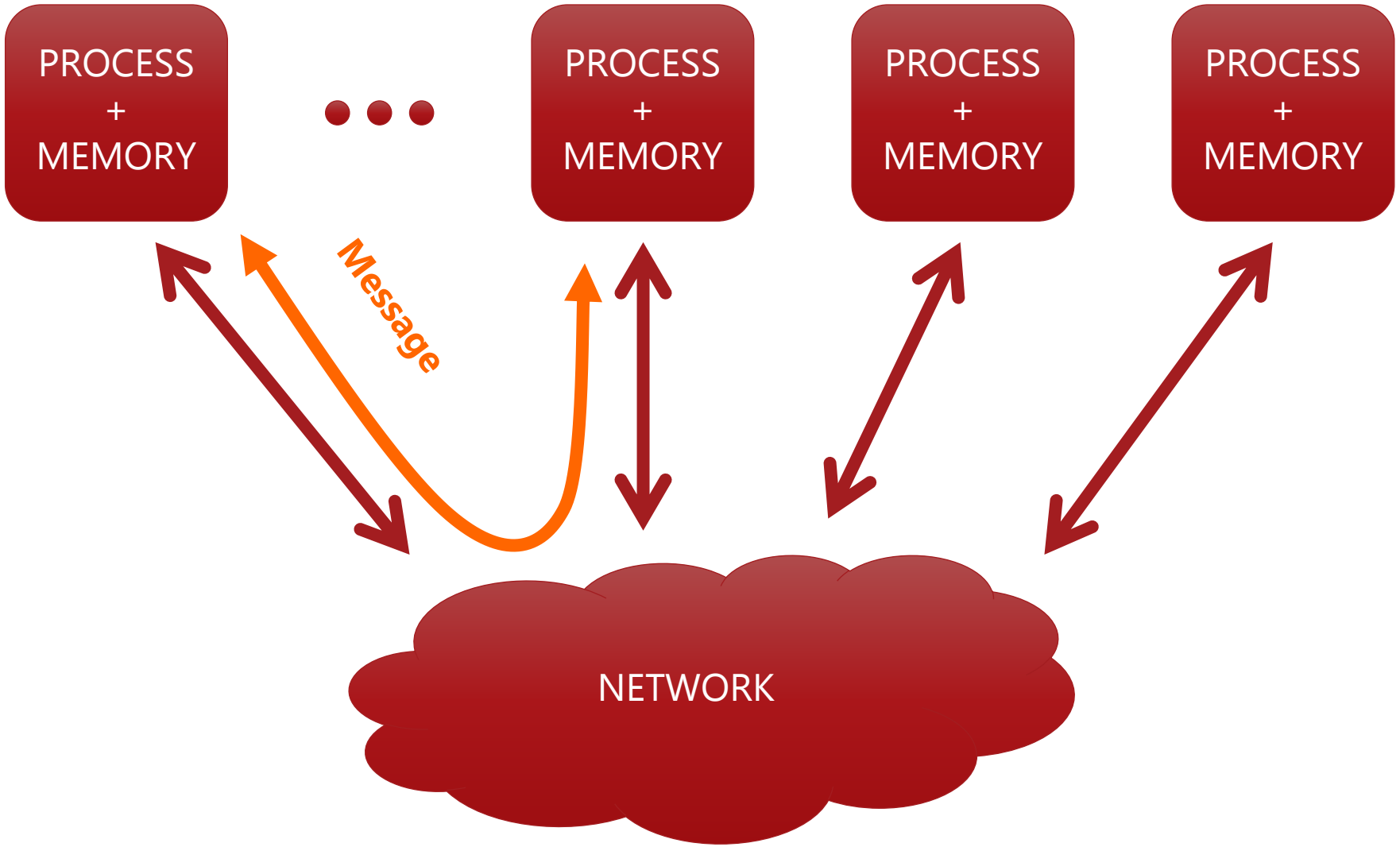
Today:

- Message parsing model: recap
- Collective communication
- Domain decomposition and guard zones
- MPI Groups and Communicators

Learning Objectives for this module:

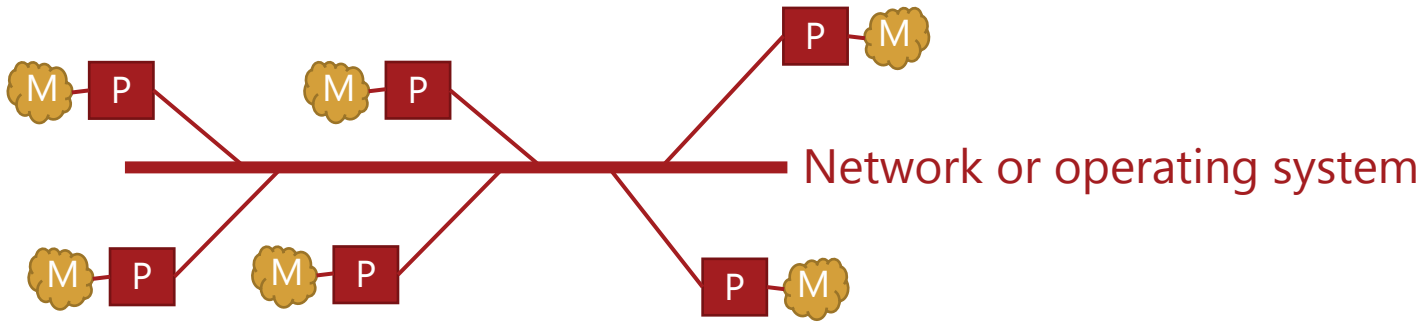
- Cluster and Network topology
- Message Passing Interface (MPI) library for distributed computing
- Hybrid Computing

Recap: Message Passing Model

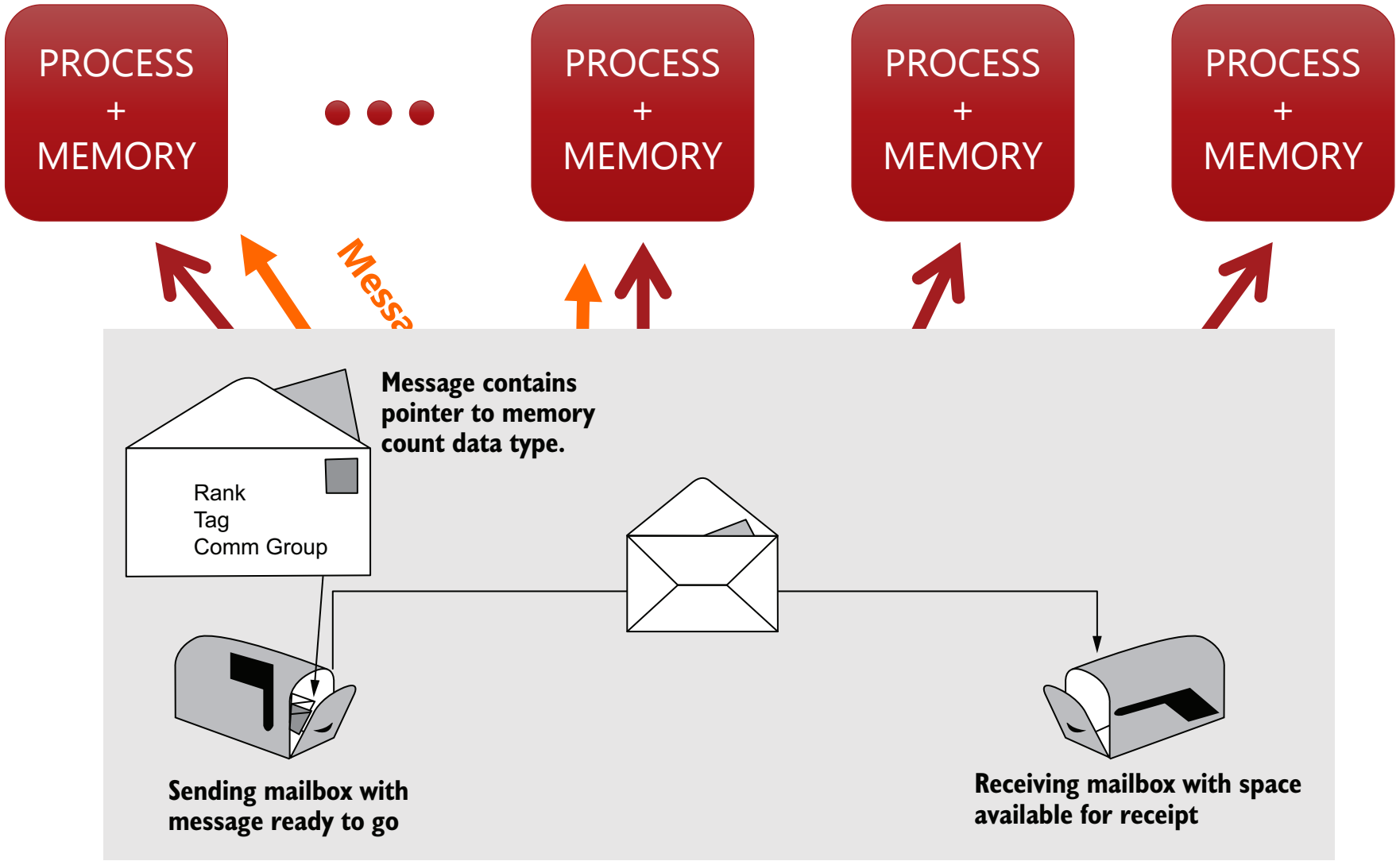


Message Passing Programming Model

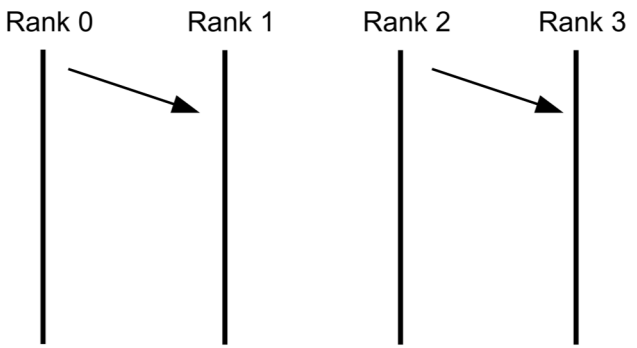
- No shared memory space
- Data transfer is explicit through matching operations by different processes. Point-to-point: a send is matched by a receive
- Each process is given a *rank* (different than *threads* inside a process!)
- Program uses the rank to perform (different) calculations on different data
- Industry standard: MPI, Main implementations: OpenMPI, MPICH
- MPI is based on processes. Each process may be running on its own core, but does not have to. You can develop on your laptop, run on Fugaku!



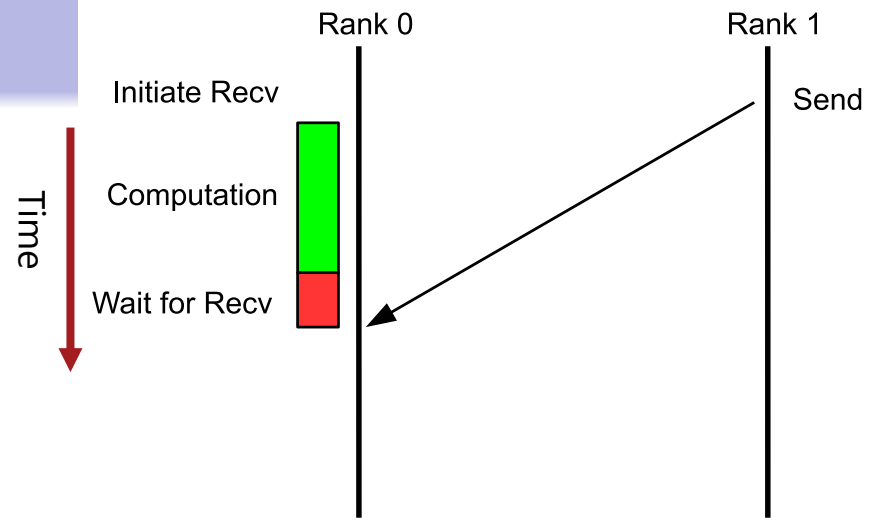
Recap: Message Passing Model



```
if RANK is even:
  Send message to RANK+1
else:
  Receive message from RANK-1
```




```cpp
#include <iostream>
#include <vector>
#include <mpi.h>

int main( int argc, char *argv[] ) {
  int rank, size;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );

  std::vector<double> data(100, rank); // Vector of my rank
  if (rank % 2 == 0 && rank < size-1) { // Even rank
    MPI_Send(&data[0], 100, MPI_DOUBLE, rank+1, 0,
             MPI_COMM_WORLD);
  } else if (rank % 2 == 1 && rank > 0) { // Odd rank
    MPI_Recv(&data[0], 100, MPI_DOUBLE, rank-1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  MPI_Finalize();
  return 0;
}
```




```
int rank, size;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

int send_rank = (rank+1) % size;
int recv_rank = (rank-1) % size;
std::vector<double> data_send(100, rank);
std::vector<double> data_recv(100);

MPI_Request send_req;
MPI_Isend(&data_send[0], 100, MPI_DOUBLE, send_rank, 0,
          MPI_COMM_WORLD, &send_req);

MPI_Request recv_req;
MPI_Irecv(&data_recv[0], 100, MPI_DOUBLE, recv_rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE, &recv_req);
```

Compute stuff while waiting →

```
MPI_Wait(&send_req, MPI_STATUS_IGNORE);
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
```

Collective Operations

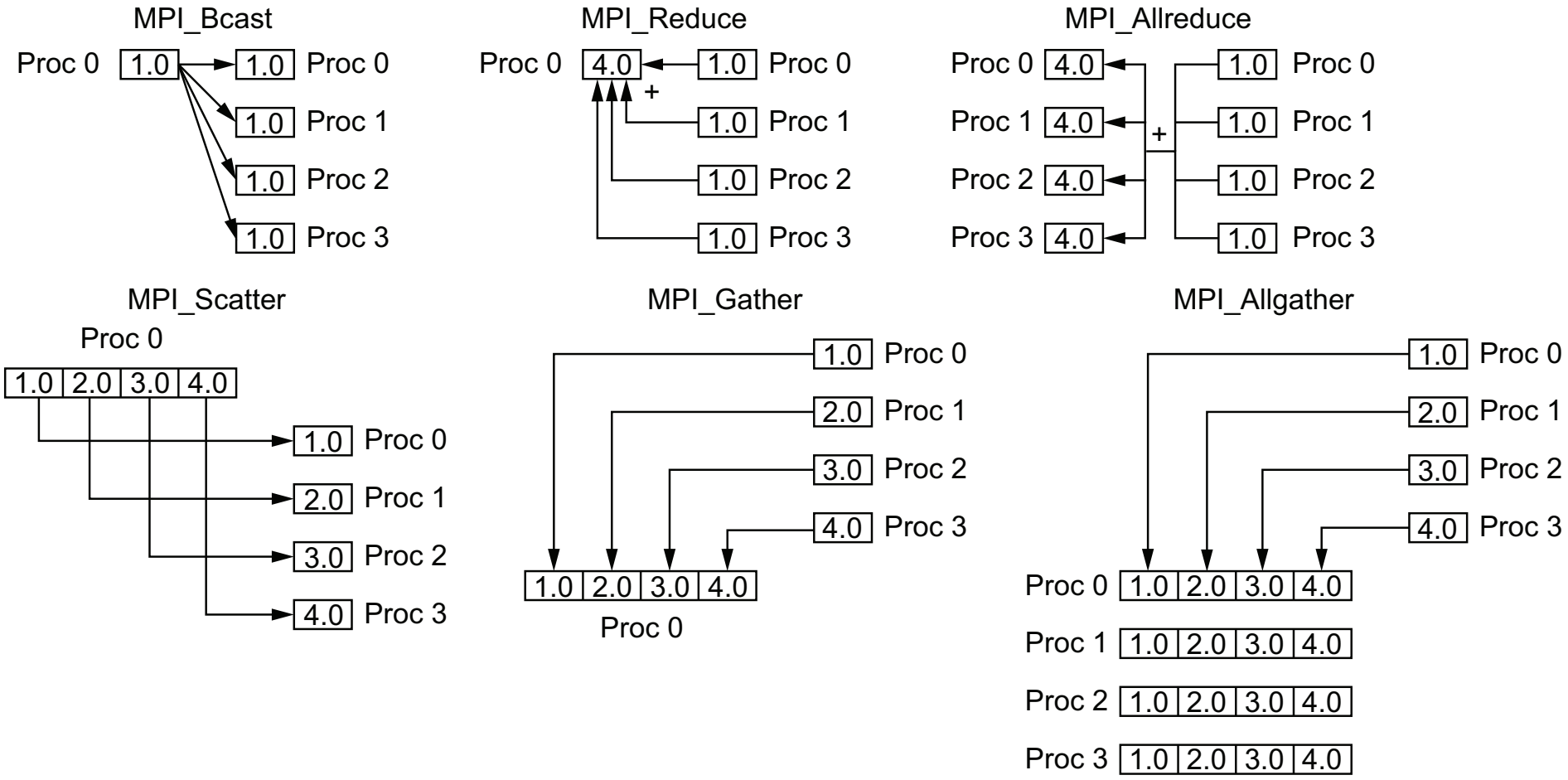
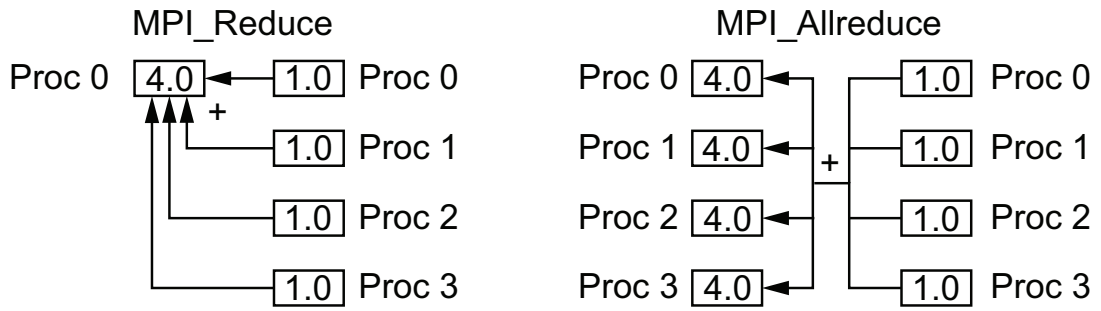


Figure 8.4 The data movement of the most common MPI collective routines provide important functions for parallel programs. Additional variants `MPI_Scatterv`, `MPI_Gatherv`, and `MPI_Allgatherv` allow a variable amount of data to be sent or received from the processes. Not shown are some additional routines such as the `MPI_Alltoall` and similar functions.

Sum of all process ranks

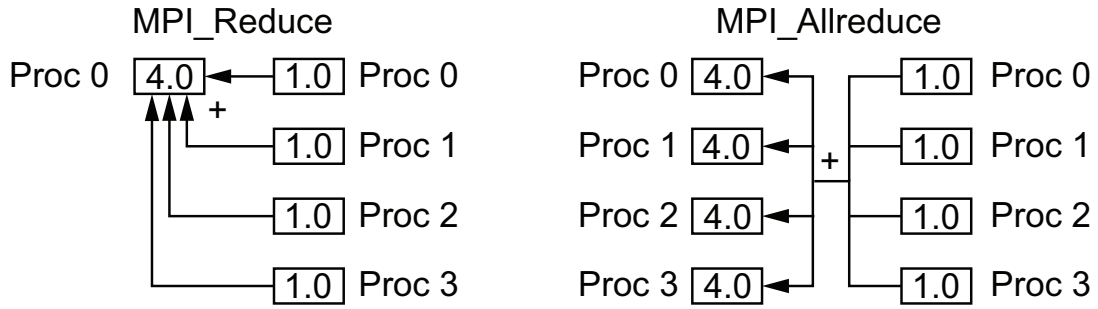
```
int sum = 0;
if (rank == 0) { // The root process
  for (int i=1; i < size; ++i) {
    int recv;
    MPI_Recv(&recv, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    sum += recv;
  }
} else {
  MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



Collective Operation

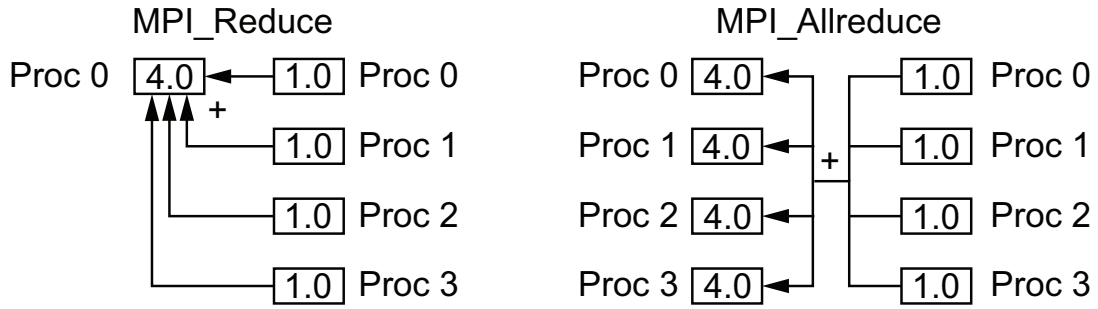
```
int MPI_Reduce(const void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Comm comm)
```

```
MPI_MAX - Maximum
MPI_MIN - Minimum
MPI_SUM - Sum
MPI_PROD - Product
MPI_LAND - Logical AND
MPI_LOR  - Logical OR
MPI_BAND - Bitwise AND
MPI_BOR -  Bitwise OR
User-defined Operations
```



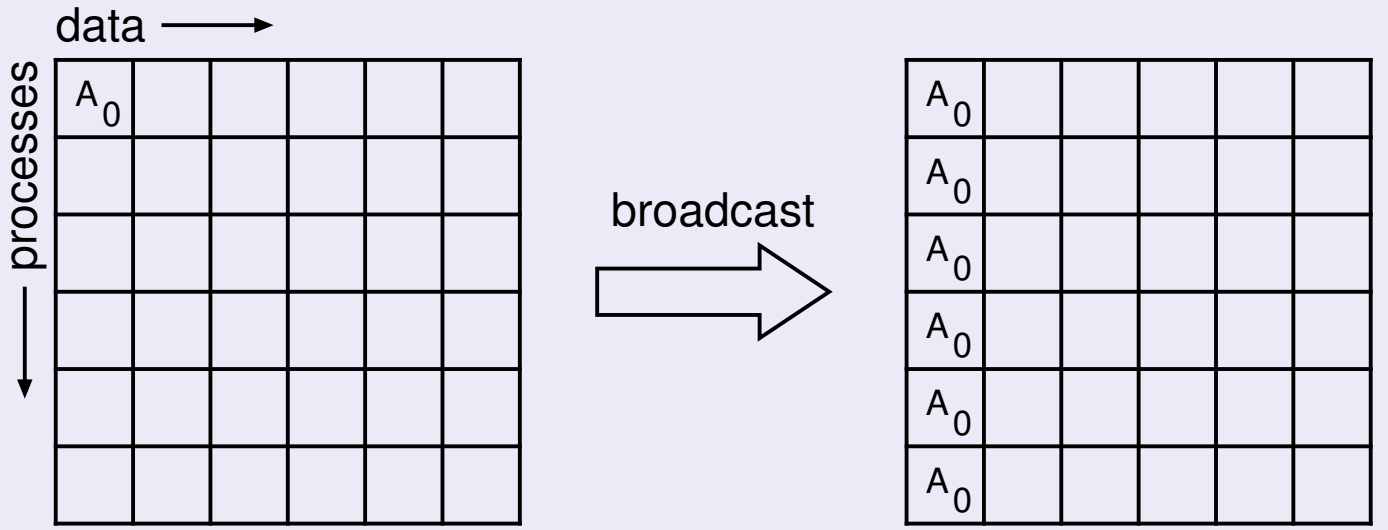
Sum of all process ranks

```
int sum;
MPI_Reduce(&rank, &sum, MPI_INT, MPI_SUM, 0,
           MPI_COMM_WORLD);

if (rank == 0) {
  std::cout << "The total sum: " << sum << std::endl;
} else {
  sum is undefined here!
}
```



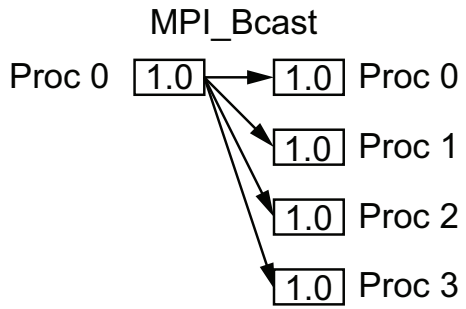
Collective Operations

Broadcast



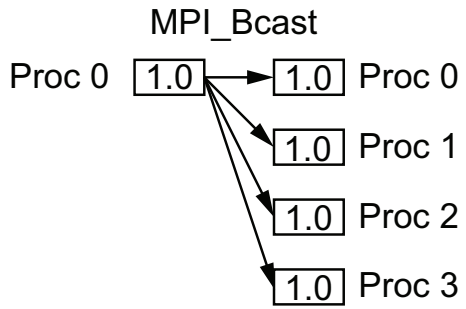
Broadcast 42 to the other processes

```
if (rank == 0) {
  int data = 42;
  for (int i = 1; i < size; ++i) {
    MPI_Send(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
} else {
  int data;
  MPI_Recv(data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}
```



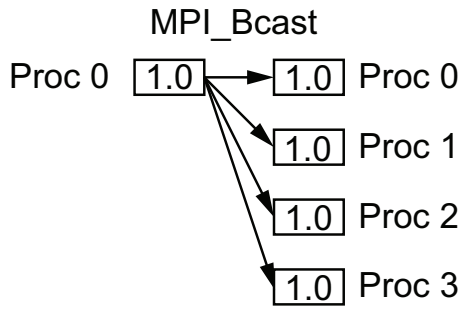
Collective Operation

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```



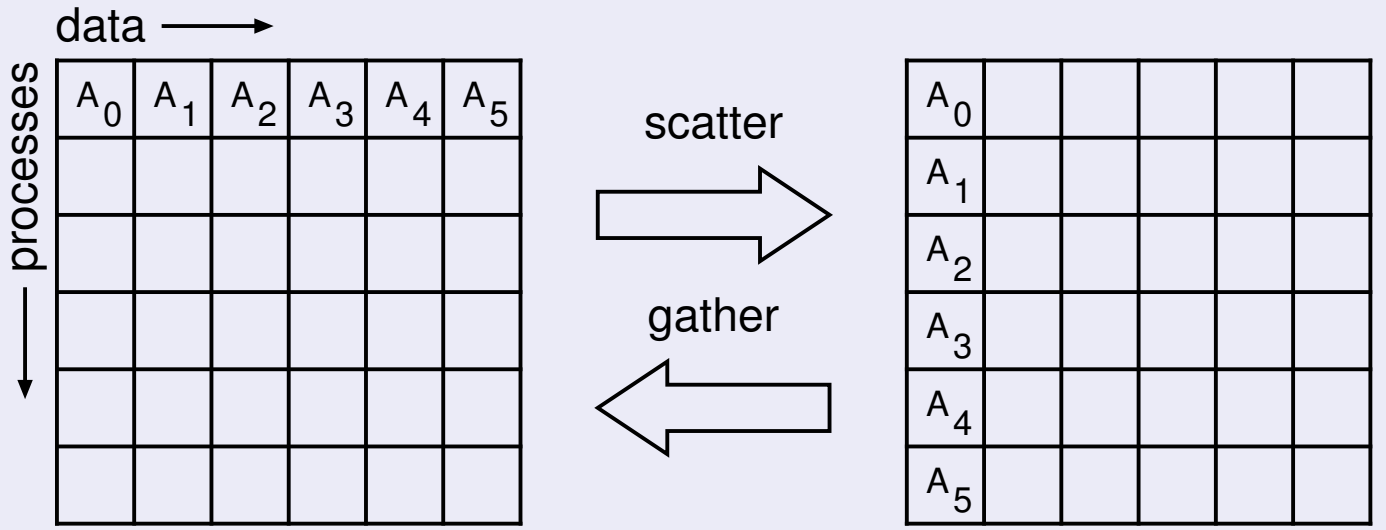
Broadcast 42 to the other processes

```
int data;
if (rank == 0) {
  data = 42;
}

MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

std::cout << "Data broadcasted: " << data << std::endl;
```



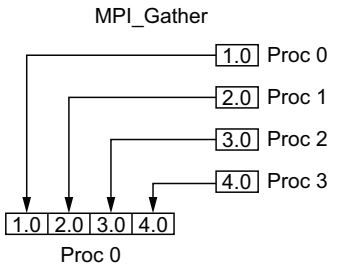
Collective Operations

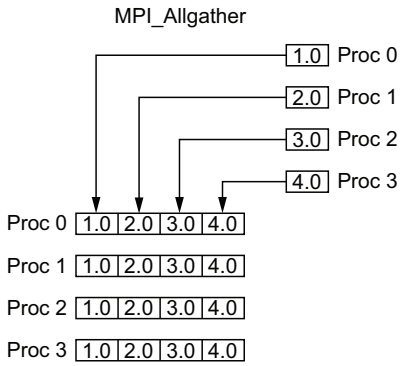
Gather / Scatter



Collective Operation

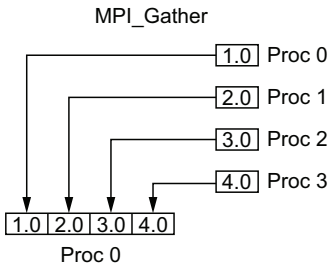
```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

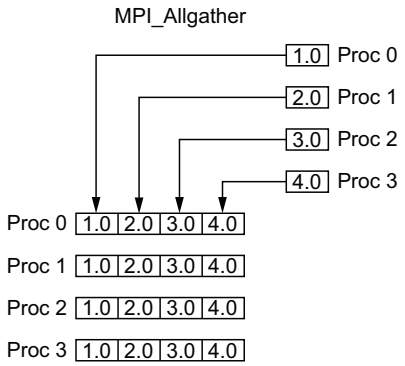




Gather 42's to the root process

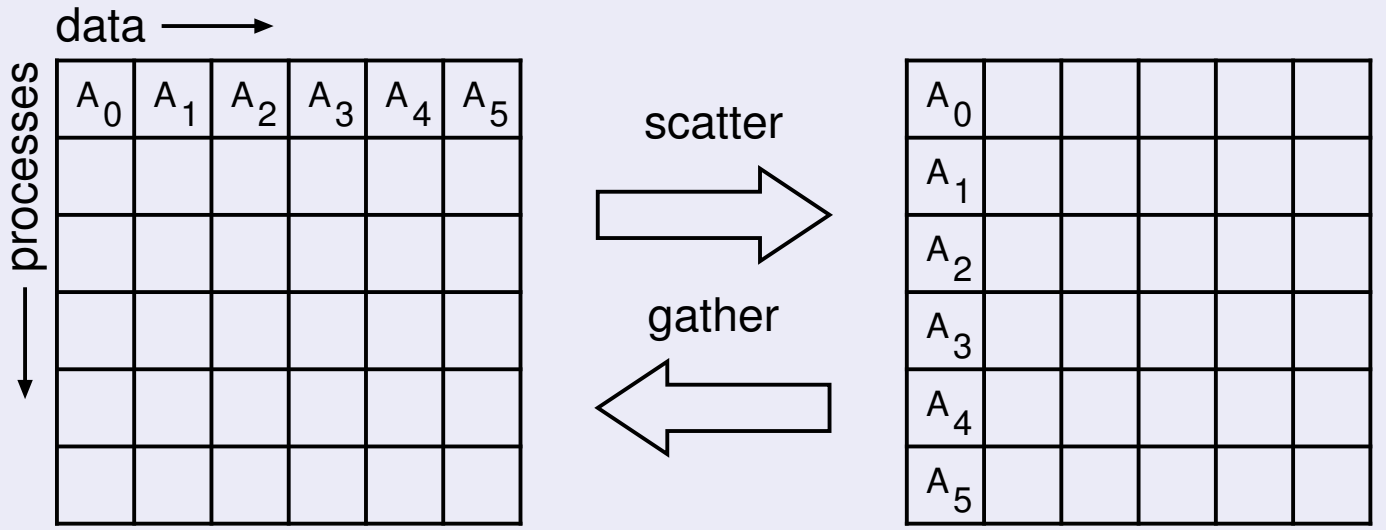
```
// An element per MPI process
std::vector<int> data_recv(size);
int data_send = 42;

if (rank == 0) {
  MPI_Gather(&data_send, 1, MPI_INT,
             &data_recv, 1, MPI_INT, 0,
             MPI_COMM_WORLD);
} else {
  MPI_Gather(&data_send, 1, MPI_INT, NULL,
             1, MPI_INT, 0,
             MPI_COMM_WORLD);
}
if (rank == 0) {
    std::cout << data_recv[0] << std::endl;
} else {
  data_recv is undefined here!
}
```





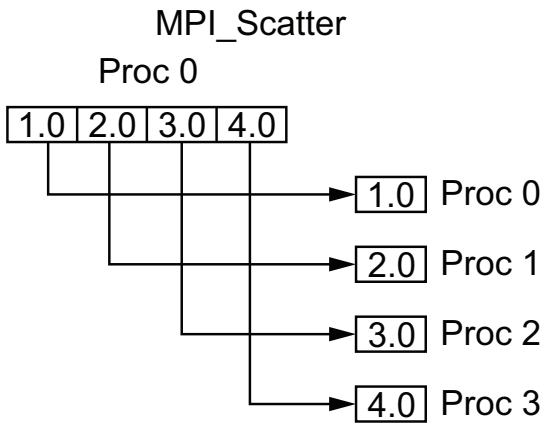
Collective Operations

Gather / Scatter



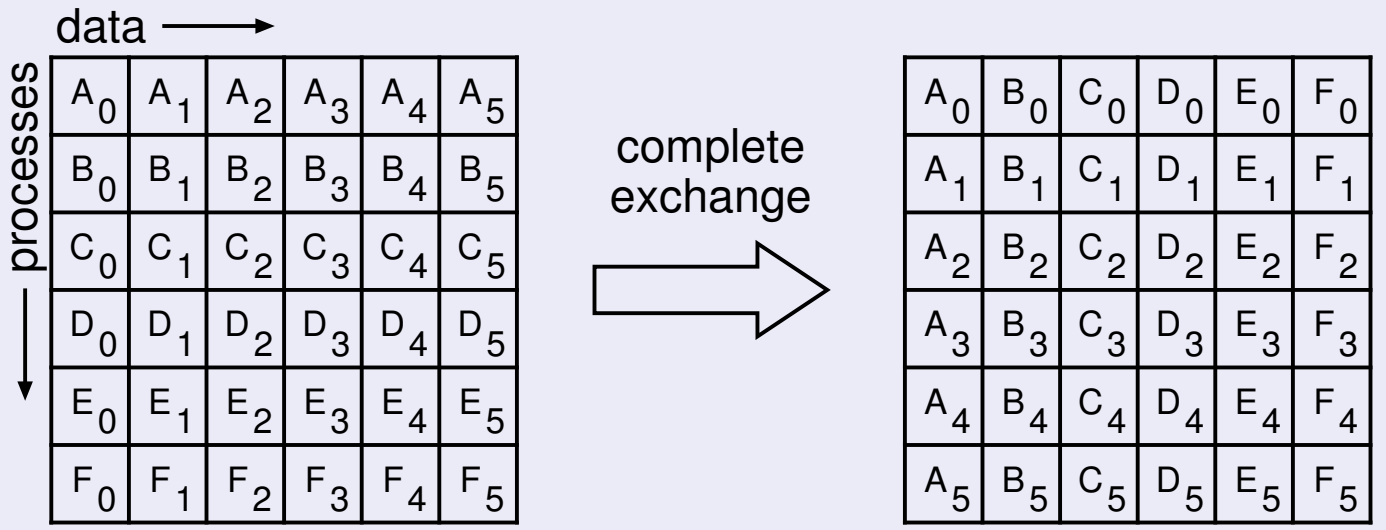
Collective Operation

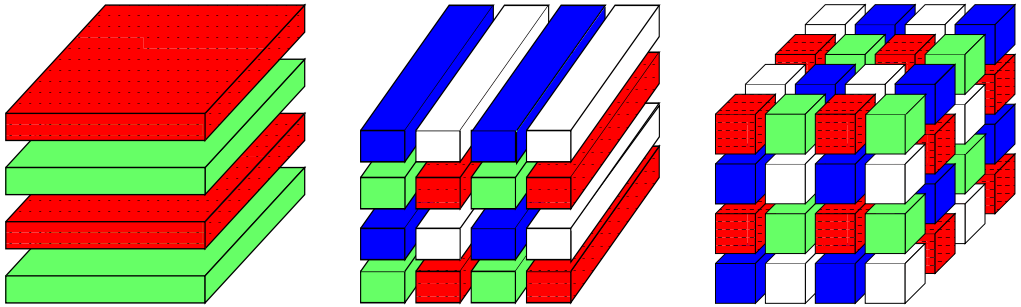
```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

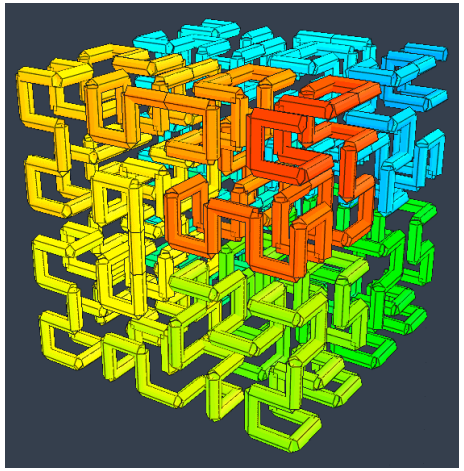


Collective Operations

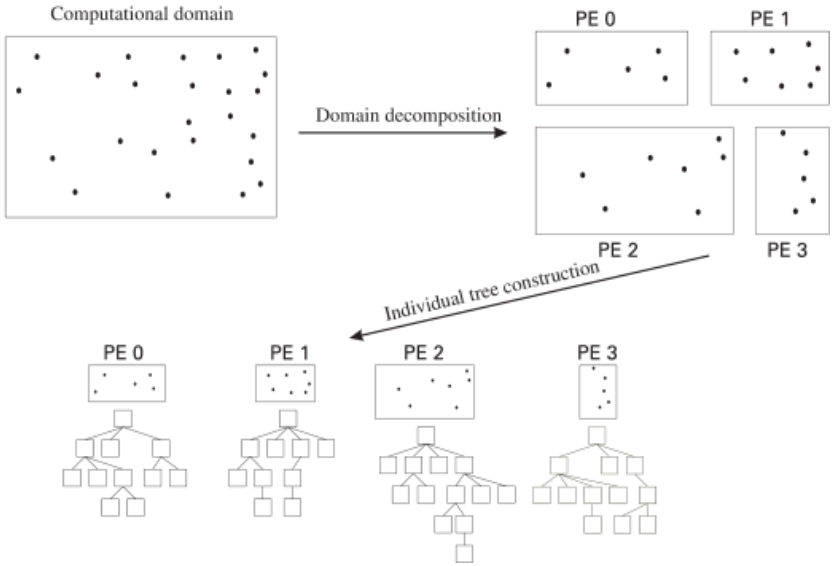
All to All

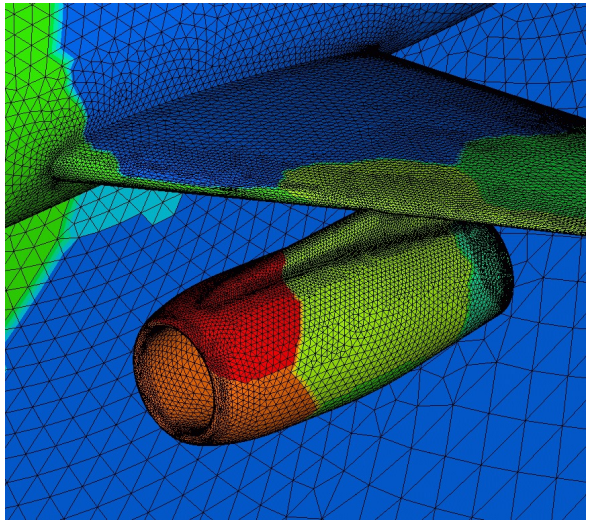






Domain Decomposition





Domain Decomposition

Challenge

- How to calculate in parallel using message passing and a distributed memory model?

Approach

- Divide the work among processes through domain decomposition
  - Divide calculations to individual processes → load-balancing of compute
  - Divide data to individual processes → load-balancing of memory
- Related goals
  - Maximise local computations (less data exchange)
  - Minimise communications and number of messages (less time lost in messaging)
  - Minimise sequential tasks (c.f. Amdahl's law)

Simple Domain Decomposition – Week 3

- If the problem can be trivially divided with no communication and independent computation it is *embarrassingly parallel* (e.g. SETI@home).
- If data size is not very large, but a lot of computation has to be done, it may be broadcast to all nodes.
- If tasks are almost independent they may be distributed by a "controller process" to other "worker processes". The controller collects results.

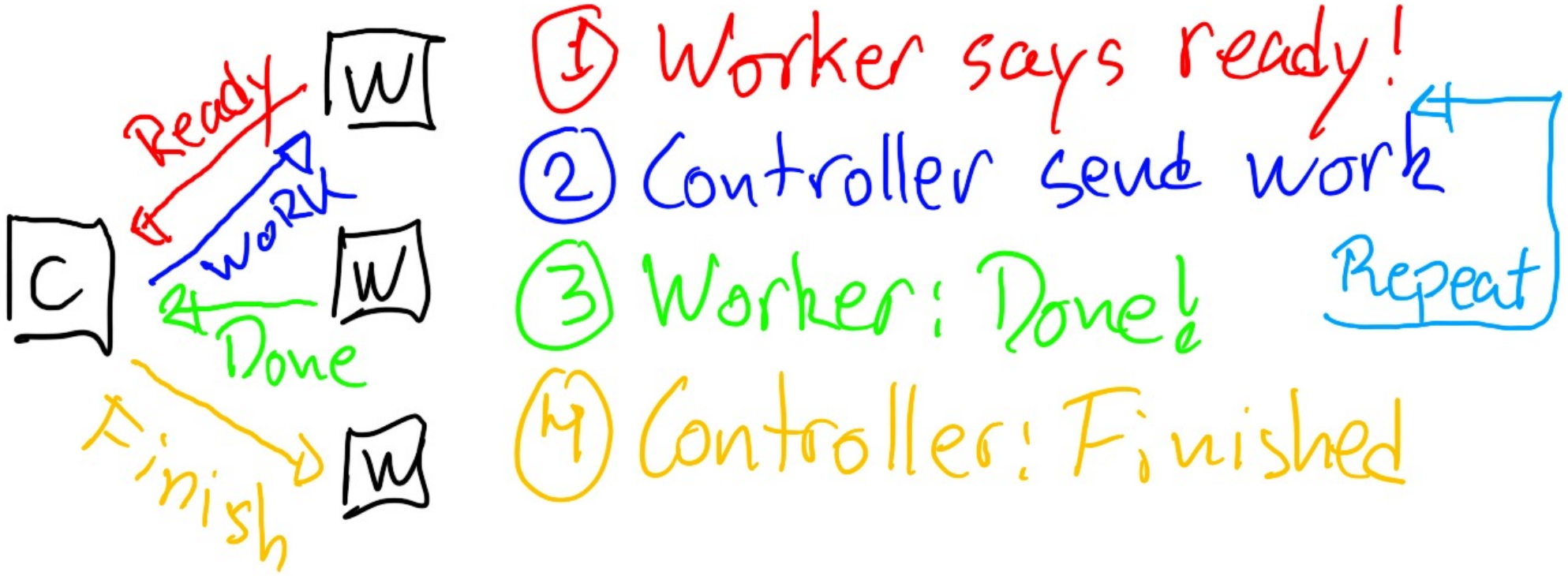
Examples:

- Monte-Carlo modelling
- Analysis of many similar data sets
- Data processing

Not really parallel in an HPC sense, but often still done on HPC platforms

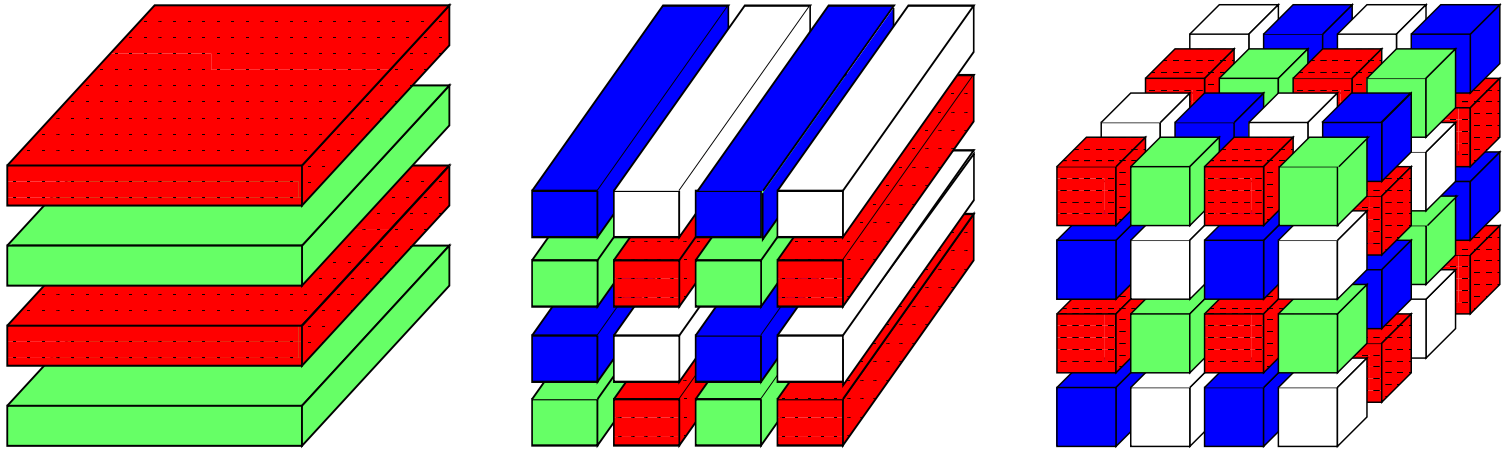
Controller – Worker with message passing

Easy to implement in e.g. Python with mpi4py



Domain Decomposition Mesh data

- Typical problem in physics:
  - Model a continuous physics process on a finite mesh
  - PDEs becomes finite difference equations
  - An example: shallow water model in the assignment
- Problem: dataset may be too large to copy to all processes; make a decomposition of the grid based on the local nature of the PDEs



Domain Decomposition Mesh data – Week 5

- PDEs encode transport and source contributions:

$$\frac{\partial u}{\partial t} = -g\frac{\partial \eta}{\partial x}, \qquad \frac{\partial v}{\partial t} = -g\frac{\partial \eta}{\partial y}, \qquad \frac{\partial \eta}{\partial t} = -\frac{\partial uh}{\partial x} - \frac{\partial vh}{\partial y}$$

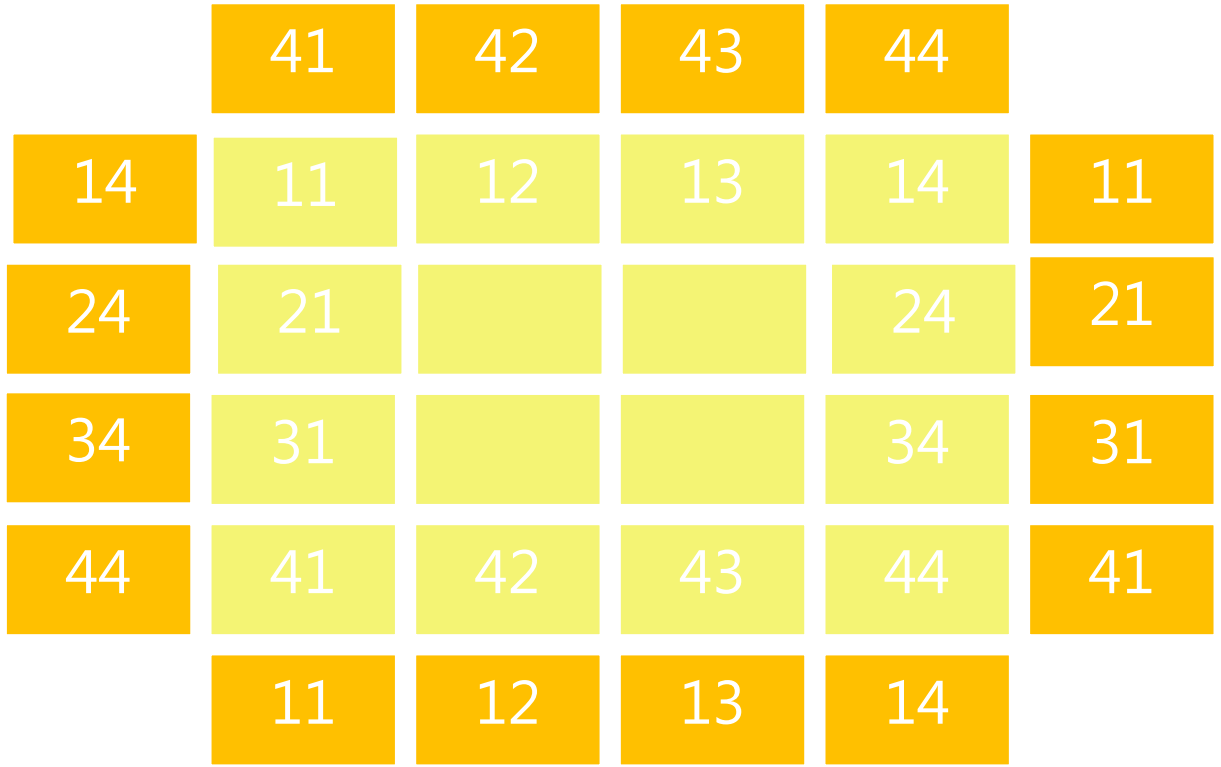
- When transformed in to a discrete version, differential operators translate in to dependencies between neighbouring cells

$$\eta_{i,j}^{n+1} = \eta_{i,j}^{n} - \Delta t\frac{u_{i,j}^{n+1} - u_{i,j-1}^{n+1}}{\Delta x} - \Delta t\frac{v_{i,j}^{n+1} - v_{i-1,j}^{n+1}}{\Delta y}$$

- To update the interior of a computational domain we therefore need access to guard zones (also called ghostzones, virtual boundaries, or halos)

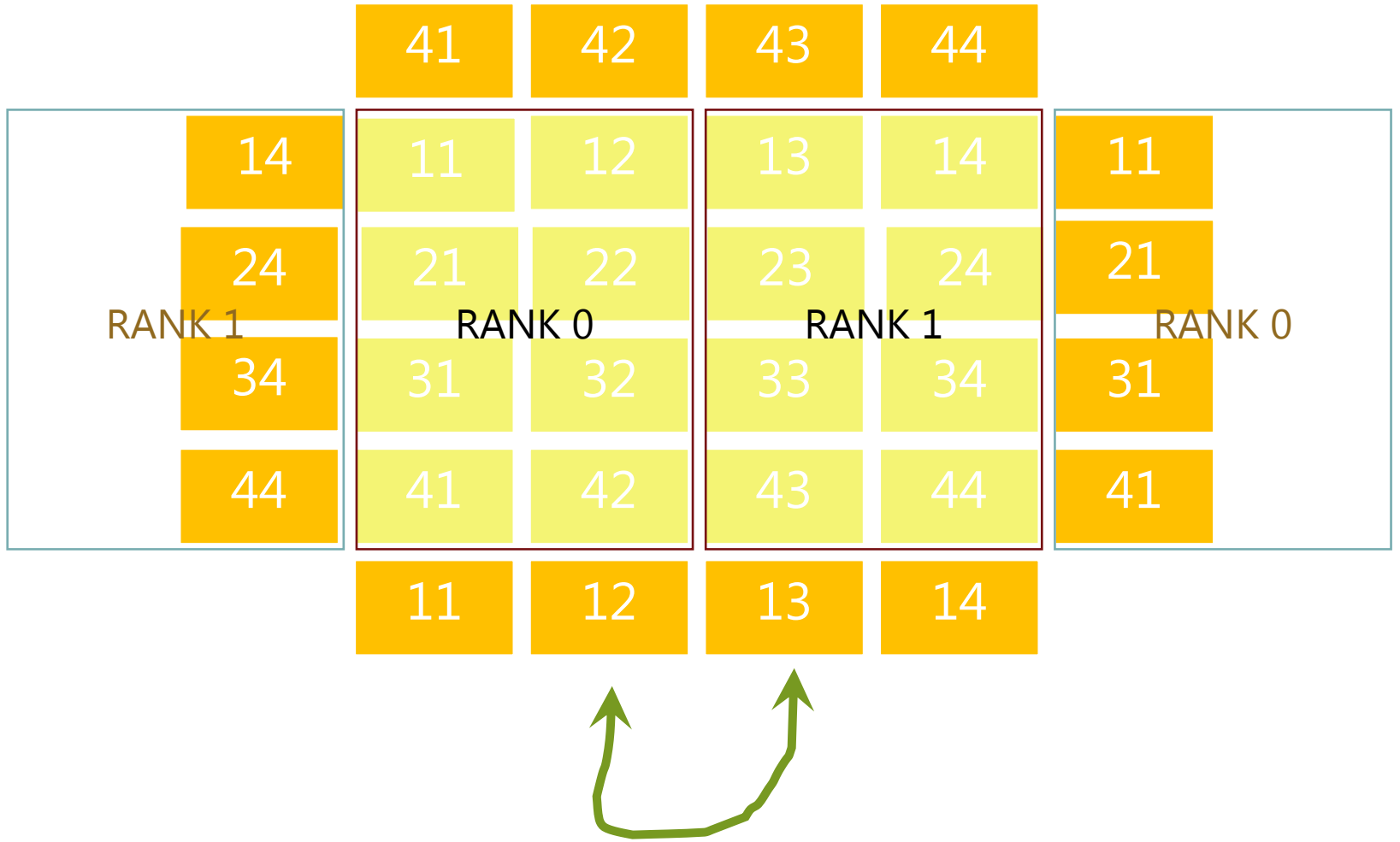
Domain Decomposition of Mesh data – periodic domain without MPI

- Simple example: 4x4 cells
- Domain is made periodic by adding two extra rows and columns at the N-S-E-W
- After each iteration we need to update data in the guard zones (dark orange) by copying from the physical domain

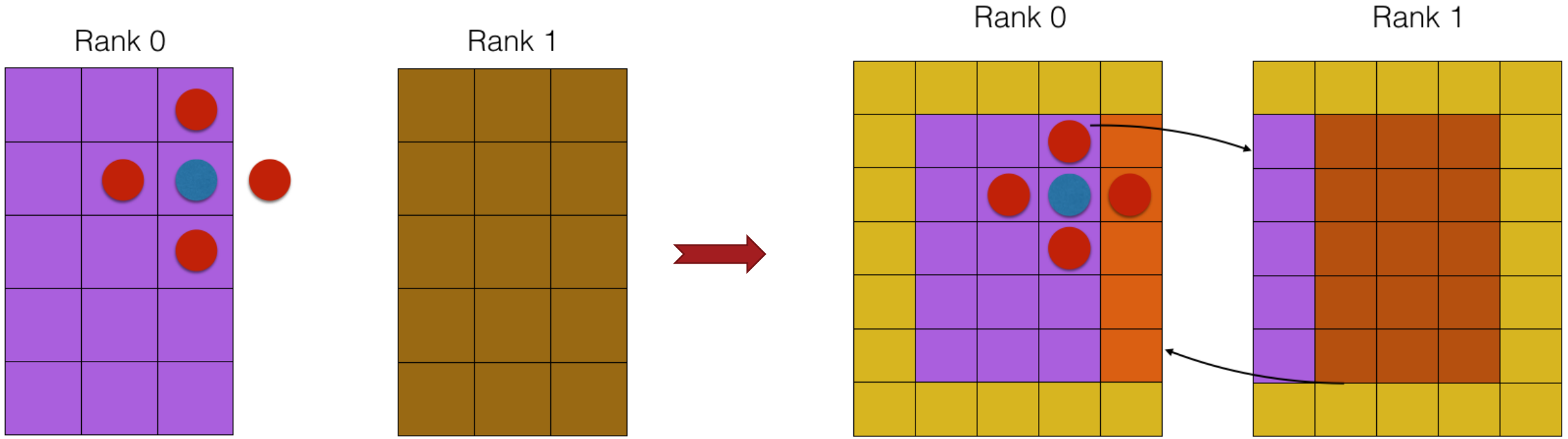


Domain Decomposition of Mesh data – periodic domain with MPI

- Split domain across two ranks
- Rank 0 needs cells 13,23,33,34 from Rank 1
- AND Rank 0 also needs cells 14,24,34,44 from Rank 1
- Slab geometry



Domain Decomposition Mesh data – with MPI

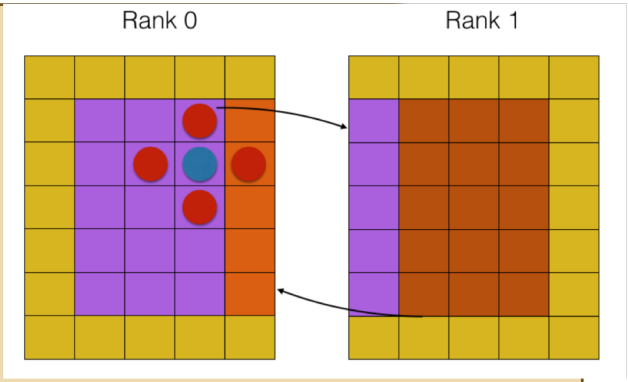


Diffusion operator: `temp[i][j] = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1])/4.0;`

This Week!

[credit: Brady and Ratcliffe]

Domain Decomposition: non-blocking MPI exchange



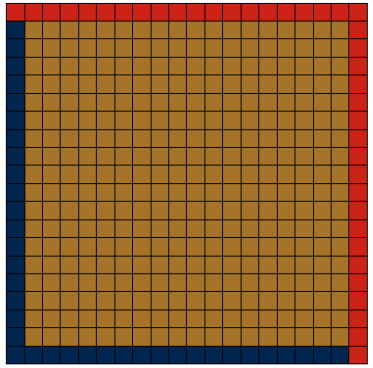

```
MPI_Init(&argc, &argv); // init MPI
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size); // number of ranks
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank); // my id
...
// Copy guard cells for other ranks to buffer
for (uint64_t i = 0; i < Ncols; ++i) {
  emit_left[i]  = data[i+Ncols];           // copy 2nd column to buffer
  emit_right[i] = data[i+(Nrows-2)*Ncols]; // copy 2nd last column to buffer
}
left = (mpi_size + mpi_rank-1) % mpi_size; right = mpi_rank+1 % mpi_size;
MPI_Isend(&emit_left[0], Ncols, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &req[0]);
MPI_Isend(&emit_right[0], Ncols, MPI_DOUBLE, right, 0, MPI_COMM_WORLD , &req[1]);
MPI_Irecv(&recv_left[0], Ncols, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &req[2]);
MPI_Irecv(&recv_right[0], Ncols, MPI_DOUBLE, right, 0, MPI_COMM_WORLD, &req[3]);
MPI_Waitall(4, &req[0], MPI_STATUSES_IGNORE);
// Copy buffers with guard cells from buffer to data array
for (uint64_t i = 0; i < Ncols; ++i) {
  data[i] = recv_left[i];                  // insert buffer in to 1st column
  data[i+(Nrows-1)*Ncols] = recv_right[i] // insert buffer in to last column
}
```

Domain Decomposition: higher dimensions – PHPC 8.5.2!

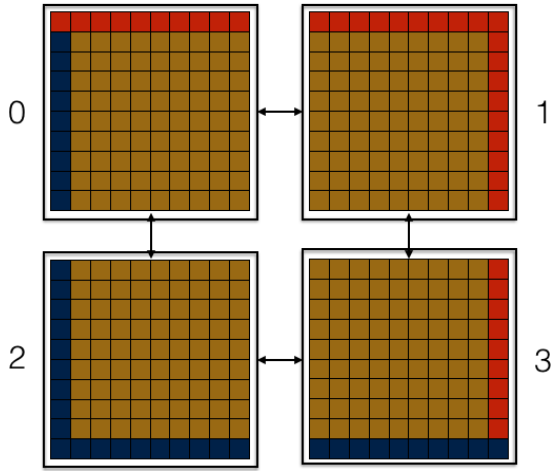
- MPI_Cart_* : routines for dealing with Cartesian topology
- MPI Graph_* : routines for dealing with arbitrary graph topologies

```
int ndims=2; int dims[2]={4,4}; int periodic[2]={1,1}; int reorder=1;
MPI_Comm comm2D; int shift=1; int mpi_dn[2]; int mpi_up[2]
MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periodic,reorder,&comm2D);
for (int i = 0; i < ndims; ++i)
  MPI_Cart_shift(comm2D,i,shift,&mpi_dn[i],&mpi_up[i]); // direction i neighbors
```

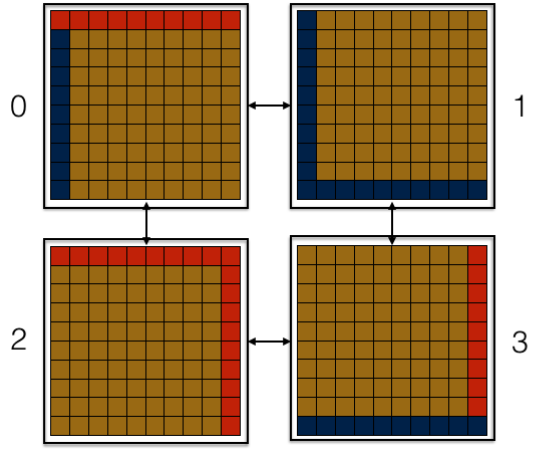
Use case: `MPI_Isend(&emit_left[0],Ncols,MPI_DOUBLE,mpi_dn[0],0,comm2D,&req[0]);`



2D Example with dims={2,2}



Works!

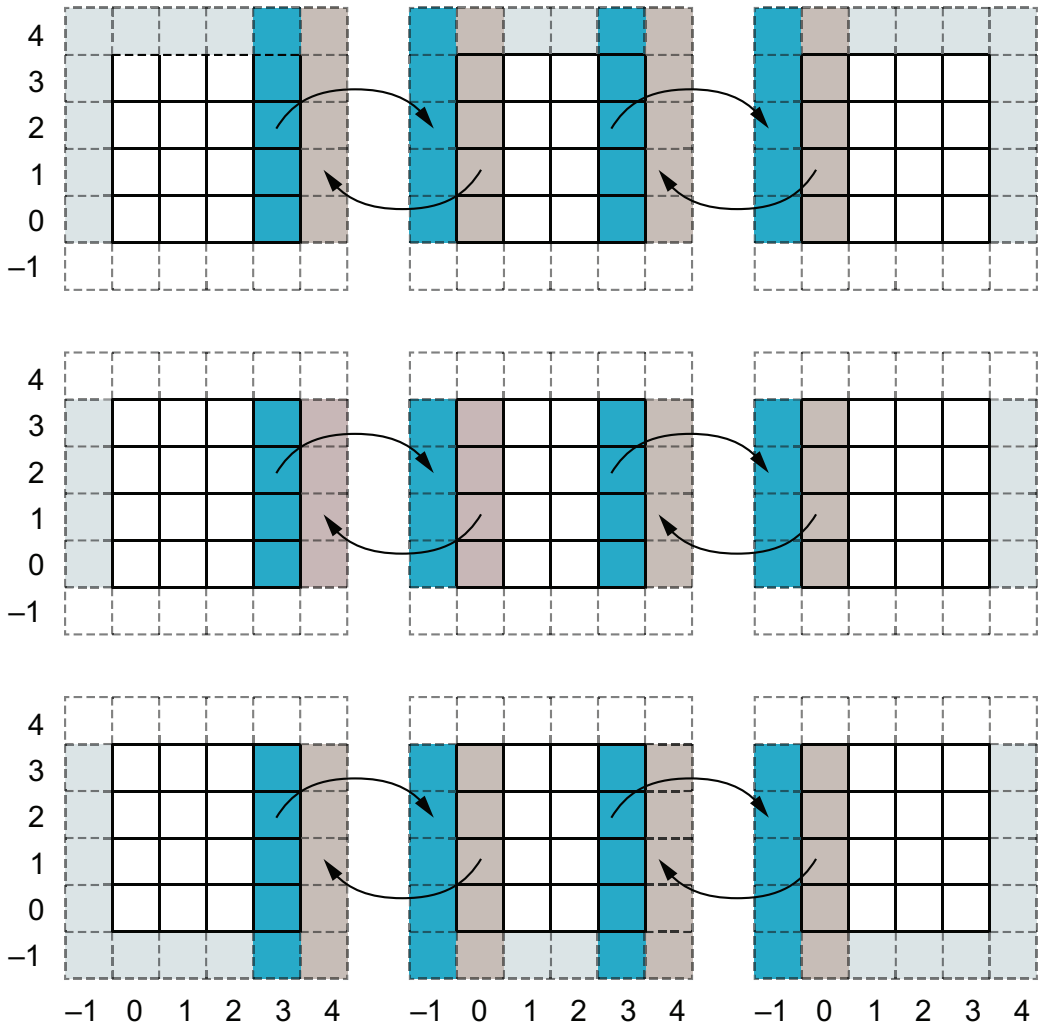


Wrong!

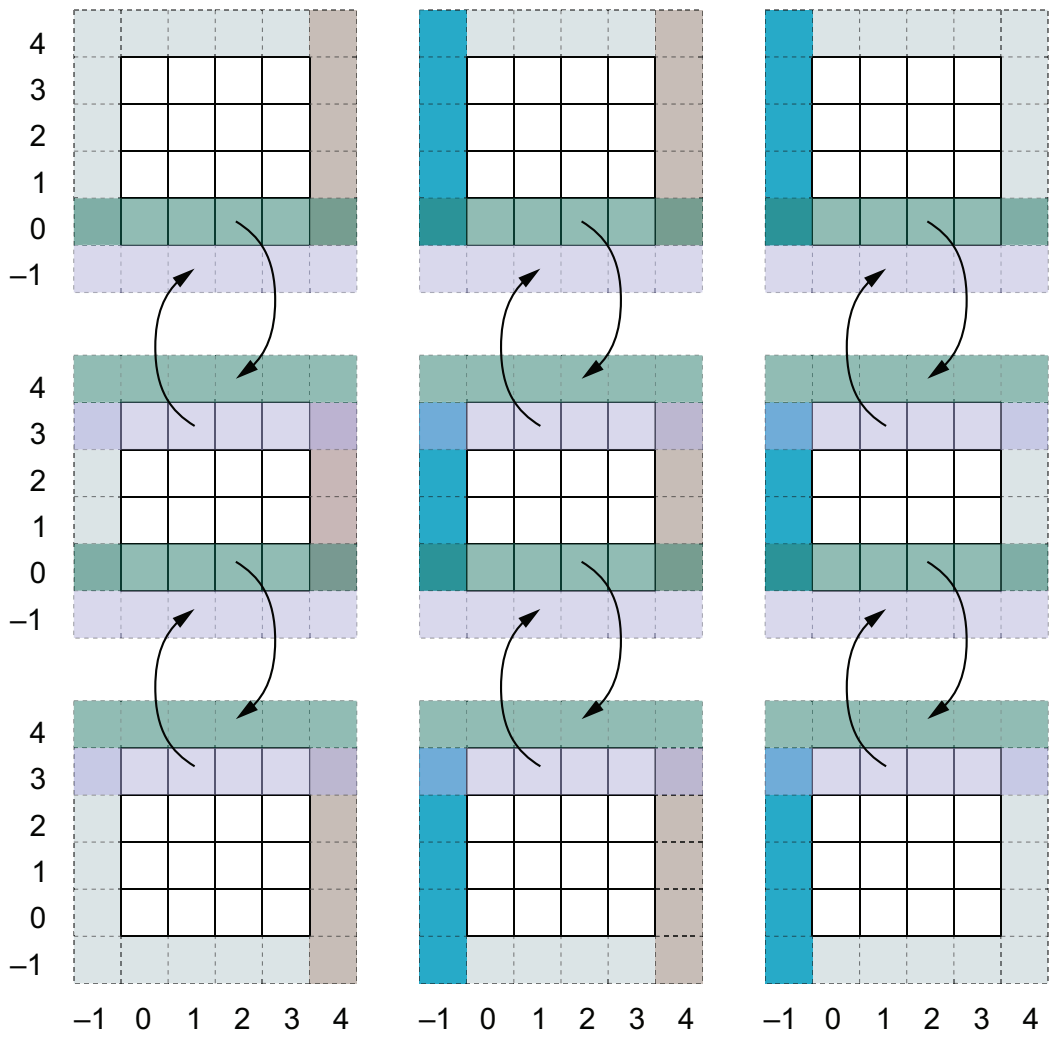
[figures: Brady and Ratcliffe]

Domain Decomposition: higher dimensions

1.



2.

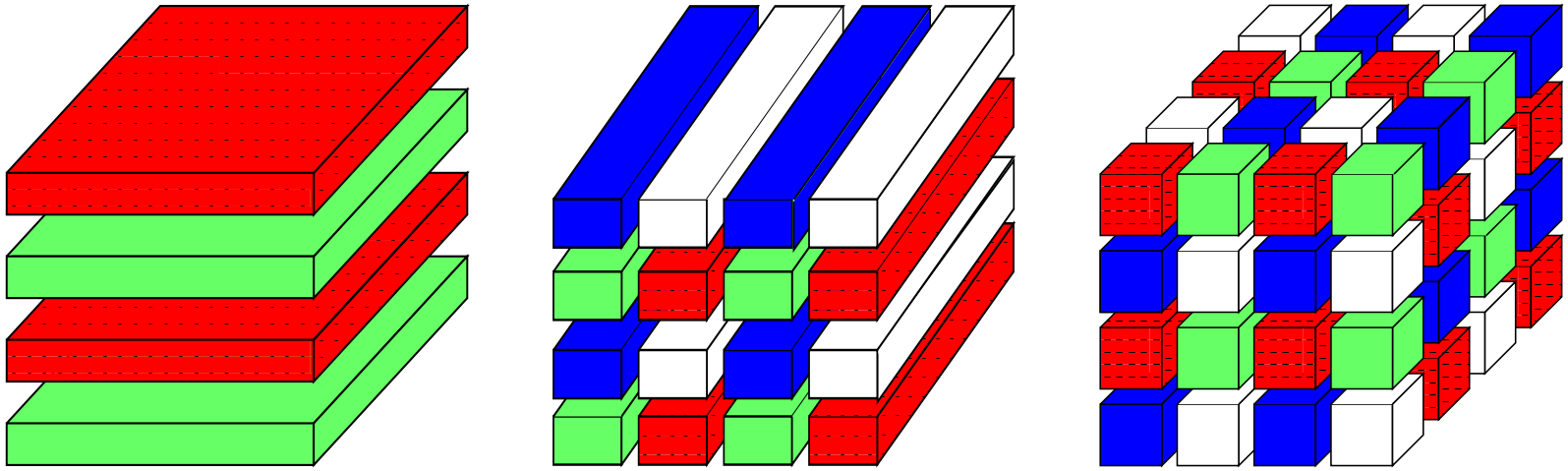


PHPC p 281 – Fig 8.5

Domain Decomposition Mesh data – summary

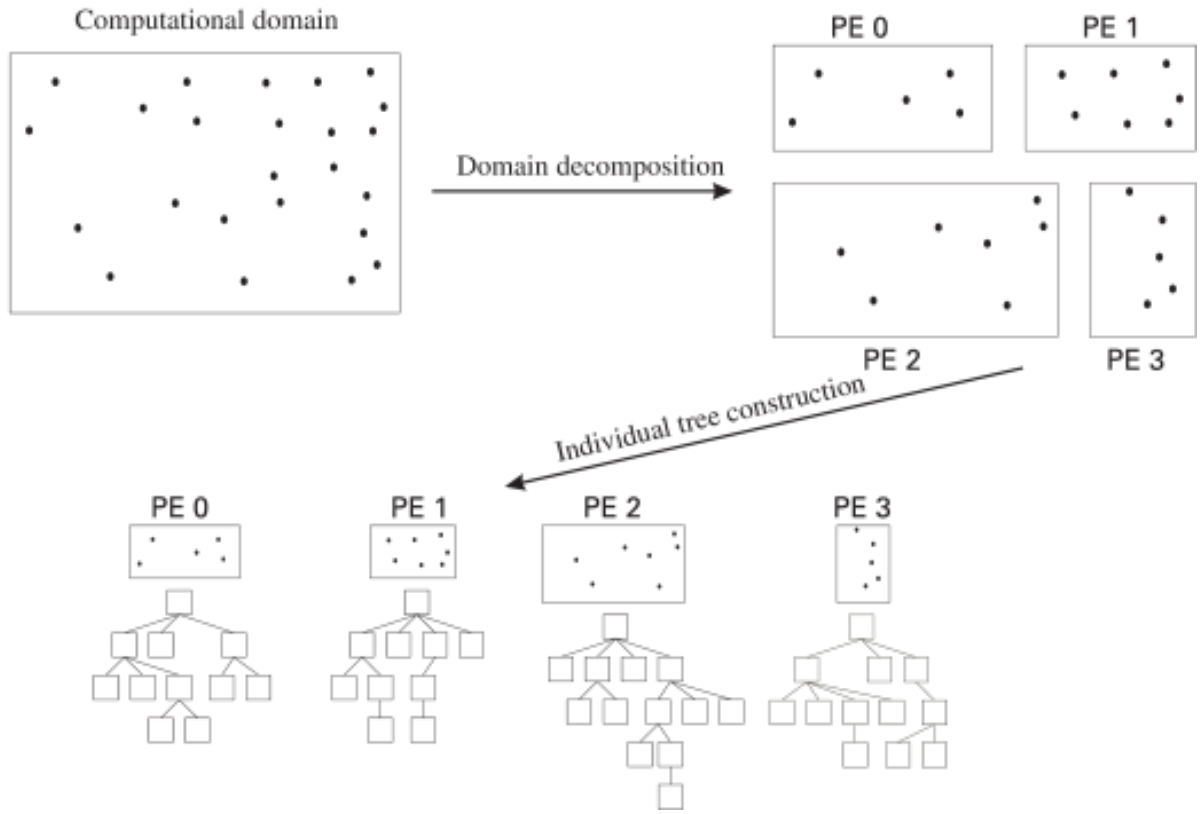
- If using slab geometry, just compute rank relations
- For higher dimensional geometry use MPI_Cart routines

- Question: given a mesh with size 2000x2000x2000 cells. What are the advantages of going beyond slab geometry?



Domain Decomposition: Going beyond

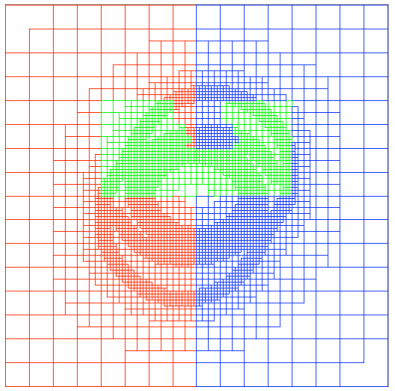
- Tree structures can be used to partition particle data
- Partition may be cut at branches, or using space-filling self-similar curves.
- Most used for oct trees are Peano-Hilbert and Morton curves



[fig: Springel

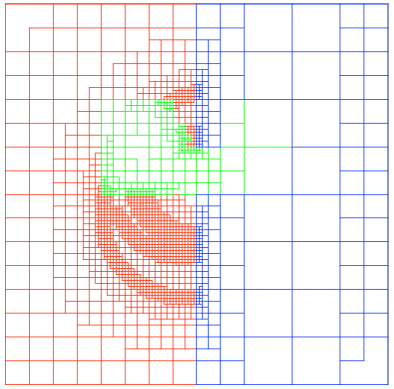
Domain Decomposition: Going beyond

- Tree structures can be used to partition particle data
- Partition may be cut at branches, or using space-filling self-similar curves.
- Most used for oct trees are Peano-Hilbert and Morton curves
- Neighbour relations more complicated and have to be calculated in the code
- BUT! Once calculated, cells communicate as either boundary (emit) or ghost (recv) as for cartesian

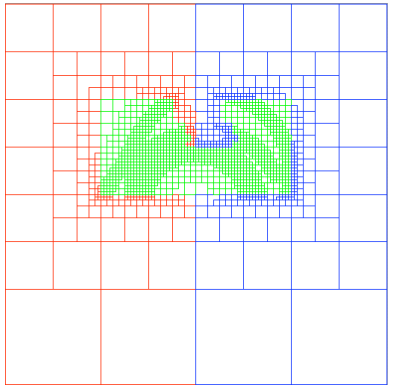


Domain decomposition over 3 processors

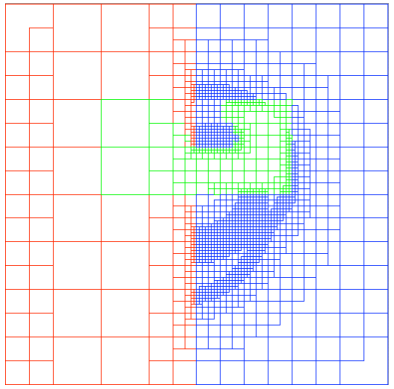
Each processor octree is surrounded by ghost cells (local copy of distant processor octrees) so that the resulting local octree contains all the necessary information.



Locally essential tree in processor #1



Locally essential tree in processor #2



Locally essential tree in processor #3

[fig: R. Teyssier]

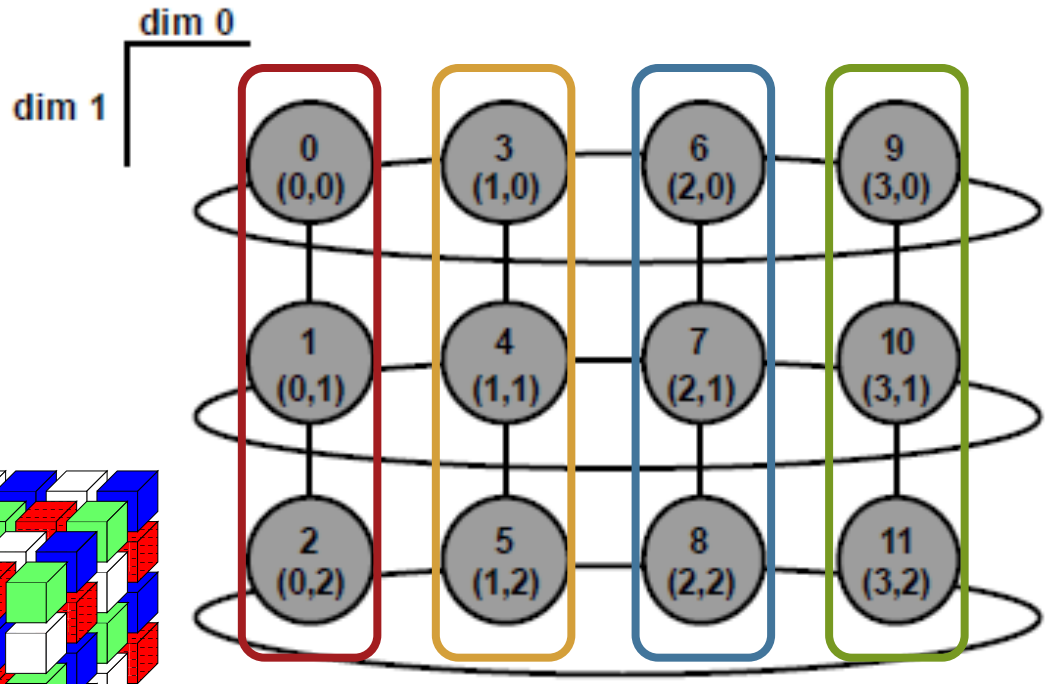
MPI Groups and Communicators

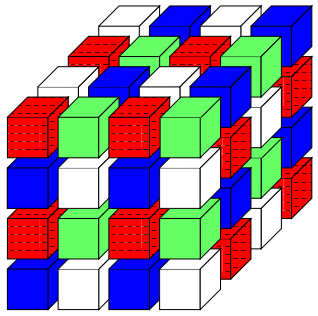
Groups and Communicators – PHPC 8.5.2 + 8.7

- MPI supports grouping processes together
- An *MPI group* designates a certain subset of processes
- An *MPI communicator* is the handle for communication inside a group (intra-communicator) or between two groups (inter-communicator)
- Default communicator for all processes: MPI_COMM_WORLD
- Another example: communicator created by MPI_Cart_create
- MPI Groups can be used to create communication topologies
- Collective communication only happens with relation to a communicator
- Groups are nice, but the important part is the *communicator*

MPI Group and Communicator Routines

- MPI_Comm_split: split a given communicator according to "colors".
- MPI_Comm_size: number of processes in communicator
- MPI_Comm_rank: rank of calling process in communicator





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

Example to make column communicator:

```
MPI_COMM_SPLIT(MPI_COMM_WORLD,
  rank / Ncol, rank,
  column_comm)
```

Useful for accumulating / projecting quantities along coordinate direction

Summary

- MPI exposes a powerful API that includes:
  - Point-to-point messages
  - Collective communication
  - Creation of abstract process groups
  - …and much more; Types and I/O covered Wednesday
- Proper Domain Decomposition important to maximise compute compared to communication.
- Proper Domain Decomposition important to be able to keep model in memory.
- For “slab geometry” we can use natural placement of ranks. For pencils and blocks, use MPI_Cart_* routines.
- MPI Communicators can be constructed to allow collective communication on sub-set of processes.

Essential MPI routines

Startup / close down / info routines

- `MPI_Init(int *argc,char ***argv)`: initialize the MPI library
- `MPI_Finalize(void)`: clean up the MPI state an deallocate buffers etc
- `MPI_Comm_size(MPI_Comm comm,int *size)`: number of ranks in communicator `comm`
- `MPI_Comm_rank(MPI_Comm comm,int *rank)`: rank of the calling process in communicator `comm`

Blocking point-to-point communication

- `MPI_Send(*buf,cnt,datatype,dest,tag,comm)`: send `buf` with `cnt` elements to rank `dest`
- `MPI_Recv(*buf,cnt,datatype,src,tag,comm, status)`: receive `buf` from rank `src`

Non-blocking point-to-point communication

- `MPI_Isend(*buf,cnt,datatype,dest,tag,comm,*req)`: send `buf` with `cnt` elements to rank `dest`
- `MPI_Irecv(*buf,cnt,datatype,src,tag,comm,*req)`: receive `buf` from rank `src`
- `MPI_Wait(*request,*status)`: Wait for the operation related to `req` to finish
- `MPI_Waitall(cnt,*array_of_requests,*array_of_statuses)`: Wait for cnt operations to finish

Essential MPI routines

Collective communication

- `MPI_Barrier(comm)`: all processes related to communicator `comm` wait for each other
- `MPI_Bcast(*buf,cnt,datatype,root,comm)`: send `buf` with `cnt` elements from `root` to everybody
- `MPI_Gather(*sendbuf,sendcnt,sendtype,*recvbuf,recvcnt,recvtype,root,comm)`: Gather `sendcnt` elements stored in `sendbuf` on root process. `recvcnt` indicates the number of elements received from a single process, not the total amount.
- `MPI_Scatter(*sendbuf,sendcnt,sendtype,*recvbuf,recvcnt,recvtype,root,comm)`: Inverse process of MPI_Gather. Scatter elements from root to everybody-
- `MPI_Reduce(*sendbuf,*recvbuf,cnt,datatype,operator,root,comm)`: Combine all elements in `sendbuf` on `root` process using `operator` (e.g. `MPI_SUM` or `MPI_MAX`) and store in `recvbuf`.
- `MPI_Allreduce(*sendbuf,*recvbuf,cnt,datatype,operator,comm)`: Combine all elements in `sendbuf` on using `operator` (e.g. `MPI_SUM` or `MPI_MAX`) and store in `recvbuf` *on all processes.*

Essential MPI variables / glossary

In the two preceding routine overview slides, some variables where repeated. Here is their type and meaning

- `MPI_Comm comm`: designates a communicator handle. `MPI_COMM_WORLD` is the default (everything)
- `MPI_Datatype datatype, sendtype, recvtype`: variable to indicate which type of data is being passed. Examples of often used types are: `MPI_Int`, `MPI_Double`, and `MPI_Float`
- `MPI_Request *req, array_of_requests`: handle for an outstanding (non-blocking) MPI routine
- `int src, dest, root`: Indicates a process number (a rank). Special values are `MPI_ANY_SOURCE` (can be used in e.g. MPI_Recv to receive from an arbitrary rank) and `MPI_PROC_NULL` (can be used in conjunction with non-periodic boundaries / special cases where no communication should be performed).
- `MPI_Status *status, *statuses`: argument to return details of the received message. Special values are `MPI_STATUS_IGNORE` (for status) and `MPI_STATUSES_IGNORE` for an array of statuses.

In the above, the first word is the variable type (e.g. `MPI_Comm`).

All MPI routines have a return value indicating successful completion. Therefore in all the examples things like

```
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
```

could have been written

```
int error = MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
```

and the error code could have been examined.